Principles of Soft Computing

(2nd Edition)

Dr. S. N. Sivanandam
Formerly Professor and Head
Department of Electrical and Electronics Engineering and
Department of Computer Science and Engineering,
PSG College of Technology,
Coimbatore

Dr. S. N. Deepa
Assistant Professor
Department of Electrical and Electronics Engineering,
Anna University of Technology, Coimbatore
Coimbatore

WILEY

About the Authors

Dr. S. N. **Sivanandam** completed his BE (Electrical and Electronics Engineering) in 1964 from Government College of Technology, Coimbatore, and MSc (Engineering) in Power System in 1966 from PSG College of Technology, Coimbatore. He acquired PhD in Control Systems in 1982 from Madras University. He received Best Teacher Award in the year 2001 and **Dhakshina Murthy Award** for Teaching Excellence from PSG College of Technology. He received The CITATION for best teaching and technical contribution in the Year 2002, Government College of Technology, Coimbatore. He has a total teaching experience (UG and PG) of 41 years. The total number of undergraduate and postgraduate projects guided by him for both Computer Science and Engineering and Electrical and Electronics Engineering is around 600. He has worked as **Professor and Head Computer Science and Engineering Department**, PSG College of Technology, Coimbatore. He has been identified as an outstanding person in the field of Computer Science and Engineering in MARQUIS 'Who's Who', October 2003 issue, USA. He has also been identified as an outstanding person in the field of Computational Science and Engineering in 'Who's Who', December 2005 issue, Saxe-Coburg Publications, UK. He has been placed as a VIP member in the continental WHO's WHO Registry of National Business Leaders, Inc., NY, August 24, 2006.

A widely published author, he has guided and coguided 30 PhD research works and at present 9 PhD research scholars are working under him. The total number of technical publications in International/National Journals/Conferences is around 700. He has also received Certificate of Merit 2005–2006 for his paper from The Institution of Engineers (India). He has chaired 7 International Conferences and 30 National Conferences. He is a member of various professional bodies like IE (India), ISTE, CSI, ACS and SSI. He is a technical advisor for various reputed industries and engineering institutions. His research areas include Modeling and Simulation, Neural Networks, Fuzzy Systems and Genetic Algorithm, Pattern Recognition, Multidimensional system analysis, Linear and Nonlinear control system, Signal and Image processing, Control System, Power system, Numerical methods, Parallel Computing, Data Mining and Database Security.

Dr. S. N. **Deepa** has completed her BE Degree from Government College of Technology, Coimbatore, 1999, ME Degree from PSG College of Technology, Coimbatore, 2004, and Ph.D. degree in Electrical Engineering from Anna University, Chennai, in the year 2008. She is currently Assistant Professor, Dept. of Electrical and Electronics Engineering, Anna University of Technology, Coimbatore. She was a gold medalist in her BE Degree Program. She has received G.D. Memorial Award in the year 1997 and Best Outgoing Student Award from PSG College of Technology, 2004. Her ME Thesis won National Award from the Indian Society of Technical Education and L&T, 2004. She has published 7 books and 32 papers in International and National Journals. Her research areas include Neural Network, Fuzzy Logic, Genetic Algorithm, Linear and Nonlinear Control Systems, Digital Control, Adaptive and Optimal Control.

Contents

1 Introduction

Learning Objectives

- Scope of soft computing.
- Various components under soft computing.
- Description on artificial neural networks with its advantages and applications.
- An overview of fuzzy logic.
- A note on genetic algorithm.
- The theory of hybrid systems.

1.1 Neural Networks

A neural network is a processing device, either an algorithm or an actual hardware, whose design was inspired by the design and functioning of animal brains and components thereof. The computing world has a lot to gain from neural networks, also known as artificial neural networks or neural net. The neural networks have the ability to learn by example, which makes them very flexible and powerful. For neural networks, there is no need to devise an algorithm to perform a specific task, that is, there is no need to understand the internal mechanisms of that task. These networks are also well suited for real-time systems because of their fast response and computational times which are because of their parallel architecture.

Before discussing artificial neural networks, let us understand how the human brain works. The human brain is an amazing processor. Its exact workings are still a mystery. The most basic element of the human brain is a specific type of cell, known as neuron, which doesn't regenerate. Because neurons aren't slowly replaced, it is assumed that they provide us with our abilities to remember, think and apply previous experiences to our every action. The human brain comprises about 100 billion neurons. Each neuron can connect with up to 200,000 other neurons, although 1,000–10,000 interconnections are typical.

The power of the human mind comes from the sheer numbers of neurons and their multiple interconnections. It also comes from genetic programming and learning. There are over 100 different classes of neurons. The individual neurons are complicated. They have a myriad of parts, subsystems and control mechanisms. They convey information via a host of electrochemical pathways. Together these neurons and their connections form a process which is not binary, not stable, and not synchronous. In short, it is nothing like the currently available electronic computers, or even artificial neural networks.

1.1.1 Artificial Neural Network: Definition

An artificial neural network (ANN) may be defined as an information-processing model that is inspired by the way biological nervous systems, such as the brain, process information. This model tries to replicate only the most basic functions of the brain. The key element of ANN is the novel structure of its information processing system. An ANN is composed of a large number of highly interconnected processing elements (neurons) working in unison to solve specific problems.

Artificial neural networks, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification through a learning process. In biological systems, learning involves adjustments to the synaptic connections that exist between the neurons. ANNs undergo a similar change that occurs when the concept on which they are built leaves the academic environment and is thrown into the harsher world of users who simply want to get a job done on computers accurately all the time. Many neural networks now being designed are statistically quite accurate, but they still leave their users with a bad taste as they falter when it comes to solving problems accurately. They might be 85–90% accurate. Unfortunately, few applications tolerate that level of error.

1.1.2 Advantages of Neural Networks

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, could be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network could be thought of as an "expert" in a particular category of information it has been given to analyze. This expert could be used to provide projections in new situations of interest and answer "what if" questions. Other advantages of working with an ANN include:

1. *Adaptive learning*: An ANN is endowed with the ability to learn how to do tasks based on the data given for training or initial experience.
2. *Self-organization*: An ANN can create its own organization or representation of the information it receives during learning time.
3. *Real-time operation*: ANN computations may be carried out in parallel. Special hardware devices are being designed and manufactured to take advantage of this capability of ANNs.
4. *Fault tolerance via redundant information coding*: Partial destruction of a neural network leads to the corresponding degradation of performance. However, some network capabilities may be retained even after major network damage.

Currently, neural networks can't function as a user interface which translates spoken words into instructions for a machine, but someday they would have this skill. Then VCRs, home security systems, CD players, and word processors would simply be activated by voice. Touch screen and voice editing would replace the word processors of today. Besides, spreadsheets and databases would be imparted such level of usability that would be pleasing to everyone. But for now, neural networks are only entering the marketplace in niche areas where their statistical accuracy is valuable.

Many of these niches indeed involve applications where answers provided by the software programs are not accurate but vague. Loan approval is one such area. Financial institutions make more money if they succeed in having the lowest bad loan rate. For these institutions, installing systems that are "90% accurate" in selecting the genuine loan applicants might be an improvement over their current selection process. Indeed, some banks have proved that the failure rate on loans approved by neural networks is lower than those approved by their

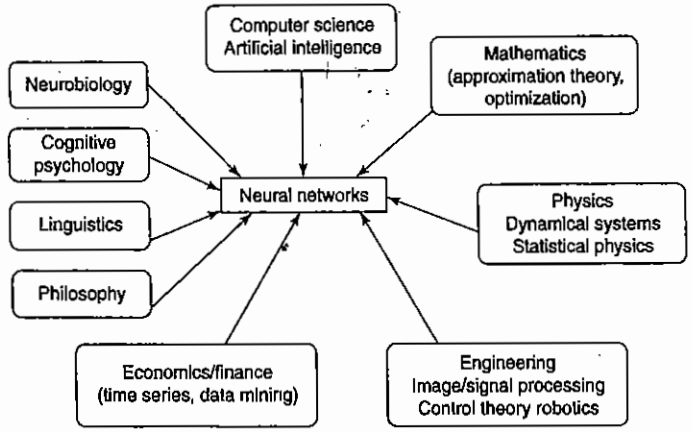


Figure 1-1 The multi-disciplinary point of view of neural networks.

best traditional methods. Also, some credit card companies are using neural networks in their application screening process.

This newest method of looking into the future by analyzing past experiences has generated its own unique set of problems. One such problem is to provide a reason behind a computer-generated answer, say, as to why a particular loan application was denied. To explain how a network learned and why it recommends a particular decision has been difficult. The inner workings of neural networks are "black boxes." Some people have even called the use of neural networks "voodoo engineering." To justify the decision-making process, several neural network tool makers have provided programs that explain which input through which node dominates the decision-making process. From this information, experts in the application may be able to infer which data plays a major role in decision-making and its importance.

Apart from filling the niche areas, neural network's work is also progressing in other more promising application areas. The next section of this chapter goes through some of these areas and briefly details the current work. The objective is to make the reader aware of various possibilities where neural networks might offer solutions, such as language processing, character recognition, image compression, pattern recognition, etc.

Neural networks can be viewed from a multi-disciplinary point of view as shown in Figure 1-1.

1.2 Application Scope of Neural Networks

The neural networks have good scope of being used in the following areas:

1. *Air traffic control* could be automated with the location, altitude, direction and speed of each radar blip taken as input to the network. The output would be the air traffic controller's instruction in response to each blip.
2. *Animal behavior, predator/prey relationships* and *population cycles* may be suitable for analysis by neural networks.
3. *Appraisal and valuation* of property, buildings, automobiles, machinery, etc. should be an easy task for a neural network.

4. *Betting* on horse races, stock markets, sporting events, etc. could be based on neural network predictions.
5. *Criminal sentencing* could be predicted using a large sample of crime details as input and the resulting sentences as output.
6. *Complex physical and chemical processes* that may involve the interaction of numerous (possibly unknown) mathematical formulas could be modeled heuristically using a neural network.
7. *Data mining, cleaning and validation* could be achieved by determining which records suspiciously diverge from the pattern of their peers.
8. *Direct mail advertisers* could use neural network analysis of their databases to decide which customers should be targeted, and avoid wasting money on unlikely targets.
9. *Echo patterns* from sonar, radar, seismic and magnetic instruments could be used to predict their targets.
10. *Econometric modeling* based on neural networks should be more realistic than older models based on classical statistics.
11. *Employee hiring* could be optimized if the neural networks were able to predict which job applicant would show the best job performance.
12. *Expert consultants* could package their intuitive expertise into a neural network to automate their services.
13. *Fraud detection* regarding credit cards, insurance or taxes could be automated using a neural network analysis of past incidents.
14. *Handwriting and typewriting* could be recognized by imposing a grid over the writing, then each square of the grid becomes an input to the neural network. This is called "Optical Character Recognition."
15. *Lake water levels* could be predicted based upon precipitation patterns and river/dam flows.
16. *Machinery control* could be automated by capturing the actions of experienced machine operators into a neural network.
17. *Medical diagnosis* is an ideal application for neural networks.
18. *Medical research* relies heavily on classical statistics to analyze research data. Perhaps a neural network should be included in the researcher's tool kit.
19. *Music composition* has been tried using neural networks. The network is trained to recognize patterns in the pitch and tempo of certain music, and then the network writes its own music.
20. *Photos and fingerprints* could be recognized by imposing a fine grid over the photo. Each square of the grid becomes an input to the neural network.
21. *Recipes and chemical formulations* could be optimized based on the predicted outcome of a formula change.
22. *Retail inventories* could be optimized by predicting demand based on past patterns.
23. *River water levels* could be predicted based on upstream reports, and time and location of each report.
24. *Scheduling of buses, airplanes and elevators* could be optimized by predicting demand.
25. *Staff scheduling* requirements for restaurants, retail stores, police stations, banks, etc., could be predicted based on the customer flow, day of week, paydays, holidays, weather, season, etc.
26. *Strategies* for games, business and war can be captured by analyzing the expert player's response to given stimuli. For example, a football coach must decide whether to kick, pass or run on the last down. The inputs for this decision include score, time, field location, yards to first down, etc.

27. *Traffic flows* could be predicted so that signal timing could be optimized. The neural network could recognize "a weekday morning rush hour during a school holiday" or "a typical winter Sunday morning."
28. *Voice recognition* could be obtained by analyzing the audio oscilloscope pattern, much like a stock market graph.
29. *Weather prediction* may be possible. Inputs would include weather reports from surrounding areas. Output(s) would be the future weather in specific areas based on the input information. Effects such as ocean currents and jet streams could be included.

Today, ANN represents a major extension to computation. Different types of neural networks are available for various applications. They perform operations akin to the human brain though to a limited extent. A rapid increase is expected in our understanding of the ANNs leading to the improved network paradigms and a host of application opportunities.

1.3 Fuzzy Logic

The concept of fuzzy logic (FL) was conceived by Lotfi Zadeh, a Professor at the University of California at Berkeley. An organized method for dealing with imprecise data is called fuzzy logic. The data are considered as fuzzy sets.

Professor Zadeh presented FL not as a control methodology but as a way of processing data by allowing partial set membership rather than crisp set membership or nonmembership. This approach to set theory was not applied to control systems until the 1970s due to insufficient computer capability. Also, earlier the systems were designed only to accept precise and accurate data. However, in certain systems it is not possible to get the accurate data. Therefore, Professor Zadeh reasoned that for processing need not always require precise and numerical information input; processing can be performed even with imprecise inputs. Suitable feedback controllers may be designed to accept noisy, imprecise input, and they would be much more effective and perhaps easier to implement. The processing with imprecise inputs led to the growth of Zadeh's FL. Unfortunately, US manufacturers have not been so quick to embrace this technology while the Europeans and Japanese have been aggressively building real products around it.

Fuzzy logic is a superset of conventional (or Boolean) logic and contains similarities and differences with Boolean logic. FL is similar to Boolean logic in that Boolean logic results are returned by FL operations when all fuzzy memberships are restricted to 0 and 1. FL differs from Boolean logic in that it is permissive of natural language queries and is more like human thinking; it is based on degrees of truth. For example, traditional sets include or do not include an individual element; there is no other case than true or false. However, fuzzy sets allow partial membership. FL is basically a multivalued logic that allows intermediate values to be defined between conventional evaluations such as *yes/no*, *true/false*, *black/white*, etc. Notions like rather warm or pretty cold can be formulated mathematically and processed with the computer. In this way, an attempt is made to apply a more human-like way of thinking in the programming of computers.

Fuzzy logic is a problem-solving control system methodology that lends itself to implementation in systems ranging from simple, small, embedded microcontrollers to large, networked, multichannel PC or workstation-based data acquisition and control systems. It can be implemented in hardware, software or a combination of both. FL provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy, or missing input information. FL's approach to control problems mimics how a person would make decisions, only much faster.

1.4 Genetic Algorithm

Genetic algorithm (GA) is reminiscent of sexual reproduction in which the genes of two parents combine to form those of their children. When it is applied to problem solving, the basic premise is that we can create an initial population of individuals representing possible solutions to a problem we are trying to solve. Each of these individuals has certain characteristics that make them more or less fit as members of the population. The more fit members will have a higher probability of mating and producing offspring that have a significant chance of retaining the desirable characteristics of their parents than the less fit members. This method is very effective at finding optimal or near-optimal solutions to a wide variety of problems because it does not impose many limitations required by traditional methods. It is an elegant generate-and-test strategy that can identify and exploit regularities in the environment; and results in solutions that are globally optimal or nearly so.

Genetic algorithms are adaptive computational procedures modeled on the mechanics of natural genetic systems. They express their ability by efficiently exploiting the historical information to speculate on new offspring with expected improved performance. GAs are executed iteratively on a set of coded solutions, called population, with three basic operators: selection/reproduction, crossover and mutation. They use only the payoff (objective function) information and probabilistic transition rules for moving to the next iteration. They are different from most of the normal optimization and search procedures in the following four ways:

1. GAs work with the coding of the parameter set, not with the parameter themselves;
2. GAs work simultaneously with multiple points, not a single point;
3. GAs search via sampling (a blind search) using only the payoff information;
4. GAs search using stochastic operators, not deterministic rules.

Since a GA works simultaneously on a set of coded solutions, it has very little chance to get stuck at local optima when used as optimization technique. Again, it does not need any sort of auxiliary information, like derivative of the optimizing function. Moreover, the resolution of the possible search space is increased by operating on coded (possible) solutions and not on the solutions themselves. Further, this search space need not be continuous. Recently, GAs are finding widespread applications in solving problems requiring efficient and effective search, in business, scientific and engineering circles like synthesis of neural network architectures, traveling salesman problem, graph coloring, scheduling, numerical optimization, and pattern recognition and image processing.

1.5 Hybrid Systems

Hybrid systems can be classified into three different systems: Neuro fuzzy hybrid system; neuron genetic hybrid system; fuzzy genetic hybrid systems. These are discussed in detail in the following sections.

1.5.1 Neuro Fuzzy Hybrid Systems

A neuro fuzzy hybrid system is a fuzzy system that uses a learning algorithm derived from or inspired by neural network theory to determine its parameters (fuzzy sets and fuzzy rules) by processing data samples.

In other words, a neuro fuzzy hybrid system refers to the combination of fuzzy set theory and neural networks having advantages of both which are listed below.

1. It can handle any kind of information (numeric, linguistic, logical, etc.).

2. It can manage imprecise, partial, vague or imperfect information.
3. It can resolve conflicts by collaboration and aggregation.
4. It has self-learning, self-organizing and self-tuning capabilities.
5. It doesn't need prior knowledge of relationships of data.
6. It can mimic human decision-making process.
7. It makes computation fast by using fuzzy number operations.

Neuro fuzzy hybrid systems combine the advantages of fuzzy systems, which deal with explicit knowledge that can be explained and understood, and neural networks, which deal with implicit knowledge that can be acquired by learning. Neural network learning provides a good way to adjust the knowledge of the expert (i.e., artificial intelligence system) and automatically generate additional fuzzy rules and membership functions to meet certain specifications. It helps reduce design time and costs. On the other hand, FL enhances the generalization capability of a neural network system by providing more reliable output when extrapolation is needed beyond the limits of the training data.

1.5.2 Neuro Genetic Hybrid Systems

Genetic algorithms (GAs) have been increasingly applied in ANN design in several ways: topology optimization, genetic training algorithms and control parameter optimization. In topology optimization, GA is used to select a topology (number of hidden layers, number of hidden nodes, interconnection pattern) for the ANN which in turn is trained using some training scheme, most commonly back propagation. In genetic training algorithms, the learning of an ANN is formulated as a weight optimization problem, usually using the inverse mean squared error as a fitness measure. Many of the control parameters such as learning rate, momentum rate, tolerance level, etc., can also be optimized using GAs. In addition, GAs have been used in many other innovative ways, to create new indicators based on existing ones, select good indicators, evolve optimal trading systems and complement other techniques such as fuzzy logic.

1.5.3 Fuzzy Genetic Hybrid Systems

The optimization abilities of GAs are used to develop the best set of rules to be used by a fuzzy inference engine, and to optimize the choice of membership functions. A particular use of GAs is in fuzzy classification systems, where an object is classified on the basis of the linguistic values of the object attributes. The most difficult part of building a system like this is to find the appropriate set of fuzzy rules. The most obvious approach is to obtain knowledge from experts and translate this into a set of fuzzy rules. But this approach is time consuming. Besides, experts may not be able to put their knowledge into an appropriate form of words. A second approach is to obtain the fuzzy rules through machine learning, whereby the knowledge is automatically extracted or deduced from sample cases. A fuzzy GA is a directed random search over all (discrete) fuzzy subsets of an interval and has features which make it applicable for solving this problem. It is capable of creating the classification rules for a fuzzy system where objects are classified by linguistic terms. Coding the rules genetically enables the system to deal with multivalue FL and is more efficient as it is consistent with numeric coding of fuzzy examples. The training data and randomly generated rules are combined to create the initial population, giving a better starting point for reproduction. Finally, a fitness function measures the strength of the rules, balancing the quality and diversity of the population.

1.6 Soft Computing

The two major problem-solving technologies include:

1. hard computing;
2. soft computing.

Hard computing deals with precise models where accurate solutions are achieved quickly. On the other hand, soft computing deals with approximate models and gives solution to complex problems. The two problem-solving technologies are shown in Figure 1-2.

Soft computing is a relatively new concept, the term really entering general circulation in 1994. The term "soft computing" was introduced by Professor Lotfi Zadeh with the objective of exploiting the tolerance for imprecision, uncertainty and partial truth to achieve tractability, robustness, low solution cost and better rapport with reality. The ultimate goal is to be able to emulate the human mind as closely as possible. Soft computing involves partnership of several fields, the most important being neural networks, GA, and FL. Also included is the field of probabilistic reasoning, employed for its uncertainty control techniques. However, this field is not examined here.

Soft computing uses a combination of GAs, neural networks and FL. A hybrid technique, in fact, would inherit all the advantages, but won't have the less desirable features of single soft computing components. It has to possess a good learning capacity, a better learning time than that of pure GAs and less sensitivity to the problem of local extremes than neural networks. In addition, it has to generate a fuzzy knowledge base, which has a linguistic representation and a very low degree of computational complexity.

An important thing about the constituents of soft computing is that they are complementary, not competitive, offering their own advantages and techniques to partnerships to allow solutions to otherwise unsolvable problems. The constituents of soft computing are examined in turn, following which existing applications of partnerships are described.

"Negotiation is the communication process of a group of agents in order to reach a mutually accepted agreement on some matter." This definition is typical of the research being done into negotiation and co-ordination in relation to software agents. It is an obvious necessity that when multiple agents interact, they will be required to co-ordinate their efforts and attempt to sort out any conflicts of resources or interest.

It is important to appreciate that agents are owned and controlled by people in order to complete tasks on their behalf. An example of a possible multiple-agent-based negotiation scenario is the competition between

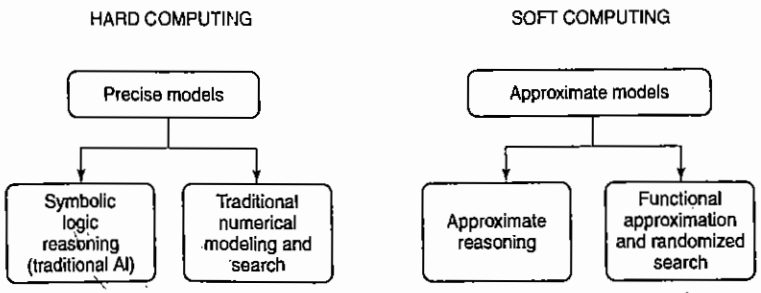


Figure 1-2 Problem-solving technologies.

long-distance phone call providers. When the consumer picks up the phone and dials, an agent will communicate on the consumer's behalf with all the available network providers. Each provider will make an offer that the consumer agent can accept or reject. A realistic goal would be to select the lowest available price for the call. However, given the first round of offers, network providers may wish to modify their offer to make it more competitive. The new offer is then submitted to the consumer agent and the process continues until a conclusion is reached. One advantage of this process is that the provider can dynamically alter its pricing strategy to account for changes in demand and competition, therefore maximizing revenue. The consumer will obviously benefit from the constant competition between providers. Best of all, the process is entirely autonomous as the agents embody and act on the beliefs and constraints of the parties they represent. Further changes can be made to the protocol so that providers can bid low without being in danger of making a loss. For example, if the consumer chooses to go with the lowest bid but pays the second lowest price, this will take away the incentive to underbid or overbid.

Much of the negotiation theory is based around human behavior models and, as a result, it is often translated using Distributed Artificial Intelligence techniques. The problems associated with machine negotiation are as difficult to solve as they are with human negotiation and involve issues such as privacy, security and deception.

1.7 Summary

The computing world has a lot to gain from neural networks whose ability to learn by example makes them very flexible and powerful. In case of neural networks, there is no need to devise an algorithm to perform a specific task, i.e., there is no need to understand the internal mechanisms of that task. Neural networks are also well suited for real-time systems because of their fast response and computational times, which are due to their parallel architecture.

Neural networks also contribute to other areas of research such as neurology and psychology. They are regularly used to model parts of living organisms and to investigate the internal mechanisms of the brain. Perhaps the most exciting aspect of neural networks is the possibility that someday "conscious" networks might be produced. Today, many scientists believe that consciousness is a "mechanical" property and that "conscious" neural networks are a realistic possibility.

Fuzzy logic was conceived as a better method for sorting and handling data but has proven to be an excellent choice for many control system applications since it mimics human control logic. It can be built into anything from small, hand-held products to large, computerized process control systems. It uses an imprecise but very descriptive language to deal with input data more like a human operator. It is robust and often works when first implemented with little or no tuning.

When applied to optimize ANNs for forecasting and classification problems, GAs can be used to search for the right combination of input data, the most suitable forecast horizon, the optimal or near-optimal network interconnection parterns and weights among the neurons, and the control parameters (learning rate, momentum rate, tolerance level, etc.) based on the training data used and the pre-set criteria. Like ANNs, GAs do not always guarantee you a perfect solution, but in many cases, you can arrive at an acceptable solution without the time and expense of an exhaustive search.

Soft computing is a relatively new concept, the term really entering general circulation in 1994, coined by Professor Lotfi Zadeh of the University of California, Berkeley, USA, it encompasses several fields of computing. The three that have been examined in this chapter are neural networks, FL and GAs. Neural networks are important for their ability to adapt and learn, FL for its exploitation of partial truth and imprecision, and GAs

for their application to optimization. The field of probabilistic reasoning is also sometimes included under the soft computing umbrella for its control of randomness and uncertainty. The importance of soft computing lies in using these methodologies in partnership – they all offer their own benefits which are generally not competitive and can therefore, work together. As a result, several hybrid systems were looked at – systems in which such partnerships exist.

2 Artificial Neural Network: An Introduction

Learning Objectives

- The fundamentals of artificial neural network.
- The evolution of neutal networks.
- Comparison between biological neuron and artificial neuron.
- Basic models of artificial neural networks.
- The different types of connections of neural networks, learning and activation functions are included.
- Various terminologies and notations used throughout the text.
- The basic fundamental neuron model – McCulloch–Pitts neuron and Hebb network.
- The concept of linear separability to form decision boundary regions.

2.1 Fundamental Concept

Neural networks are those information processing systems, which are constructed and implemented to model the human brain. The main objective of the neural network research is to develop a computational device for modeling the brain to perform various computational tasks at a faster rate than the traditional systems. Artificial neural networks perform various tasks such as pattern-matching and classification, optimization function, approximation, vector quantization, and data clustering. These tasks are very difficult for traditional computers, which are faster in algorithmic computational tasks and precise arithmetic operations. Therefore, for implementation of artificial neural networks, high-speed digital computers are used, which makes the simulation of neural processes feasible.

2.1.1 Artificial Neural Network

As already stated in Chapter 1, an artificial neural network (ANN) is an efficient information processing system which resembles in characteristics with a biological neural network. ANNs possess large number of highly interconnected processing elements called *nodes* or *units* or *neurons*, which usually operate in parallel and are configured in regular architectures. Each neuron is connected with the other by a connection link. Each connection link is associated with weights which contain information about the input signal. This information is used by the neuron net to solve a particular problem. ANNs' collective behavior is characterized by their ability to learn, recall and generalize training patterns or data similar to that of a human brain. They have the capability to model networks of original neurons as found in the brain. Thus, the ANN processing elements are called *neurons* or *artificial neurons*.

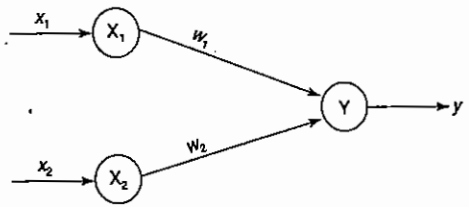


Figure 2-1 Architecture of a simple artificial neuron net.

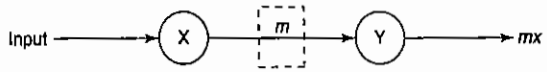


Figure 2-2 Neural net of pure linear equation.

It should be noted that each neuron has an internal state of its own. This internal state is called the *activation* or *activity level* of neuron, which is the function of the inputs the neuron receives. The activation signal of a neuron is transmitted to other neurons. Remember, a neuron can send only one signal at a time, which can be transmitted to several other neurons.

To depict the basic operation of a neural net, consider a set of neurons, say X_1 and X_2, transmitting signals to another neuron, Y. Here X_1 and X_2 are input neurons, which transmit signals, and Y is the output neuron, which receives signals. Input neurons X_1 and X_2 are connected to the output neuron Y, over a weighted interconnection links (W_1 and W_2) as shown in Figure 2-1.

For the above simple neuron net architecture, the net input has to be calculated in the following way:

$$y_{in} = + x_1 w_1 + x_2 w_2$$

where x_1 and x_2 are the activations of the input neurons X_1 and X_2, i.e., the output of input signals. The output y of the output neuron Y can be obtained by applying activations over the net input, i.e., the function of the net input:

$$y = f(y_{in})$$

Output = Function (net input calculated)

The function to be applied over the net input is called *activation function*. There are various activation functions, which will be discussed in the forthcoming sections. The above calculation of the net input is similar to the calculation of output of a pure linear straight line equation ($y = mx$). The neural net of a pure linear equation is as shown in Figure 2-2.

Here, to obtain the output y, the slope m is directly multiplied with the input signal. This is a linear equation. Thus, when slope and input are linearly varied, the output is also linearly varied, as shown in Figure 2-3. This shows that the weight involved in the ANN is equivalent to the slope of the linear straight line.

2.1.2 Biological Neural Network

It is well-known that the human brain consists of a huge number of neurons, approximately 10^{11}, with numerous interconnections. A schematic diagram of a biological neuron is shown in Figure 2-4.

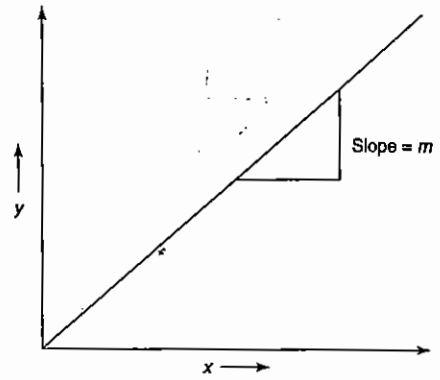


Figure 2-3 Graph for $y = mx$.

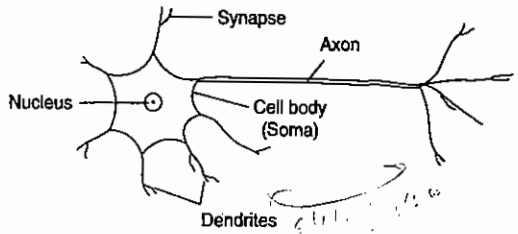


Figure 2-4 Schematic diagram of a biological neuron.

The biological neuron depicted in Figure 2-4 consists of three main parts:

1. *Soma* or *cell body* – where the cell nucleus is located.
2. *Dendrites* – where the nerve is connected to the cell body.
3. *Axon* – which carries the impulses of the neuron.

Dendrites are tree-like networks made of nerve fiber connected to the cell body. An axon is a single, long connection extending from the cell body and carrying signals from the neuron. The end of the axon splits into fine strands. It is found that each strand terminates into a small bulb-like organ called *synapse*. It is through synapse that the neuron introduces its signals to other nearby neurons. The receiving ends of these synapses on the nearby neurons can be found both on the dendrites and on the cell body. There are approximately 10^4 synapses per neuron in the human brain.

Electric impulses are passed between the synapse and the dendrites. This type of signal transmission involves a chemical process in which specific transmitter substances are released from the sending side of the junction. This results in increase or decrease in the electric potential inside the body of the receiving cell. If the electric potential reaches a threshold then the receiving cell fires and a *pulse* or *action potential* of fixed strength and duration is sent out through the axon to the synaptic junctions of the other cells. After firing, a cell has to wait for a period of time called the *refractory period* before it can fire again. The synapses are said to be *inhibitory* if they let passing impulses hinder the firing of the receiving cell or *excitatory* if they let passing impulses cause the firing of the receiving cell.

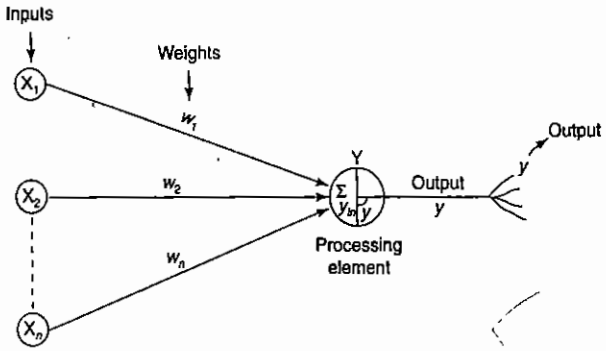


Figure 2-5 Mathematical model of artificial neuron.

Table 2-1 Terminology relationships between biological and artificial neurons

Biological neuron	Artificial neuron
Cell	Neuron
Dendrites	Weights or interconnections
Soma	Net input
Axon	Output

Figure 2-5 shows a mathematical representation of the above-discussed chemical processing taking place in an artificial neuron.

In this model, the net input is elucidated as

$$y_{in} = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n = \sum_{i=1}^{n} x_i w_i$$

where i represents the ith processing element. The activation function is applied over it to calculate the output. The weight represents the strength of synapse connecting the input and the output neurons. A positive weight corresponds to an excitatory synapse, and a negative weight corresponds to an inhibitory synapse.

The terms associated with the biological neuron and their counterparts in artificial neuron are presented in Table 2-1.

2.1.3 Brain vs. Computer – Comparison Between Biological Neuron and Artificial Neuron (Brain vs. Computer)

A comparison could be made between biological and artificial neurons on the basis of the following criteria:

1. *Speed:* The cycle time of execution in the ANN is of few nanoseconds whereas in the case of biological neuron it is of a few milliseconds. Hence, the artificial neuron modeled using a computer is more faster.

2. *Processing:* Basically, the biological neuron can perform massive parallel operations simultaneously. The artificial neuron can also perform several parallel operations simultaneously, but, in general, the artificial neuron network process is faster than that of the brain.
3. *Size and complexity:* The total number of neurons in the brain is about 10^{11} and the total number of interconnections is about 10^{15}. Hence, it can be noted that the complexity of the brain is comparatively higher, i.e. the computational work takes places not only in the brain cell body, but also in axon, synapse, etc. On the other hand, the size and complexity of an ANN is based on the chosen application and the network designer. The size and complexity of a biological neuron is more than that of an artificial neuron.
4. *Storage capacity (memory):* The biological neuron stores the information in its interconnections or in synapse strength but in an artificial neuron it is stored in its contiguous memory locations. In an artificial neuron, the continuous loading of new information may sometimes overload the memory locations. As a result, some of the addresses containing older memory locations may be destroyed. But in case of the brain, new information can be added in the interconnections by adjusting the strength without destroying the older information. A disadvantage related to brain is that sometimes its memory may fail to recollect the stored information whereas in an artificial neuron, once the information is stored in its memory locations, it can be retrieved. Owing to these facts, the adaprability is more toward an artificial neuron.
5. *Tolerance:* The biological neuron possesses fault tolerant capability whereas the artificial neuron has no fault tolerance. The distributed nature of the biological neurons enables to store and retrieve information even when the interconnections in them get disconnected. Thus biological neurons are fault tolerant. But in case of artificial neurons, the information gets corrupted if the network interconnections are disconnected. Biological neurons can accept redundancies, which is not possible in artificial neurons. Even when some cells die, the human nervous system appears to be performing with the same efficiency.
6. *Control mechanism:* In an artificial neuron modeled using a computer, there is a control unit present in Central Processing Unit, which can transfer and control precise scalar values from unit to unit, but there is no such control unit for monitoring in the brain. The strength of a neuron in the brain depends on the active chemicals present and whether neuron connections are strong or weak as a result of structure layer rather than individual synapses. However, the ANN possesses simpler interconnections and is free from chemical actions similar to those taking place in brain (biological neuron). Thus, the control mechanism of an artificial neuron is very simple compared to that of a biological neuron.

So, we have gone through a comparison between ANNs and biological neural networks. In short, we can say that an ANN possesses the following characteristics:

1. It is a neurally implemented mathematical model.
2. There exist a large number of highly interconnected processing elements called *neurons* in an ANN.
3. The interconnections with their weighted linkages hold the informative knowledge.
4. The input signals arrive at the processing elements through connections and connecting weights.
5. The processing elements of the ANN have the ability to learn, recall and generalize from the given data by suitable assignment or adjustment of weights.
6. The computational power can be demonstrated only by the collective behavior of neurons, and it should be noted that no single neuron carries specific information.

The above-mentioned characteristics make the ANNs as connectionist models, parallel distributed processing models, self-organizing systems, neuro-computing systems and neuro-morphic systems.

2.2 Evolution of Neural Networks

The evolution of neural networks has been facilitated by the rapid development of architectures and algorithms that are currently being used. The history of the development of neural networks along with the names of their designers is outlined Table 2-2.

In the later years, the discovery of the neural net resulted in the implementation of optical neural nets, Boltzmann machine, spatiotemporal nets, pulsed neural networks and support vector machines.

Table 2-2 Evolution of neural networks

Year	Neural network	Designer	Description
1943	McCulloch and Pitts neuron	McCulloch and Pitts	The arrangement of neurons in this case is a combination of logic functions. Unique feature of this neuron is the concept of threshold.
1949	Hebb network	Hebb	It is based upon the fact that if two neurons are found to be active simulraneously then the strength of the connection between them should be increased.
1958, 1959, 1962, 1988	Perceptron	Frank Rosenblatt, Block, Minsky and Papert	Here the weights on the connection path can be adjusted.
1960	Adaline	Widrow and Hoff	Here the weights are adjusted to reduce the difference between the net input to the output unit and the desired output. The result here is very negligible. Mean squared error is obtained.
1972	Kohonen self-organizing feature map	Kohonen	The concept behind this network is that the inputs are clustered together to obtain a fired output neuron. The clustering is performed by winner-take all policy.
1982, 1984, 1985, 1986, 1987	Hopfield network	John Hopfield and Tank	This neural network is based on fixed weights. These nets can also act as associative memory nets.
1986	Back-propagation network	Rumelhart, Hinton and Williams	This network is multi-layer with error being propagated backwards from the output units to the hidden units.
1988	Counter-propagation network	Grossberg	This network is similar to the Kohonen network; here the learning occurs for all units in a particular layer, and there exists no competition among these units.
1987–1990	Adaptive Resonance Theory (ART)	Carpenter and Grossberg	The ART network is designed for both binary inputs and analog valued inputs. Here the input patterns can be presented in any order.
1988	Radial basis function network	Broomhead and Lowe	This resembles a back propagation network but the activation function used is a Gaussian function.
1988	Neo cognitron	Fukushima	This network is essential for character recognition. The deficiency occurred in cognitron network (1975) was corrected by this network.

2.3 Basic Models of Artificial Neural Network

The models of ANN are specified by the three basic entities namely:

1. the model's synaptic interconnections;
2. the training or learning rules adopted for updating and adjusting the connection weights;
3. their activation functions.

2.3.1 Connections

The neurons should be visualized for their arrangements in layers. An ANN consists of a set of highly interconnected processing elements (neurons) such that each processing element output is found to be connected through weights to the other processing elements or to itself; delay lead and lag-free connections are allowed. Hence, the arrangements of these processing elements and the geometry of their interconnections are essential for an ANN. The point where the connection originates and terminates should be noted, and the function of each processing element in an ANN should be specified.

Besides the simple neuron shown in Figure ??, there exist several other types of neural network connections. The arrangement of neurons to form layers and the connection pattern formed within and between layers is called the *network architecture*. There exist five basic types of neuron connection architectures. They are:

1. single-layer feed-forward network;
2. multilayer feed-forward network;
3. single node with its own feedback;
4. single-layer recurrent network;
5. multilayer recurrent network.

Figures 2-6–2-10 depict the five types of neural network architectures. Basically, neural nets are classified into single-layer or multilayer neural nets. A layer is formed by taking a processing element and combining it with other processing elements. Practically, a layer implies a stage, going stage by stage, i.e., the input stage and the output stage are linked with each other. These linked interconnections lead to the formation of various network architectures. When a layer of the processing nodes is formed, the inputs can be connected to these

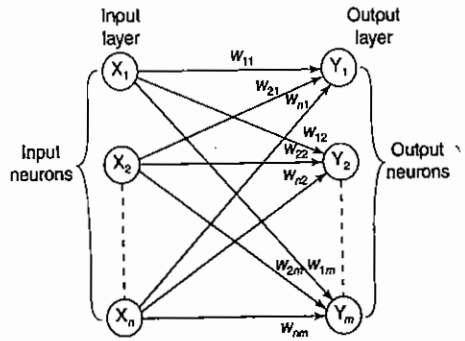


Figure 2-6 Single-layer feed-forward network.

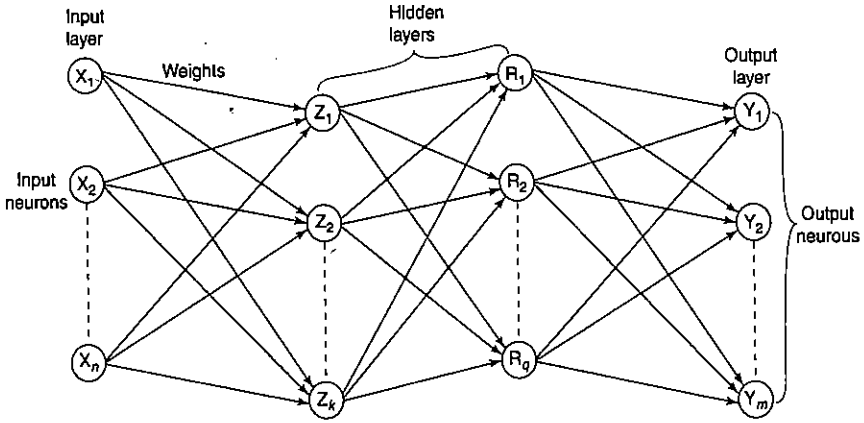


Figure 2-7 Multilayer feed-forward network.

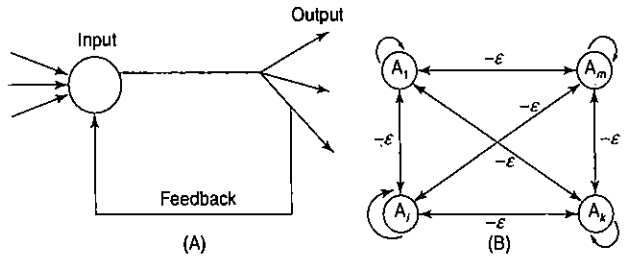


Figure 2-8 (A) Single node with own feedback. (B) Competitive nets.

nodes with various weights, resulting in a series of outputs, one per node. Thus, a single-layer *feed-forward network* is formed.

A multilayer feed-forward network (Figure 2-7) is formed by the interconnection of several layers. The input layer is that which receives the input and this layer has no function except buffering the input signal. The output layer generates the output of the network. Any layer that is formed between the input and output layers is called *hidden layer*. This hidden layer is internal to the network and has no direct contact with the external environment. It should be noted that there may be zero to several hidden layers in an ANN. More the number of the hidden layers, more is the complexity of the network. This may, however, provide an efficient output response. In case of a fully connected network, every output from one layer is connected to each and every node in the next layer.

A network is said to be a feed-forward network if no neuron in the output layer is an input to a node in the same layer or in the preceding layer. On the other hand, when outputs can be directed back as inputs to same or preceding layer nodes then it results in the formation of *feedback networks*.

If the feedback of the output of the processing elements is directed back as input to the processing elements in the same layer then it is called *lateral feedback*. Recurrent networks are feedback networks with closed loop. Figure 2-8(A) shows a simple recurrent neural network having a single neuron with

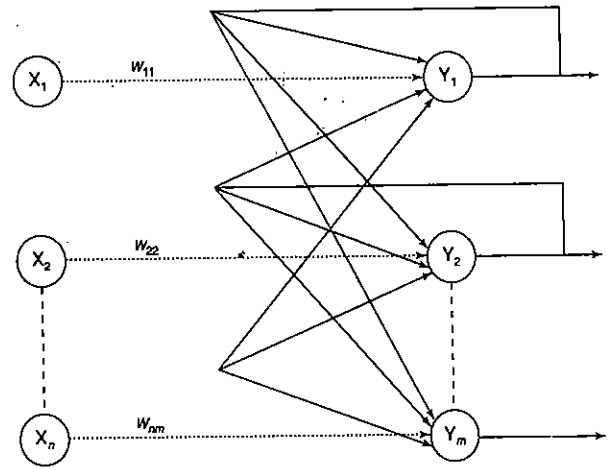


Figure 2-9 Single-layer recurrent network.

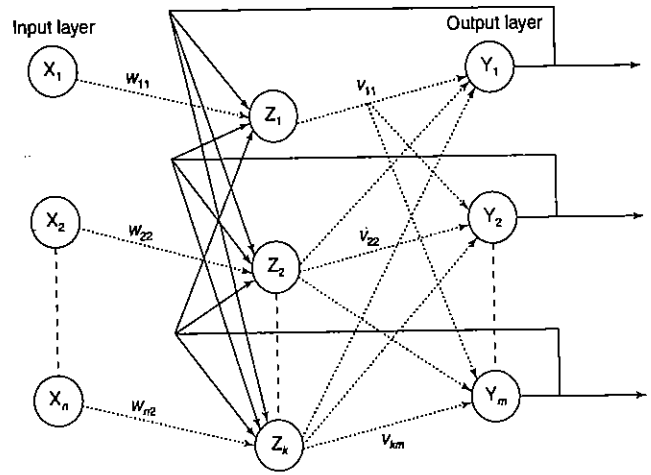


Figure 2-10 Multilayer recurrent network.

feedback to itself. Figure 2-9 shows a single-layer network with a feedback connection in which a processing element's output can be directed back to the processing element itself or to the other processing element or to both.

The architecture of a competitive layer is shown in Figure 2-8(B), the competitive interconnections having fixed weights of $-\varepsilon$. This net is called *Maxnet*, and will be discussed in the unsupervised learning network category. Apart from the network architectures discussed so far, there also exists another type of architecture with lateral feedback, which is called the *on-center-off-surround* or *lateral inhibition structure*. In this

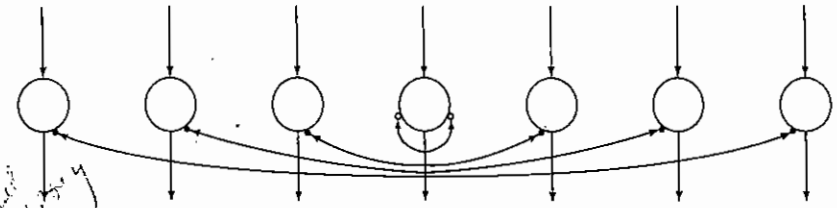


Figure 2-11 Lateral inhibition structure.

structure, each processing neuron receives two different classes of inputs – "excitatory" input from nearby processing elements and "inhibitory" inputs from more distantly located processing elements. This type of interconnection is shown in Figure 2-11.

In Figure 2-11, the connections with open circles are excitatory connections and the links with solid connective circles are inhibitory connections. From Figure 2-10, it can be noted that a processing element output can be directed back to the nodes in a preceding layer, forming a *multilayer recurrent network*. Also, in these networks, a processing element output can be directed back to the processing element itself and to other processing elements in the same layer. Thus, the various network architectures as discussed from Figures 2-6–2-11 can be suitably used for giving effective solution to a problem by using ANN.

2.3.2 Learning

The main property of an ANN is its capability to learn. Learning or training is a process by means of which a neural network adapts itself to a stimulus by making proper parameter adjustments, resulting in the production of desired response. Broadly, there are two kinds of learning in ANNs:

1. *Parameter learning:* It updates the connecting weights in a neural net.
2. *Structure learning:* It focuses on the change in network structure (which includes the number of processing elements as well as their connection types).

The above two types of learning can be performed simultaneously or separately. Apart from these two categories of learning, the learning in an ANN can be generally classified into three categories as: supervised learning; unsupervised learning; reinforcement learning. Let us discuss these learning types in detail.

2.3.2.1 *Supervised Learning*

The learning here is performed with the help of a teacher. Let us take the example of the learning process of a small child. The child doesn't know how to read/write. He/she is being taught by the parents at home and by the teacher in school. The children are trained and molded to recognize the alphabets, numerals, etc. Their each and every action is supervised by a teacher. Actually, a child works on the basis of the output that he/she has to produce. All these real-time events involve supervised learning methodology. Similarly, in ANNs following the supervised learning, each input vector requires a corresponding target vector, which represents the desired output. The input vector along with the target vector is called *training pair*. The network here is informed precisely about what should be emitted as output. The block diagram of Figure 2-12 depicts the working of a supervised learning network.

During training, the input vector is presented to the network, which results in an output vector. This output vector is the actual output vector. Then the actual output vector is compared with the desired (target) output vector. If there exists a difference between the two output vectors then an error signal is generated by

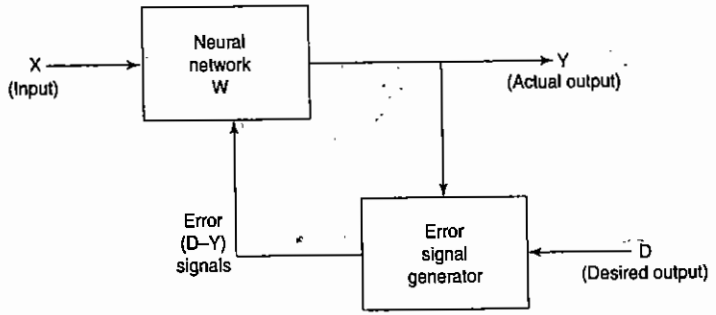


Figure 2-12 Supervised learning.

the network. This error signal is used for adjustment of weights until the actual output matches the desired (target) output. In this type of training, a supervisor or teacher is required for error minimization. Hence, the network trained by this method is said to be using supervised training methodology. In supervised learning, it is assumed that the correct "target" output values are known for each input pattern.

2.3.2.2 *Unsupervised Learning*

The learning here is performed without the help of a teacher. Consider the learning process of a tadpole, it learns by itself, that is, a child fish learns to swim by itself, it is not taught by its mother. Thus, its learning process is independent and is not supervised by a teacher. In ANNs following unsupervised learning, the input vectors of similar type are grouped without the use of training data to specify how a member of each group looks or to which group a number belongs. In the training process, the network receives the input patterns and organizes these patterns to form clusters. When a new input pattern is applied, the neural network gives an output response indicating the class to which the input pattern belongs. If for an input, a pattern class cannot be found then a new class is generated. The block diagram of unsupervised learning is shown in Figure 2-13.

From Figure 2-13 it is clear that there is no feedback from the environment to inform what the outputs should be or whether the outputs are correct. In this case, the network must itself discover patterns, regularities, features or categories from the input data and relations for the input data over the output. While discovering all these features, the network undergoes change in its parameters. This process is called *self-organizing* in which exact clusters will be formed by discovering similarities and dissimilarities among the objects.

2.3.2.3 *Reinforcement Learning*

This learning process is similar to supervised learning. In the case of supervised learning, the correct target output values are known for each input pattern. But, in some cases, less information might be available.

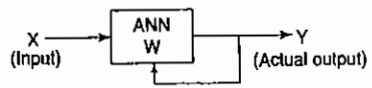


Figure 2-13 Unsupervised learning.

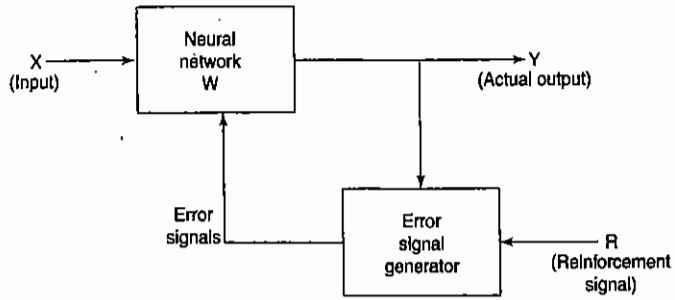


Figure 2-14 Reinforcement learning.

For example, the network might be told that its actual output is only "50% correct" or so. Thus, here only critic information is available, not the exact information. The learning based on this critic information is called *reinforcement learning* and the feedback sent is called *reinforcement signal*.

The block diagram of reinforcement learning is shown in Figure 2-14. The reinforcement learning is a form of supervised learning because the network receives some feedback from its environment. However, the feedback obtained here is only evaluative and not instructive. The external reinforcement signals are processed in the critic signal generator, and the obtained critic signals are sent to the ANN for adjustment of weights properly so as to get better critic feedback in future. The reinforcement learning is also called learning with a critic as opposed to learning with a teacher, which indicates supervised learning.

So, now you've a fair understanding of the three generalized learning rules used in the training process of ANNs.

2.3.3 Activation Functions

To better understand the role of the activation function, let us assume a person is performing some work. To make the work more efficient and to obtain exact output, some force or activation may be given. This activation helps in achieving the exact output. In a similar way, the activation function is applied over the net input to calculate the output of an ANN.

The information processing of a processing element can be viewed as consisting of two major parts: input and output. An integration function (say f) is associated with the input of a processing element. This function serves to combine activation, information or evidence from an external source or other processing elements into a net input to the processing element. The nonlinear activation function is used to ensure that a neuron's response is bounded – that is, the actual response of the neuron is conditioned or dampened as a result of large or small activating stimuli and is thus controllable.

Certain nonlinear functions are used to achieve the advantages of a multilayer network from a single-layer network. When a signal is fed through a multilayer network with linear activation functions, the output obtained remains same as that could be obtained using a single-layer network. Due to this reason, nonlinear functions are widely used in multilayer networks compared to linear functions.

There are several activation functions. Let us discuss a few in this section:

1. *Identity function*: It is a linear function and can be defined as

$$f(x) = x \quad \text{for all } x$$

The output here remains the same as input. The input layer uses the identity activation function.

2. *Binary step function*: This function can be defined as

$$f(x) = \begin{cases} 1 & \text{if } x \geq \theta \\ 0 & \text{if } x < \theta \end{cases}$$

where θ represents the threshold value. This function is most widely used in single-layer nets to convert the net input to an output that is a binary (1 or 0).

3. *Bipolar step function*: This function can be defined as

$$f(x) = \begin{cases} 1 & \text{if } x \geq \theta \\ -1 & \text{if } x < \theta \end{cases}$$

where θ represents the threshold value. This function is also used in single-layer nets to convert the net input to an output that is bipolar (+1 or −1).

4. *Sigmoidal functions*: The sigmoidal functions are widely used in back-propagation nets because of the relationship between the value of the functions at a point and the value of the derivative at that point which reduces the computational burden during training.
Sigmoidal functions are of two types:

- *Binary sigmoid function*: It is also termed as logistic sigmoid function or unipolar sigmoid function. It can be defined as

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

where λ is the steepness parameter. The derivative of this function is

$$f'(x) = \lambda f(x)[1 - f(x)]$$

Here the range of the sigmoid function is from 0 to 1.

- *Bipolar sigmoid function*: This function is defined as

$$f(x) = \frac{2}{1 + e^{-\lambda x}} - 1 = \frac{1 - e^{-\lambda x}}{1 + e^{-\lambda x}}$$

where λ is the steepness parameter and the sigmoid function range is between −1 and +1. The derivative of this function can be

$$f'(x) = \frac{\lambda}{2}[1 + f(x)][1 - f(x)]$$

The bipolar sigmoidal function is closely related to hyperbolic tangent function, which is written as

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

The derivative of the hyperbolic tangent function is

$$h'(x) = [1 + h(x)][1 - h(x)]$$

If the network uses a binary data, it is better to convert it to bipolar form and use the bipolar sigmoidal activation function or hyperbolic tangent function.

5. *Ramp function*: The ramp function is defined as

$$f(x) = \begin{cases} 1 & \text{if } x > 1 \\ x & \text{if } 0 \le x \le 1 \\ 0 & \text{if } x < 0 \end{cases}$$

The graphical representations of all the activation functions are shown in Figure 2-15(A)–(F).

2.4 Important Terminologies of ANNs

This section introduces you to the various terminologies related with ANNs.

2.4.1 Weights

In the architecture of an ANN, each neuron is connected to other neurons by means of directed communication links, and each communication link is associated with weights. The weights contain information about the input signal. This information is used by the net to solve a problem. The weight can be represented in terms of matrix. The weight matrix can also be called *connection matrix*. To form a mathematical notation, it is assumed that there are "n" processing elements in an ANN and each processing element has exactly "m" adaptive weights. Thus, the weight matrix W is defined by

$$\mathrm{W} = \begin{bmatrix} w_1^{\mathrm{T}} \\ w_2^{\mathrm{T}} \\ \vdots \\ w_n^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & \vdots & \cdots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{bmatrix}$$

where $w_i = [w_{i1}, w_{i2}, \ldots, w_{im}]^{\mathrm{T}}, i = 1, 2, \ldots, n$, is the weight vector of processing element and w_{ij} is the weight from processing element "i" (source node) to processing element "j" (destination node).

If the weight matrix W contains all the adaptive elements of an ANN, then the set of all W matrices will determine the set of all possible information processing configurations for this ANN. The ANN can be realized by finding an appropriate matrix W. Hence, the weights encode long-term memory (LTM) and the activation states of neurons encode short-term memory (STM) in a neural network.

2.4.2 Bias

The bias included in the network has its impact in calculating the net input. The bias is included by adding a component $x_0 = 1$ to the input vector X. Thus, the input vector becomes

$$\mathrm{X} = (1, \mathrm{X}_1, \ldots, \mathrm{X}_i, \ldots, \mathrm{X}_n)$$

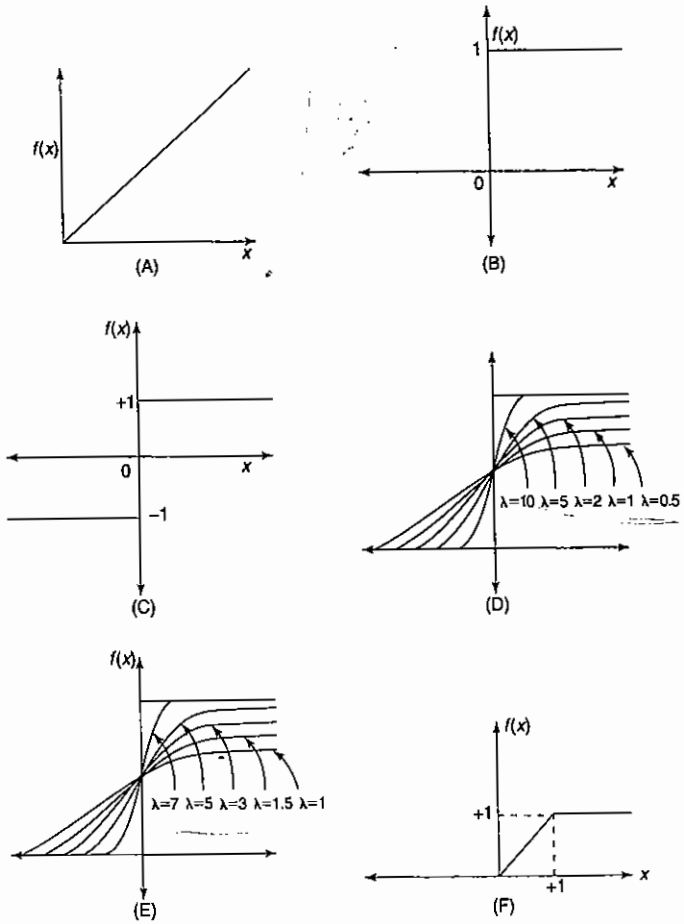


Figure 2-15 Depiction of activation functions: (A) identity function; (B) binary step function; (C) bipolar step function; (D) binary sigmoidal function; (E) bipolar sigmoidal function; (F) ramp function.

The bias is considered like another weight, that is, $w_{0j} = b_j$. Consider a simple network shown in Figure 2-16 with bias. From Figure 2-16, the net input to the output neuron Y_j is calculated as

$$y_{inj} = \sum_{i=0}^{n} x_i w_{ij} = x_0 w_{0j} + x_1 w_{1j} + x_2 w_{2j} + \cdots + x_n w_{nj}$$

$$= w_{0j} + \sum_{i=1}^{n} x_i w_{ij}$$

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

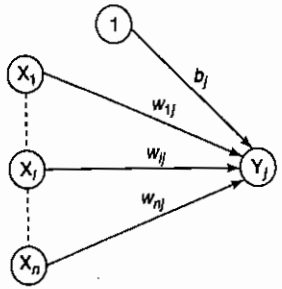


Figure 2-16 Simple net with bias.

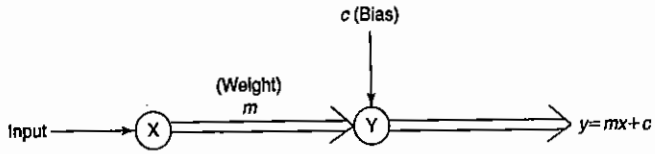


Figure 2-17 Block diagram for straight line.

The activation function discussed in Section 2.3.3 is applied over this net input to calculate the output. The bias can also be explained as follows: Consider an equation of straight line,

$$y = mx + c$$

where x is the input, m is the weight, c is the bias and y is the output. The equation of the straight line can also be represented as a block diagram shown in Figure 2-17. Thus, bias plays a major role in determining the output of the network.

The bias can be of two types: positive bias and negative bias. The positive bias helps in increasing the net input of the network and the negative bias helps in decreasing the net input of the network. Thus, as a result of the bias effect, the output of the network can be varied.

2.4.3 Threshold

Threshold is a set value based upon which the final output of the network may be calculated. The threshold value is used in the activation function. A comparison is made between the calculated net input and the threshold to obtain the network output. For each and every application, there is a threshold limit. Consider a direct current (DC) motor. If its maximum speed is 1500 rpm then the threshold based on the speed is 1500 rpm. If the motor is run on a speed higher than its set threshold, it may damage motor coils. Similarly, in neural networks, based on the threshold value, the activation functions are defined and the output is calculated. The activation function using threshold can be defined as

$$f(\text{net}) = \begin{cases} 1 & \text{if net} \geqslant \theta \\ -1 & \text{if net} < \theta \end{cases}$$

where θ is the fixed threshold value.

2.4.4 Learning Rate

The learning rate is denoted by "α." It is used to control the amount of weight adjustment at each step of training. The learning rate, ranging from 0 to 1, determines the rate of learning at each time step.

2.4.5 Momentum Factor

Convergence is made faster if a momentum factor is added to the weight updation process. This is generally done in the back propagation network. If momentum has to be used, the weights from one or more previous training patterns must be saved. Momentum helps the net in reasonably large weight adjustments until the corrections are in the same general direction for several patterns.

2.4.6 Vigilance Parameter

The vigilance parameter is denoted by "ρ." It is generally used in adaptive resonance theory (ART) network. The vigilance parameter is used to control the degree of similarity required for patterns to be assigned to the same cluster unit. The choice of vigilance parameter ranges approximately from 0.7 to 1 to perform useful work in controlling the number of clusters.

2.4.7 Notations

The notations mentioned in this section have been used in this textbook for explaining each network.

x_i: Activation of unit X_i, input signal.
y_j: Activation of unit Y_j, $y_j = f(y_{inj})$
w_{ij}: Weight on connection from unit X_i to unit Y_j.
b_j: Bias acting on unit j. Bias has a constant activation of 1.
W: Weight matrix, $W = \{w_{ij}\}$
y_{inj}: Net input to unit Y_j given by $y_{inj} = b_j + \sum_i x_i w_{ij}$

$\|x\|$: Norm of magnitude vector X.
θ_j: Threshold for activation of neuron Y_j.
S: Training input vector, $S = (s_1, \ldots, s_i, \ldots, s_n)$
T: Training output vector, $T = (t_1, \ldots, t_j, \ldots, t_n)$
X: Input vector, $X = (x_1, \ldots, x_i, \ldots, x_n)$
Δw_{ij}: Change in weights given by $\Delta w_{ij} = w_{ij}(\text{new}) - w_{ij}(\text{old})$
α: Learning rate; it controls the amount of weight adjustment at each step of training.

2.5 McCulloch–Pitts Neuron

2.5.1 Theory

The McCulloch–Pitts neuron was the earliest neural network discovered in 1943. It is usually called as *M–P neuron*. The M–P neurons are connected by directed weighted paths. It should be noted that the activation of a M–P neuron is binary, that is, at any time step the neuron may fire or may not fire. The weights associated with the communication links may be excitatory (weight is positive) or inhibitory (weight is negative). All the

excitatory connected weights entering into a particular neuron will have same weights. The threshold plays a major role in M–P neuron: There is a fixed threshold for each neuron, and if the net input to the neuron is greater than the threshold then the neuron fires. Also, it should be noted that any nonzero inhibitory input would prevent the neuron from firing. The M–P neurons are most widely used in the case of logic functions.

2.5.2 Architecture

A simple M–P neuron is shown in Figure 2-18. As already discussed, the M–P neuron has both excitatory and inhibitory connections. It is excitatory with weight ($w > 0$) or inhibitory with weight $-p(p < 0)$. In Figure 2-18, inputs from x_1 to x_n possess excitatory weighted connections and inputs from x_{n+1} to x_{n+m} possess inhibitory weighted interconnections. Since the firing of the output neuron is based upon the threshold, the activation function here is defined as

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq \theta \\ 0 & \text{if } y_{in} < \theta \end{cases}$$

For inhibition to be absolute, the threshold with the activation function should satisfy the following condition:

$$\theta > nw - p$$

The output will fire if it receives say "k" or more excitatory inputs but no inhibitory inputs, where

$$kw \geq \theta > (k-1)w$$

The M–P neuron has no particular training algorithm. An analysis has to be performed to determine the values of the weights and the threshold. Here the weights of the neuron are set along with the threshold to make the neuron perform a simple logic function. The M–P neurons are used as building blocks on which we can model any function or phenomenon, which can be represented as a logic function.

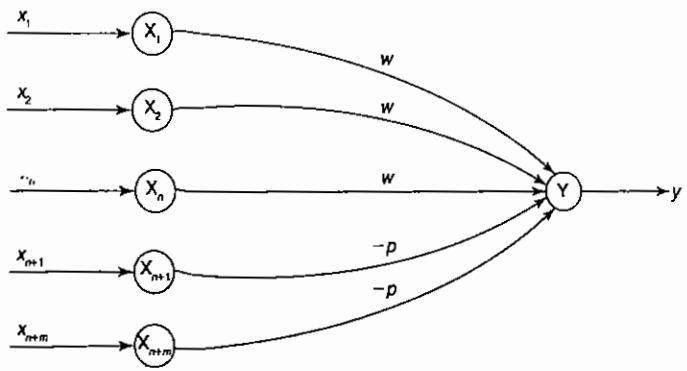


Figure 2-18 McCulloch–Pitts neuron model.

2.6 Linear Separability

An ANN does not give an exact solution for a nonlinear problem. However, it provides possible approximate solutions to nonlinear problems. Linear separability is the concept wherein the separation of the input space into regions is based on whether the network response is positive or negative.

A decision line is drawn to separate positive and negative responses. The decision line may also be called as the *decision-making line* or *decision-support line* or *linear-separable line*. The necessity of the linear separability concept was felt to classify the patterns based upon their output responses. Generally the net input calculated to the output unit is given as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

For example, if a bipolar step activation function is used over the calculated net input (y_{in}) then the value of the function is 1 for a positive net input and -1 for a negative net input. Also, it is clear that there exists a boundary between the regions where $y_{in} > 0$ and $y_{in} < 0$. This region may be called as *decision boundary* and can be determined by the relation

$$b + \sum_{i=1}^{n} x_i w_i = 0$$

On the basis of the number of input units in the network, the above equation may represent a line, a plane or a hyperplane. The linear separability of the network is based on the decision-boundary line. If there exist weights (with bias) for which the training input vectors having positive (correct) response, $+1$, lie on one side of the decision boundary and all the other vectors having negative (incorrect) response, -1, lie on the other side of the decision boundary then we can conclude the problem is "linearly separable."

Consider a single-layer network as shown in Figure 2-19 with bias included. The net input for the network shown in Figure 2-19 is given as

$$y_{in} = b + x_1 w_1 + x_2 w_2$$

The separating line for which the boundary lies between the values x_1 and x_2, so that the net gives a positive response on one side and negative response on other side, is given as

$$b + x_1 w_1 + x_2 w_2 = 0$$

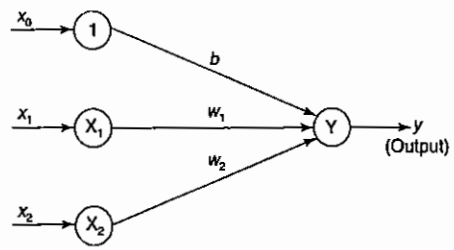


Figure 2-19 A single-layer neural net.

If weight w_2 is not equal to 0 then we get

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

Thus, the requirement for the positive response of the net is

$$b + x_1 w_1 + x_2 w_2 > 0$$

During training process, the values of w_1, w_2 and b are determined so that the net will produce a positive (correct) response for the training data. If on the other hand, threshold value is being used, then the condition for obtaining the positive response from output unit is

$$\text{Net input received} > \theta \text{ (threshold)}$$

$$y_{in} > \theta$$

$$x_1 w_1 + x_2 w_2 > \theta$$

The separating line equation will then be

$$x_1 w_1 + x_2 w_2 = \theta$$

$$x_2 = -\frac{w_1}{w_2}x_1 + \frac{\theta}{w_2} \quad (\text{with } w_2 \neq 0)$$

During training process, the values of w_1 and w_2 have to be determined, so that the net will have a correct response to the training data. For this correct response, the line passes close through the origin. In certain situations, even for correct response, the separating line does not pass through the origin.

Consider a network having positive response in the first quadrant and negative response in all other quadrants (AND function) with either binary or bipolar data, then the decision line is drawn separating the positive response region from the negative response region. This is depicted in Figure 2-20.

Thus, based on the conditions discussed above, the equation of this decision line may be obtained. Also, in all the networks that we would be discussing, the representation of data plays a major role.

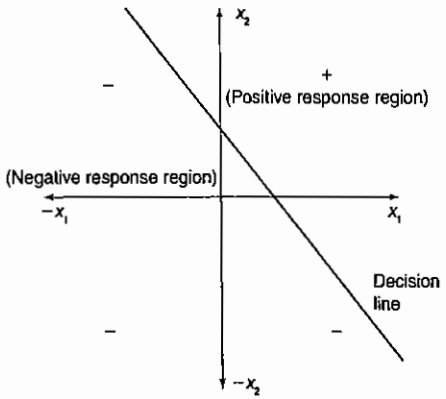


Figure 2-20 Decision boundary line.

However, the data representation mode has to be decided – whether it would be in binary form or in bipolar form. It may be noted that the bipolar representation is better than the binary representation. Using bipolar data representation, the missing data can be distinguished from mistaken data. Missing values are represented by 0 and mistakes can be represented by reversing the input value from +1 to −1 or vice-versa.

2.7 Hebb Network

2.7.1 Theory

For a neural net, the Hebb learning rule is a simple one. Let us understand it. Donald Hebb stated in 1949 that in the brain, the learning is performed by the change in the synaptic gap. Hebb explained it: "When an axon of cell A is near enough to excite cell B, and repeatedly or permanently takes place in firing it, some growth process or metabolic change takes place in one or both the cells such that A's efficiency, as one of the cells firing B, is increased."

According to the Hebb rule, the weight vector is found to increase proportionately to the product of the input and the learning signal. Here the learning signal is equal to the neuron's output. In Hebb learning, if two interconnected neurons are 'on' simultaneously then the weights associated with these neurons can be increased by the modification made in their synaptic gap (strength). The weight update in Hebb rule is given by

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$

The Hebb rule is more suited for bipolar data than binary data. If binary data is used, the above weight updation formula cannot distinguish two conditions namely:

1. A training pair in which an input unit is "on" and target value is "off."
2. A training pair in which both the input unit and the target value are "off."

Thus, there are limitations in Hebb rule application over binary data. Hence, the representation using bipolar data is advantageous.

2.7.2 Flowchart of Training Algorithm

The training algorithm is used for the calculation and adjustment of weights. The flowchart for the training algorithm of Hebb network is given in Figure 2-21. The notations used in the flowchart have already been discussed in Section 2.4.7.

In Figure 2-21, $s:t$ refers to each training input and target output pair. Till there exists a pair of training input and target output, the training process takes place; else, it is stopped.

2.7.3 Training Algorithm

The training algorithm of Hebb network is given below:

Step 0: First initialize the weights. Basically in this network they may be set to zero, i.e., $w_i = 0$ for $i = 1$ to n where "n" may be the total number of input neurons.

Step 1: Steps 2–4 have to be performed for each input training vector and target output pair, $s:t$.

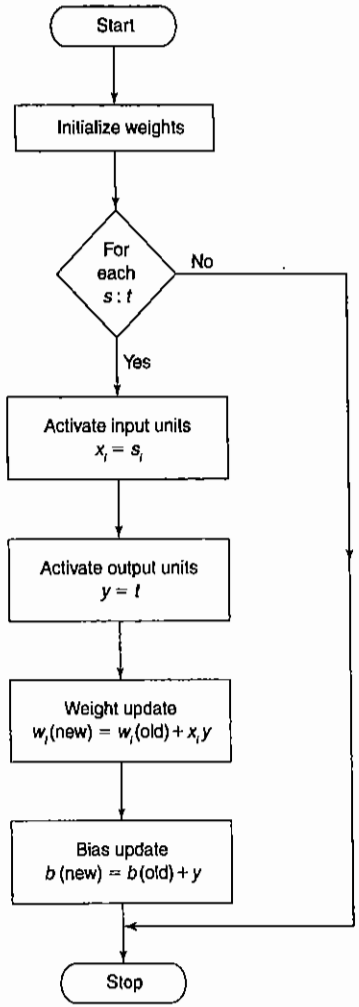


Figure 2-21 Flowchart of Hebb training algorithm.

Step 2: Input units activations are set. Generally, the activation function of input layer is identity function:

$x_i = s_i$ for $i = 1$ to n.

Step 3: Output units activations are set: $y = t$.

Step 4: Weight adjustments and bias adjustments are performed:

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$
$$b(\text{new}) = b(\text{old}) + y$$

The above five steps complete the algorithmic process. In Step 4, the weight updation formula can also be given in vector form as

$$w(\text{new}) = w(\text{old}) + xy$$

Here the change in weight can be expressed as

$$\Delta w = xy$$

As a result,

$$w(\text{new}) = w(\text{old}) + \Delta w$$

The Hebb rule can be used for pattern association, pattern categorization, pattern classification and over a range of other areas.

2.8 Summary

In this chapter we have discussed the basics of an ANN and its growth. A detailed comparison between biological neuron and artificial neuron has been included to enable the reader understand the basic difference between them. An ANN is constructed with few basic building blocks. The building blocks are based on the models of artificial neurons and the topology of few basic structures. Concepts of supervised learning, unsupervised learning and reinforcement learning are briefly included in this chapter. Various activation functions and different types of layered connections are also considered here. The basic terminologies of ANN are discussed with their typical values. A brief description on McCulloch–Pitts neuron model is provided. The concept of linear separability is discussed and illustrated with suitable examples. Details are provided for the effective training of a Hebb network.

2.9 Solved Problems

1. For the network shown in Figure 1, calculate the net input to the output neuron.

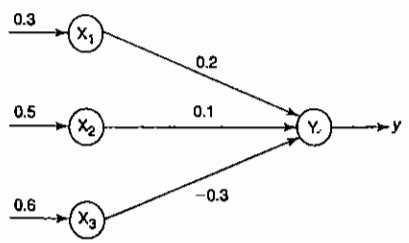


Figure 1 Neural net.

Solution: The given neural net consists of three input neurons and one output neuron. The inputs and weights are

$$[x_1, x_2, x_3] = [0.3, 0.5, 0.6]$$
$$[w_1, w_2, w_3] = [0.2, 0.1, -0.3]$$

The net input can be calculated as

$$y_{in} = x_1 w_1 + x_2 w_2 + x_3 w_3$$
$$= 0.3 \times 0.2 + 0.5 \times 0.1 + 0.6 \times (-0.3)$$
$$= 0.06 + 0.05 - 0.18 = -0.07$$

2. Calculate the net input for the network shown in Figure 2 with bias included in the network.

Solution: The given net consists of two input neurons, a bias and an output neuron. The inputs are

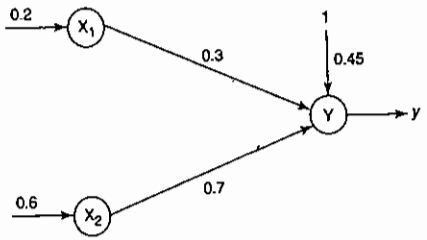


Figure 2 Simple neural net.

$[x_1, x_2] = [0.2, 0.6]$ and the weights are $[w_1, w_2] = [0.3, 0.7]$. Since the bias is included $b = 0.45$ and bias input x_0 is equal to 1, the net input is calculated as

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0.45 + 0.2 \times 0.3 + 0.6 \times 0.7$$
$$= 0.45 + 0.06 + 0.42 = 0.93$$

Therefore $y_{in} = 0.93$ is the net input.

3. Obtain the output of the neuron Y for the network shown in Figure 3 using activation functions as: (i) binary sigmoidal and (ii) bipolar sigmoidal.

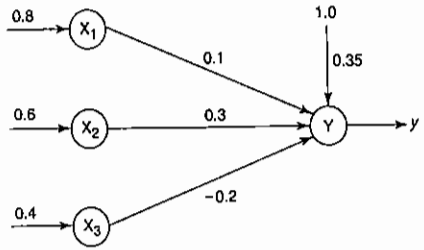


Figure 3 Neural net.

Solution: The given network has three input neurons with bias and one output neuron. These form a single-layer network. The inputs are given as $[x_1, x_2, x_3] = [0.8, 0.6, 0.4]$ and the weights are $[w_1, w_2, w_3] = [0.1, 0.3, -0.2]$ with bias $b = 0.35$ (its input is always 1).

The net input to the output neuron is

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

$[n = 3$, because only 3 input neurons are given$]$

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3$$
$$= 0.35 + 0.8 \times 0.1 + 0.6 \times 0.3 + 0.4 \times (-0.2)$$
$$= 0.35 + 0.08 + 0.18 - 0.08 = 0.53$$

(i) For binary sigmoidal activation function,

$$y = f(y_{in}) = \frac{1}{1 + e^{-y_{in}}} = \frac{1}{1 + e^{-0.53}} = 0.625$$

(ii) For bipolar sigmoidal activation function,

$$y = f(y_{in}) = \frac{2}{1 + e^{-y_{in}}} - 1 = \frac{2}{1 + e^{-0.53}} - 1$$
$$= 0.259$$

4. Implement AND function using McCulloch–Pitts neuron (take binary data).

Solution: Consider the truth table for AND function (Table 1).

Table 1

x_1	x_2	y
1	1	1
1	0	0
0	1	0
0	0	0

In McCulloch–Pitts neuron, only analysis is being performed. Hence, assume the weights be $w_1 = 1$ and $w_2 = 1$. The network architecture is shown in Figure 4. With these assumed weights, the net input is calculated for four inputs: For inputs

$(1, 1)$, $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 1 \times 1 = 2$
$(1, 0)$, $y_{in} = x_1 w_1 + x_2 w_2 = 1 \times 1 + 0 \times 1 = 1$
$(0, 1)$, $y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 1 \times 1 = 1$
$(0, 0)$, $y_{in} = x_1 w_1 + x_2 w_2 = 0 \times 1 + 0 \times 1 = 0$

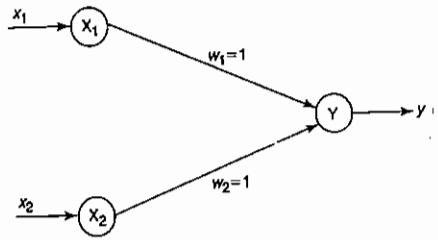


Figure 4 Neural net.

For an AND function, the output is high if both the inputs are high. For this condition, the net input is calculated as 2. Hence, based on this net input, the threshold is set, i.e. if the threshold value is greater than or equal to 2 then the neuron fires, else it does not fire. So the threshold value is set equal to $2(\theta = 2)$. This can also be obtained by

$$\theta \geq nw - p$$

Here, $n = 2, w = 1$ (excitatory weights) and $p = 0$ (no inhibitory weights). Substituting these values in the above-mentioned equation we get

$$\theta \geq 2 \times 1 - 0 \Rightarrow \theta \geq 2$$

Thus, the output of neuron Y can be written as

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 2 \\ 0 & \text{if } y_{in} < 2 \end{cases}$$

where "2" represents the threshold value.

5. Implement ANDNOT function using McCulloch–Pitts neuron (use binary data representation).

Solution: In the case of ANDNOT function, the response is true if the first input is true and the second input is false. For all other input variations, the response is false. The truth table for ANDNOT function is given in Table 2.

Table 2

x_1	x_2	y
0	0	0
0	1	0
1	0	1
1	1	0

The given function gives an output only when $x_1 = 1$ and $x_2 = 0$. The weights have to be decided only after the analysis. The net can be represented as shown in Figure 5.

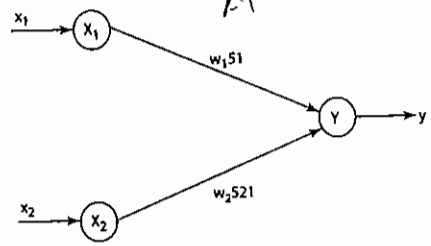


Figure 5 Neural net (weights fixed after analysis).

Case 1: Assume that both weights w_1 and w_2 are excitatory, i.e.,

$$w_1 = w_2 = 1$$

Then for the four inputs calculate the net input using

$$y_{in} = x_1 w_1 + x_2 w_2$$

For inputs

$(1, 1)$, $y_{in} = 1 \times 1 + 1 \times 1 = 2$
$(1, 0)$, $y_{in} = 1 \times 1 + 0 \times 1 = 1$
$(0, 1)$, $y_{in} = 0 \times 1 + 1 \times 1 = 1$
$(0, 0)$, $y_{in} = 0 \times 1 + 0 \times 1 = 0$

From the calculated net inputs, it is not possible to fire the neuron for input (1, 0) only. Hence, these weights are not suitable.

Assume one weight as excitatory and the other as inhibitory, i.e.,

$$w_1 = 1, \; w_2 = -1$$

Now calculate the net input. For the inputs

$(1,1)$, $y_{in} = 1 \times 1 + 1 \times -1 = 0$
$(1,0)$, $y_{in} = 1 \times 1 + 0 \times -1 = 1$
$(0,1)$, $y_{in} = 0 \times 1 + 1 \times -1 = -1$
$(0,0)$, $y_{in} = 0 \times 1 + 0 \times -1 = 0$

From the calculated net inputs, now it is possible to fire the neuron for input (1, 0) only by fixing a threshold of 1, i.e., $\theta \geq 1$ for Y unit. Thus,

$$w_1 = 1; \; w_2 = -1; \; \theta \geq 1$$

Note: The value of θ is calculated using the following:

$\theta \geq nw - p$
$\theta \geq 2 \times 1 - 1$ [for "p" inhibitory only magnitude considered]
$\theta \geq 1$

Thus, the output of neuron Y can be written as

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 1 \\ 0 & \text{if } y_{in} < 1 \end{cases}$$

6. Implement XOR function using McCulloch–Pitts neuron (consider binary data).

Solution: The truth table for XOR function is given in Table 3.

Table 3

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

In this case, the output is "ON" for only odd number of 1's. For the rest it is "OFF." XOR function cannot be represented by simple and single logic function; it is represented as

$$y = x_1\overline{x_2} + \overline{x_1}\,x_2$$
$$y = z_1 + z_2$$

where

$z_1 = x_1\overline{x_2}$ (function 1)
$z_2 = \overline{x_1}x_2$ (function 2)
$y = z_1(\text{OR})z_2$ (function 3)

A single-layer net is not sufficient to represent the function. An intermediate layer is necessary.

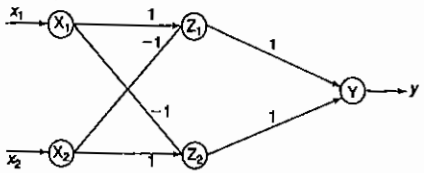


Figure 6 Neural net for XOR function (the weights shown are obtained after analysis).

- **First function** ($z_1 = x_1\overline{x_2}$): The truth table for function z_1 is shown in Table 4.

Table 4

x_1	x_2	z_1
0	0	0
0	1	0
1	0	1
1	1	0

The net representation is given as
Case 1: Assume both weights as excitatory, i.e.,

$$w_{11} = w_{21} = 1$$

Calculate the net inputs. For inputs,

$(0,0)$, $z_{1in} = 0 \times 1 + 0 \times 1 = 0$
$(0,1)$, $z_{1in} = 0 \times 1 + 1 \times 1 = 1$
$(1,0)$, $z_{1in} = 1 \times 1 + 0 \times 1 = 1$
$(1,1)$, $z_{1in} = 1 \times 1 + 1 \times 1 = 2$

Hence, it is not possible to obtain function z_1 using these weights.
Case 2: Assume one weight as excitatory and the other as inhibitory, i.e.,

$$w_{11} = 1; \quad w_{21} = -1$$

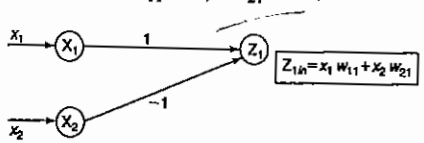


Figure 7 Neural net for Z_1.

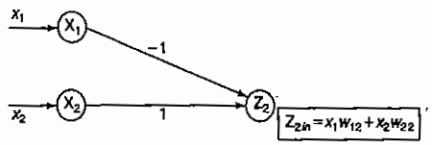


Figure 8 Neural net for Z_2.

Calculate the net inputs. For inputs

$(0,0)$, $z_{1in} = 0 \times 1 + 0 \times -1 = 0$
$(0,1)$, $z_{1in} = 0 \times 1 + 1 \times -1 = -1$
$(1,0)$, $z_{1in} = 1 \times 1 + 0 \times -1 = 1$
$(1,1)$, $z_{1in} = 1 \times 1 + 1 \times -1 = 0$

On the basis of this calculated net input, it is possible to get the required output. Hence,

$w_{11} = 1$
$w_{21} = -1$
$\theta \geq 1$ for the Z_1 neuron

- **Second function** ($z_2 = \overline{x_1}x_2$): The truth table for function z_2 is shown in Table 5.

Table 5

x_1	x_2	z_2
0	0	0
0	1	1
1	0	0
1	1	0

The net representation is given as follows:
Case 1: Assume both weights as excitatory, i.e.,

$$w_{12} = w_{22} = 1$$

Now calculate the net inputs. For the inputs

$(0,0)$, $z_{2in} = 0 \times 1 + 0 \times 1 = 0$
$(0,1)$, $z_{2in} = 0 \times 1 + 1 \times 1 = 1$
$(1,0)$, $z_{2in} = 1 \times 1 + 0 \times 1 = 1$
$(1,1)$, $z_{2in} = 1 \times 1 + 1 \times 1 = 2$

Hence, it is not possible to obtain function z_2 using these weights.

Case 2: Assume one weight as excitatory and the other as inhibitory, i.e.,

$$w_{12} = -1; \quad w_{22} = 1$$

Now calculate the net inputs. For the inputs

$(0,0)$, $z_{2in} = 0 \times -1 + 0 \times 1 = 0$
$(0,1)$, $z_{2in} = 0 \times -1 + 1 \times 1 = 1$
$(1,0)$, $z_{2in} = 1 \times -1 + 0 \times 1 = -1$
$(1,1)$, $z_{2in} = 1 \times -1 + 1 \times 1 = 0$

Thus, based on this calculated net input, it is possible to get the required output. Hence,

$w_{12} = -1$
$w_{22} = 1$
$\theta \geq 1$ for the Z_2 neuron

- **Third function** ($y = z_1$ OR z_2): The truth table for this function is shown in Table 6.

Table 6

x_1	x_2	y	z_1	z_2
0	0	0	0	0
0	1	1	0	1
1	0	1	1	0
1	1	0	0	0

Here the net input is calculated using

$$y_{in} = z_1v_1 + z_2v_2$$

Case 1: Assume both weights as excitatory, i.e.,

$$v_1 = v_2 = 1$$

Now calculate the net input. For inputs

$(0,0)$, $y_{in} = 0 \times 1 + 0 \times 1 = 0$
$(0,1)$, $y_{in} = 0 \times 1 + 1 \times 1 = 1$
$(1,0)$, $y_{in} = 1 \times 1 + 0 \times 1 = 1$
$(1,1)$, $y_{in} = 0 \times 1 + 0 \times 1 = 0$

(because for $x_1 = 1$ and $x_2 = 1, z_1 = 0$ and $z_2 = 0$)

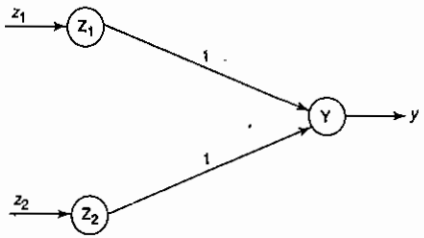


Figure 9 Neural net for $Y(Z_1$ OR $Z_2)$.

Setting a threshold of $\theta \geq 1$, $v_1 = v_2 = 1$, which implies that the net is recognized. Therefore, the analysis is made for XOR function using M–P neurons. Thus for XOR function, the weights are obtained as

$$w_{11} = w_{22} = 1 \quad \text{(excitatory)}$$
$$w_{12} = w_{21} = -1 \quad \text{(inhibitory)}$$
$$v_1 = v_2 = 1 \quad \text{(excitatory)}$$

7. Using the linear separability concept, obtain the response for OR function (take bipolar inputs and bipolar targets).

Solution: Table 7 is the truth table for OR function with bipolar inputs and targets.

Table 7

x_1	x_2	y
1	1	1
1	−1	1
−1	1	1
−1	−1	−1

The truth table inputs and corresponding outputs have been plotted in Figure 10. If output is 1, it is denoted as "+" else "−." Assuming the coordinates as (−1, 0) and (0, −1); (x_1, y_1) and (x_2, y_2), the slope "m" of the straight line can be obtained as

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{-1 - 0}{0 + 1} = \frac{-1}{1} = -1$$

We now calculate c:

$$c = y_1 - mx_1 = 0 - (-1)(-1) = -1$$

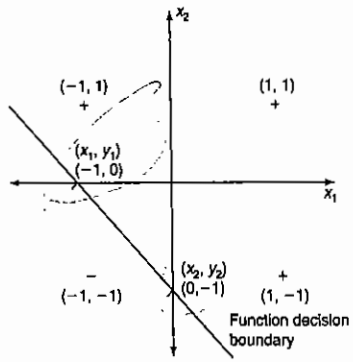


Figure 10 Graph for 'OR' function.

Using this value the equation for the line is given as

$$y = mx + c = (-1)x - 1 = -x - 1$$

Here the quadrants are not x and y but x_1 and x_2, so the above equation becomes

$$x_2 = -x_1 - 1 \tag{2.1}$$

This can be written as

$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{b}{w_2} \tag{2.2}$$

Comparing Eqs. (2.1) and (2.2), we get

$$\frac{w_1}{w_2} = \frac{1}{1}; \quad \frac{b}{w_2} = \frac{1}{1}$$

Therefore, $w_1 = 1$, $w_2 = 1$ and $b = 1$. Calculating the net input and output of OR function on the basis of these weights and bias, we get entries in Table 8.

Table 8

x_1	x_2	b	$y_{in} = b + x_1 w_1 + x_2 w_2$	y
1	1	1	3	1
1	−1	1	1	1
−1	1	1	1	1
−1	−1	1	−1	−1

Thus, the output of neuron Y can be written as

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} \geq 1 \\ 0 & \text{if } y_{in} < 1 \end{cases}$$

where the threshold is taken as "1" ($\theta = 1$) based on the calculated net input. Hence, using the linear separability concept, the response is obtained for "OR" function.

8. Design a Hebb net to implement logical AND function (use bipolar inputs and targets).

Solution: The training data for the AND function is given in Table 9.

Table 9

Inputs			Target
x_1	x_2	b	y
1	1	1	1
1	−1	1	−1
−1	1	1	−1
−1	−1	1	−1

The network is trained using the Hebb network training algorithm discussed in Section 2.7.3. Initially the weights and bias are set to zero, i.e.,

$$w_1 = w_2 = b = 0$$

- First input $[x_1\ x_2\ b] = [1\ 1\ 1]$ and target = 1 [i.e., $y = 1$]: Setting the initial weights as old weights and applying the Hebb rule, we get

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$
$$w_1(\text{new}) = w_1(\text{old}) + x_1 y = 0 + 1 \times 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + x_2 y = 0 + 1 \times 1 = 1$$
$$b(\text{new}) = b(\text{old}) + y = 0 + 1 = 1$$

The weights calculated above are the final weights that are obtained after presenting the first input. These weights are used as the initial weights when the second input pattern is presented. The weight change here is $\Delta w_i = x_i y$. Hence weight changes relating to the first input are

$$\Delta w_1 = x_1 y = 1 \times 1 = 1$$
$$\Delta w_2 = x_2 y = 1 \times 1 = 1$$
$$\Delta b = y = 1$$

- Second input $[x_1\ x_2\ b] = [1\ -1\ 1]$ and $y = -1$: The initial or old weights here are the final (new) weights obtained by presenting the first input pattern, i.e.,

$$[w_1\ w_2\ b] = [1\ 1\ 1]$$

The weight change here is

$$\Delta w_1 = x_1 y = 1 \times -1 = -1$$
$$\Delta w_2 = x_2 y = -1 \times -1 = 1$$
$$\Delta b = y = -1$$

The new weights here are

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 1 - 1 = 0$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 1 + 1 = 2$$
$$b(\text{new}) = b(\text{old}) + \Delta b = 1 - 1 = 0$$

Similarly, by presenting the third and fourth input patterns, the new weights can be calculated. Table 10 shows the values of weights for all inputs.

Table 10

Inputs				Weight changes			Weights		
x_1	x_2	b	y	Δw_1	Δw_2	Δb	w_1	w_2	b
							(0	0	0)
1	1	1	1	1	1	1	1	1	1
1	−1	1	−1	−1	1	−1	0	2	0
−1	1	1	−1	1	−1	−1	1	1	−1
−1	−1	1	−1	1	1	−1	2	2	−2

The separating line equation is given by

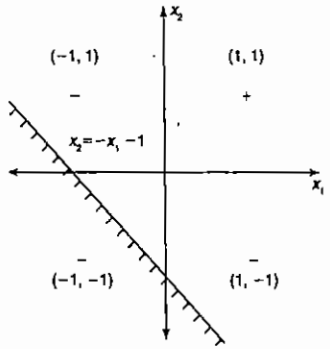
$$x_2 = \frac{-w_1}{w_2} x_1 - \frac{b}{w_2}$$

For all inputs, use the final weights obtained for each input to obtain the separating line. For the first input [1 1 1], the separating line is given by

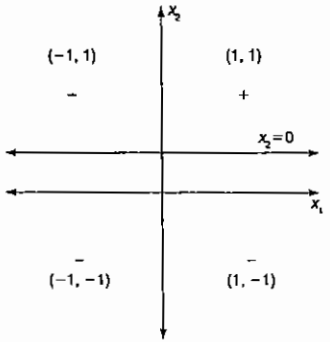
$$x_2 = \frac{-1}{1} x_1 - \frac{1}{1} \Rightarrow x_2 = -x_1 - 1$$

Similarly, for the second input [1 −1 1], the separating line is

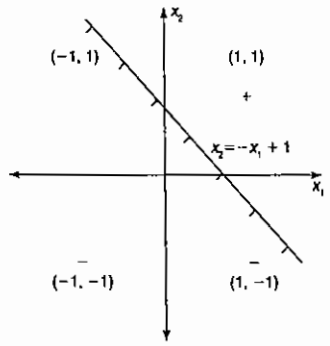
$$x_2 = \frac{-0}{2} x_1 - \frac{0}{2} \Rightarrow x_2 = 0$$



(A) First Input



(B) Second input



(C) Third and fourth inputs

Figure 11 Decision boundary for AND function using Hebb rule for each training pair.

For the third input [−1 1 1], it is

$$x_2 = \frac{-1}{1}x_1 + \frac{1}{1} \Rightarrow x_2 = -x_1 + 1$$

Finally, for the fourth input [−1 −1 1], the separating line is

$$x_2 = \frac{-2}{2}x_1 + \frac{2}{2} \Rightarrow x_2 = -x_1 + 1$$

The graphs for each of these separating lines obtained are shown in Figure 11. In this figure "+" mark is used for output "1" and "−" mark is used for output "−1." From Figure 11, it can be noticed that for the first input, the decision boundary differentiates only the first and fourth inputs, and not all negative responses are separated from positive responses. When the second input pattern is presented, the decision boundary separates (1, 1) from (1, −1) and (−1, −1) and not (−1, 1). But the boundary line is same for the both third and fourth training pairs. And, the decision boundary line obtained from these input training pairs separates the positive response region from the negative response region. Hence, the weights obtained from this are the final weights and are given as

$$w_1 = 2; \quad w_2 = 2; \quad b = -2$$

The network can be represented as shown in Figure 12.

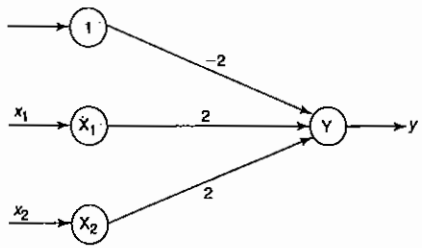


Figure 12 Hebb net for AND function.

9. Design a Hebb net to implement OR function (consider bipolar inputs and targets).

Solution: The training pair for the OR function is given in Table 11.

Table 11

Inputs			Target
x_1	x_2	b	y
1	1	1	1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

Initially the weights and bias are set to zero, i.e.,

$$w_1 = w_2 = b = 0$$

The network is trained and the final weights are outlined using the Hebb training algorithm discussed in Section 2.7.3. The weights are considered as final weights if the boundary line obtained from these weights separates the positive response region and negative response region.

By presenting all the input patterns, the weights are calculated. Table 12 shows the weights calculated for all the inputs.

Table 12

Inputs				Weight changes			Weights		
x_1	x_2	b	y	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
1	1	1	1	1	1	1	1	1	1
1	−1	1	1	1	−1	1	2	0	2
−1	1	1	1	−1	1	1	1	1	3
−1	−1	1	−1	1	1	−1	2	2	2

Using the final weights, the boundary line equation can be obtained. The separating line equation is

$$x_2 = \frac{-w_1}{w_2}x_1 - \frac{b}{w_2} = \frac{-2}{2}x_1 - \frac{2}{2} = -x_1 - 1$$

The decision region for this net is shown in Figure 13. It is observed in Figure 13 that straight line $x_2 = -x_1 - 1$ separates the pattern space into two regions. The input patterns [(1, 1), (1, −1), (−1, 1)] for which the output response is "1" lie on one side of the boundary, and the input pattern (−1, −1) for which the output response is "−1" lies on the other side of the boundary. Thus, the final weights are

$$w_1 = 2; \quad w_2 = 2; \quad b = 2$$

The network can be represented as shown in Figure 14.

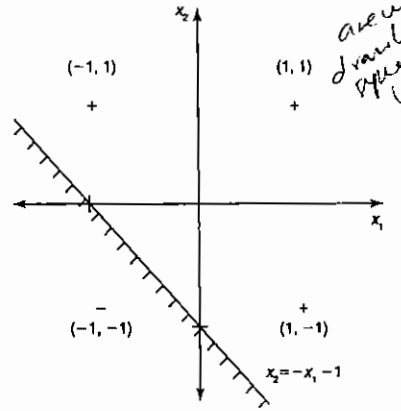


Figure 13 Decision boundary for OR function.

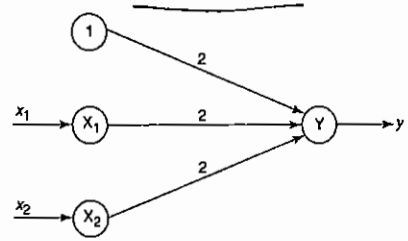


Figure 14 Hebb net for OR function.

10. Use the Hebb rule method to implement XOR function (take bipolar inputs and targets).

Solution: The training patterns for an XOR function are shown in Table 13.

Table 13

Inputs			Target
x_1	x_2	b	y
1	1	1	−1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

Here, a single-layer network with two input neurons, one bias and one output neuron is considered. In this case also, the initial weights are assumed to be zero:

$$w_1 = w_2 = b = 0$$

By using the Hebb training algorithm, the network is trained and the final weights are calculated as shown in the following Table 14.

Table 14

Inputs				Weight changes			Weights		
x_1	x_2	b	y	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
1	1	1	−1	−1	−1	−1	−1	−1	−1
1	−1	1	1	1	−1	1	0	−2	0
−1	1	1	1	−1	1	1	−1	−1	1
−1	−1	1	−1	1	1	−1	0	0	0

The final weights obtained after presenting all the input patterns do not give correct output for all patterns. Figure 15 shows that the input patterns are linearly non-separable. The graph shown in Figure 15 indicates that the four input pairs that are present cannot be divided by a single line to separate them into two regions. Thus XOR function is a case of a pattern classification problem, which is not linearly separable.

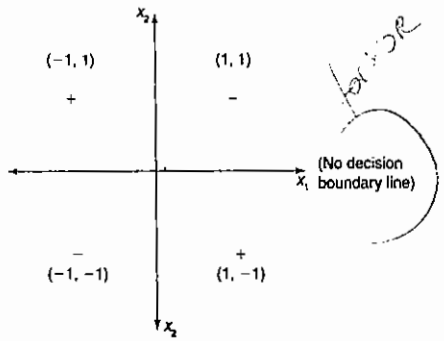


Figure 15 Decision boundary for XOR function.

The XOR function can be made linearly separable by solving it in a manner as discussed in Problem 6. This method of solving will result in two decision boundary lines for separating positive and negative regions of XOR function.

11. Using the Hebb rule, find the weights required to perform the following classifications of the given input patterns shown in Figure 16. The pattern is shown as 3 × 3 matrix form in the squares. The "+" symbols represent the value "1" and empty squares indicate "−1." Consider "I" belongs to the members of class (so has target value 1) and "O" does not belong to the members of class (so has target value −1).

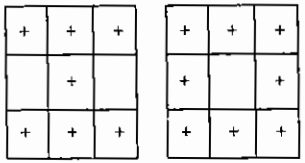


Figure 16 Data for input patterns.

Solution: The training input patterns for the given net (Figure 16) are indicated in Table 15.

Table 15

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
I	1	1	1	−1	1	−1	1	1	1	1	1
O	1	1	1	1	−1	1	1	1	1	1	−1

Here a single-layer network with nine input neurons, one bias and one output neuron is formed. Set the initial weights and bias to zero, i.e.,

$$w_1 = w_2 = w_3 = w_4 = w_5$$
$$= w_6 = w_7 = w_8 = w_9 = b = 0$$

Case 1: Presenting first input pattern (I), we calculate change in weights:

$$\Delta w_i = x_i y, \quad i = 1 \text{ to } 9$$
$$\Delta w_1 = x_1 y = 1 \times 1 = 1$$

$$\Delta w_2 = x_2 y = 1 \times 1 = 1$$
$$\Delta w_3 = x_3 y = 1 \times 1 = 1$$
$$\Delta w_4 = x_4 y = -1 \times 1 = -1$$
$$\Delta w_5 = x_5 y = 1 \times 1 = 1$$
$$\Delta w_6 = x_6 y = -1 \times 1 = -1$$
$$\Delta w_7 = x_7 y = 1 \times 1 = 1$$
$$\Delta w_8 = x_8 y = 1 \times 1 = 1$$
$$\Delta w_9 = x_9 y = 1 \times 1 = 1$$
$$\Delta b = y = 1$$

We now calculate the new weights using the formula

$$w_i(\text{new}) = w_i(\text{old}) + \Delta w_i$$

Setting the old weights as the initial weights here, we obtain

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0 + 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0 + 1 = 1$$
$$w_3(\text{new}) = w_3(\text{old}) + \Delta w_3 = 0 + 1 = 1$$

Similarly, calculating for other weights we get

$w_4(\text{new}) = -1$, $w_5(\text{new}) = 1$, $w_6(\text{new}) = -1$, $w_7(\text{new}) = 1$, $w_8(\text{new}) = 1$, $w_9(\text{new}) = 1$, $b(\text{new}) = 1$

The weights after presenting first input pattern are

$$W_{(\text{new})} = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1]$$

Case 2: Now we present the second input pattern (O). The initial weights used here are the final weights obtained after presenting the first input pattern. Here, the weights are calculated as shown below ($y = -1$ with the initial weights being [1 1 1 −1 1 −1 1 1 1 1]).

$$w_i(\text{new}) = w_i(\text{old}) + \Delta x_i \quad [\Delta w_i = x_i y]$$
$$w_1(\text{new}) = w_1(\text{old}) + x_1 y = 1 + 1 \times -1 = 0$$
$$w_2(\text{new}) = w_2(\text{old}) + x_2 y = 1 + 1 \times -1 = 0$$
$$w_3(\text{new}) = w_3(\text{old}) + x_3 y = 1 + 1 \times -1 = 0$$
$$w_4(\text{new}) = w_4(\text{old}) + x_4 y = -1 + 1 \times -1 = -2$$
$$w_5(\text{new}) = w_5(\text{old}) + x_5 y = 1 + -1 \times -1 = 2$$
$$w_6(\text{new}) = w_6(\text{old}) + x_6 y = -1 + 1 \times -1 = -2$$
$$w_7(\text{new}) = w_7(\text{old}) + x_7 y = 1 + 1 \times -1 = 0$$
$$w_8(\text{new}) = w_8(\text{old}) + x_8 y = 1 + 1 \times -1 = 0$$
$$w_9(\text{new}) = w_9(\text{old}) + x_9 y = 1 + 1 \times -1 = 0$$
$$b(\text{new}) = b(\text{old}) + y = 1 + 1 \times -1 = 0$$

The final weights after presenting the second input pattern are given as

$$W_{(\text{new})} = [0\ 0\ 0\ -2\ 2\ -2\ 0\ 0\ 0\ 0]$$

The weights obtained are indicated in the Hebb net shown in Figure 17.

12. Find the weights required to perform the following classifications of given input patterns using the Hebb rule. The inputs are "1" where "+" symbol is present and "−1" where "." is present. "L" pattern belongs to the class (target value +1) and "U" pattern does not belong to the class (target value −1).

Solution: The training input patterns for Figure 18 are given in Table 16.

Table 16

Pattern	Inputs										Target
	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y
L	1	−1	−1	1	−1	−1	1	1	1	1	1
U	1	−1	1	1	−1	1	1	1	1	1	−1

A single-layer network with nine input neurons, one bias and one output neuron is formed. Set the initial weights and bias to zero, i.e.,

$$w_1 = w_2 = w_3 = w_4 = w_5$$
$$= w_6 = w_7 = w_8 = w_9 = b = 0$$

The weights are calculated using

$$w_i(\text{new}) = w_i(\text{old}) + x_i y$$

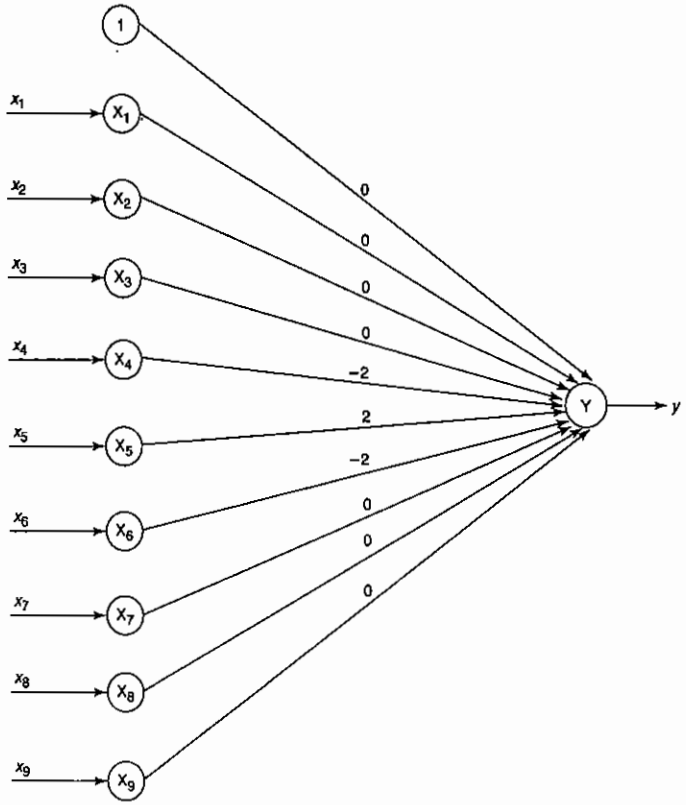


Figure 17 Hebb net for the data matrix shown in Figure 16.

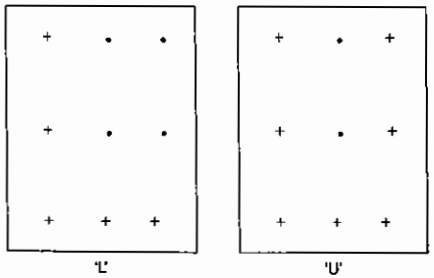


Figure 18 Input data for given patterns.

The calculated weights are given in Table 17.

Table 17

Inputs										Target	Weights									
x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	b	y	w_1	w_2	w_3	w_4	w_5	w_6	w_7	w_8	w_9	b
											(0	0	0	0	0	0	0	0	0	0)
1	−1	−1	1	−1	−1	1	1	1	1	1	1	−1	−1	1	−1	−1	1	1	1	1
1	−1	1	1	−1	1	1	1	1	1	−1	0	0	−2	0	0	−2	0	0	0	0

The final weights after presenting the two input patterns are

$$W_{(\text{new})} = [0\ 0\ -2\ 0\ 0\ -2\ 0\ 0\ 0\ 0]$$

The obtained weights are indicated in the Hebb net shown in Figure 19.

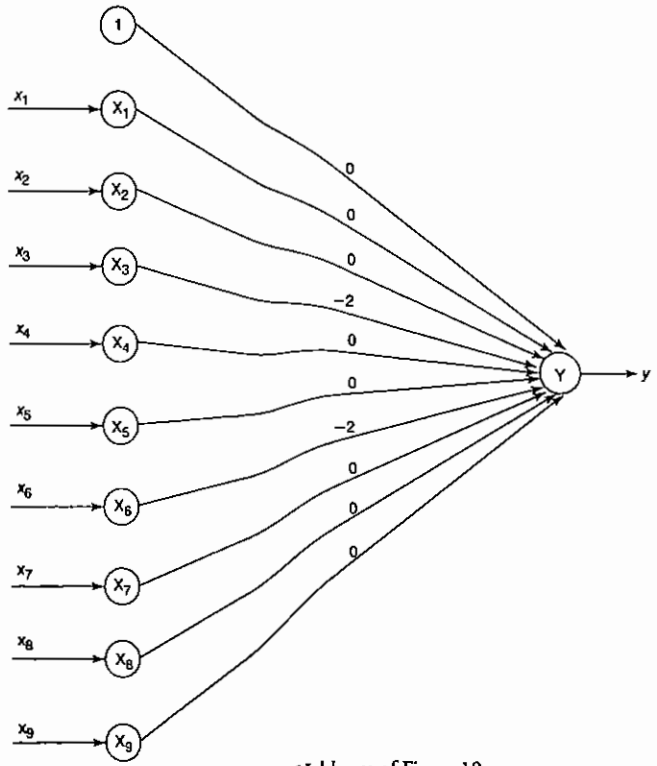


Figure 19 Hebb net of Figure 18.

2.10 Review Questions

1. Define an artificial neural network.
2. State the properties of the processing element of an artificial neural network.
3. How many signals can be sent by a neuron at a particular time instant?
4. Draw a simple artificial neuron and discuss the calculation of net input.
5. What is the influence of a linear equation over the net input calculation?
6. List the main components of the biological neuron.
7. Compare and contrast biological neuron and artificial neuron.
8. State the characteristics of an artificial neural network.
9. Discuss in detail the historical development of artificial neural networks.
10. What are the basic models of an artificial neural network?
11. Define net architecture and give its classifications.
12. Define learning.
13. Differentiate between supervised and unsupervised learning.
14. How is the critic information used in the learning process?
15. What is the necessity of activation function?
16. List the commonly used activation functions.
17. What is the impact of weight in an artificial neural network?
18. What is the other name for weight?
19. Define bias and threshold.
20. What is a learning rate parameter?
21. How does a momentum factor make faster convergence of a network?
22. State the role of vigilance parameter in ART network.
23. Why is the McCulloch–Pitts neuron widely used in logic functions?
24. Indicate the difference between excitatory and inhibitory weighted interconnections.
25. Define linear separability.
26. Justify – XOR function is non-linearly separable by a single decision boundary line.
27. How can the equation of a straight line be formed using linear separability?
28. In what ways is bipolar representation better than binary representation?
29. State the training algorithm used for the Hebb network.
30. Compare feed-forward and feedback network.

2.11 Exercise Problems

1. For the network shown in Figure 20, calculate the net input to the output neuron.

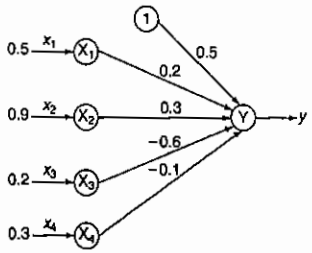


Figure 20 Neural net.

2. Calculate the output of neuron Y for the net shown in Figure 21. Use binary and bipolar sigmoidal activation functions.

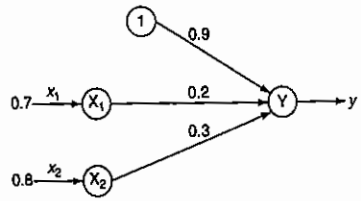


Figure 21 Neural net.

3. Design neural networks with only one M–P neuron that implements the three basic logic operations:
   (i) NOT (x_1);
   (ii) OR (x_1, x_2);
   (iii) NAND (x_1, x_2), where x_1 and $x_2 \in \{0, 1\}$.
4. (a) Show that the derivative of unipolar sigmoidal function is

$$f'(x) = \lambda f(x)[1 - f(x)]$$

   (b) Show that the derivative of bipolar sigmoidal function is

$$f'(x) = \frac{\lambda}{2}[1 + f(x)][1 - f(x)]$$

5. (a) Construct a feed-forward network with five input nodes, three hidden nodes and four output nodes that has lateral inhibition structure in the output layer.
   (b) Construct a recurrent network with four input nodes, three hidden nodes and two output nodes that has feedback links from the hidden layer to the input layer.
6. Using linear separability concept, obtain the response for NAND function.
7. Design a Hebb net to implement logical AND function with
   (a) binary inputs and targets and
   (b) binary inputs and bipolar targets.
8. Implement NOR function using Hebb net with
   (a) bipolar inputs and targets and
   (b) bipolar inputs and binary targets.
9. Classify the input patterns shown in Figure 22 using Hebb training algorithm.

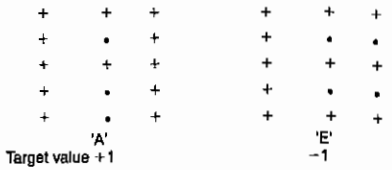


Figure 22 Input pattern.

10. Using Hebb rule, find the weights required to perform following classifications. The vectors (1 −1 1 −1) and (1 1 1 −1) belong to class (target value +1); vectors (−1 −1 1 1) and (1 1 −1 −1) do not belong to class (target value −1). Also using each of training x vectors as input, test the response of net.

2.12 Projects

1. Write a program to classify the letters and numerals using Hebb learning rule. Take a pair of letters or numerals of your own. Also, after training the network, test the response of the net using suitable activation function. Perform the classification using bipolar data as well as binary data.
2. Write suitable programs for implementing logic functions using McCulloch–Pitts neuron.
3. Write a computer program to train a Madaline to perform AND function, using MRI algorithm.
4. Write a program for implementing BPN for training a single-hidden-layer back-propagation

network with bipolar sigmoidal units ($\lambda = 1$) to achieve the following two-to-one mappings:

- $y = 6\sin(\pi x_1) + \cos(\pi x_2)$
- $y = \sin(\pi x_1)\cos(0.2\pi x_2)$

Set up two sets of data, each consisting of 10 input-output pairs, one for training and other for testing. The input–output data are obtained by varying input variables (x_1, x_2) within $[-1, +1]$ randomly. Also the output data are normalized within $[-1, 1]$. Apply training to find proper weights in the network.

Omsairam

3 Supervised Learning Network

Learning Objectives

- The basic networks in supervised learning.
- How the perceptron learning rule is better than the Hebb rule.
- Original perceptron layer description.
- Delta rule with single output unit.
- Architecture, flowchart, training algorithm and testing algorithm for perceptron, Adaline, Madaline, back-propagation and radial basis function network.
- The various learning factors used in BPN.
- An overview of Time Delay, Function Link, Wavelet and Tree Neural Networks.
- Difference between back-propagation and RBF networks.

3.1 Introduction

The chapter covers major topics involving supervised learning networks and their associated single-layer and multilayer feed-forward networks. The following topics have been discussed in detail – the perceptron learning rule for simple perceptrons, the delta rule (Widrow-Hoff rule) for Adaline and single-layer feed-forward networks with continuous activation functions, and the back-propagation algorithm for multilayer feed-forward networks with continuous activation functions. In short, all the feed-forward networks have been explored.

3.2 Perceptron Networks

3.2.1 Theory

Perceptron networks come under single-layer feed-forward networks and are also called *simple perceptrons.* As described in Table 2-2 (Evolution of Neural Networks) in Chapter 2, various types of perceptrons were designed by Rosenblatt (1962) and Minsky-Papert (1969, 1988). However, a simple perceptron network was discovered by Block in 1962.

The key points to be noted in a perceptron network are:

1. The perceptron network consists of three units, namely, sensory unit (input unit), associator unit (hidden unit), response unit (output unit).

2. The sensory units are connected to associator units with fixed weights having values 1, 0 or −1, which are assigned at random.
3. The binary activation function is used in sensory unit and associator unit.
4. The response unit has an activation of 1, 0 or −1. The binary step with fixed threshold θ is used as activation for associator. The output signals that are sent from the associator unit to the response unit are only binary.
5. The output of the perceptron network is given by

$$y = f(y_{in})$$

where $f(y_{in})$ is activation function and is defined as

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

6. The perceptron learning rule is used in the weight updation between the associator unit and the response unit. For each training input, the net will calculate the response and it will determine whether or not an error has occurred.
7. The error calculation is based on the comparison of the values of targets with those of the calculated outputs.
8. The weights on the connections from the units that send the nonzero signal will get adjusted suitably.
9. The weights will be adjusted on the basis of the learning rule if an error has occurred for a particular training pattern, i.e.,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

If no error occurs, there is no weight updation and hence the training process may be stopped. In the above equations, the target value "t" is +1 or −1 and α is the learning rate. In general, these learning rules begin with an initial guess at the weight values and then successive adjustments are made on the basis of the evaluation of an objective function. Eventually, the learning rules reach a near-optimal or optimal solution in a finite number of steps.

A perceptron network with its three units is shown in Figure 3-1. As shown in Figure 3-1, a sensory unit can be a two-dimensional matrix of 400 photodetectors upon which a lighted picture with geometric black and white pattern impinges. These detectors provide a binary (0) electrical signal if the input signal is found to exceed a certain value of threshold. Also, these detectors are connected randomly with the associator unit. The associator unit is found to consist of a set of subcircuits called *feature predicates*. The feature predicates are hard-wired to detect the specific feature of a pattern and are equivalent to the *feature detectors*. For a particular feature, each predicate is examined with a few or all of the responses of the sensory unit. It can be found that the results from the predicate units are also binary (0 or 1). The last unit, i.e. response unit, contains the pattern-recognizers or perceptrons. The weights present in the input layers are all fixed, while the weights on the response unit are trainable.

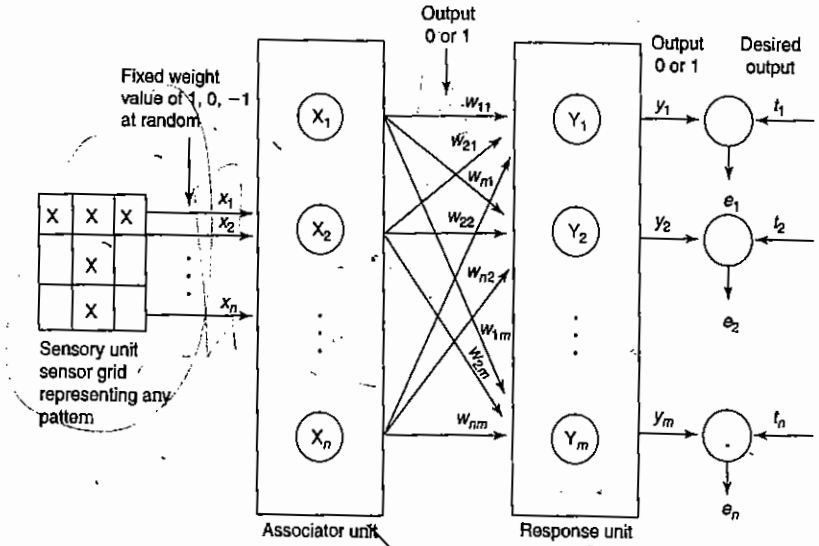


Figure 3-1 Original perceptron network.

3.2.2 Perceptron Learning Rule

In case of the perceptron learning rule, the learning signal is the difference between the desired and actual response of a neuron. The perceptron learning rule is explained as follows:

Consider a finite "n" number of input training vectors, with their associated target (desired) values $x(n)$ and $t(n)$, where "n" ranges from 1 to N. The target is either +1 or −1. The output "y" is obtained on the basis of the net input calculated and activation function being applied over the net input.

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

The weight updation in case of perceptron learning is as shown.

If $y \neq t$, then

$$w(\text{new}) = w(\text{old}) + \alpha t x \quad (\alpha - \text{learning rate})$$

else, we have

$$w(\text{new}) = w(\text{old})$$

The weights can be initialized at any values in this method. The perceptron rule convergence theorem states that "If there is a weight vector W such that $f(x(n)W) = t(n)$, for all n, then for any starting vector w_1, the perceptron learning rule will converge to a weight vector that gives the correct response for all

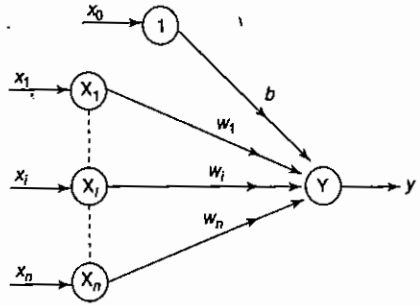


Figure 3-2 Single classification perceptron network.

training patterns, and this learning takes place within a finite number of steps provided that the solution exists."

3.2.3 Architecture

In the original perceptron network, the output obtained from the associator unit is a binary vector, and hence that output can be taken as input signal to the response unit, and classification can be performed. Here only the weights between the associator unit and the output unit can be adjusted, and the weights between the sensory and associator units are fixed. As a result, the discussion of the network is limited to a single portion. Thus, the associator unit behaves like the input unit. A simple perceptron network architecture is shown in Figure 3-2.

In Figure 3-2, there are n input neurons, 1 output neuron and a bias. The input-layer and output-layer neurons are connected through a directed communication link, which is associated with weights. The goal of the perceptron net is to classify the input pattern as a member or not a member to a particular class.

3.2.4 Flowchart for Training Process

The flowchart for the perceptron network training is shown in Figure 3-3. The network has to be suitably trained to obtain the response. The flowchart depicted here presents the flow of the training process.

As depicted in the flowchart, first the basic initialization required for the training process is performed. The entire loop of the training process continues until the training input pair is presented to the network. The training (weight updation) is done on the basis of the comparison between the calculated and desired output. The loop is terminated if there is no change in weight.

3.2.5 Perceptron Training Algorithm for Single Output Classes

The perceptron algorithm can be used for either binary or bipolar input vectors, having bipolar targets, threshold being fixed and variable bias. The algorithm discussed in this section is not particularly sensitive to the initial values of the weights or the value of the learning rate. In the algorithm discussed below, initially the inputs are assigned. Then the net input is calculated. The output of the network is obtained by applying the activation function over the calculated net input. On performing comparison over the calculated and

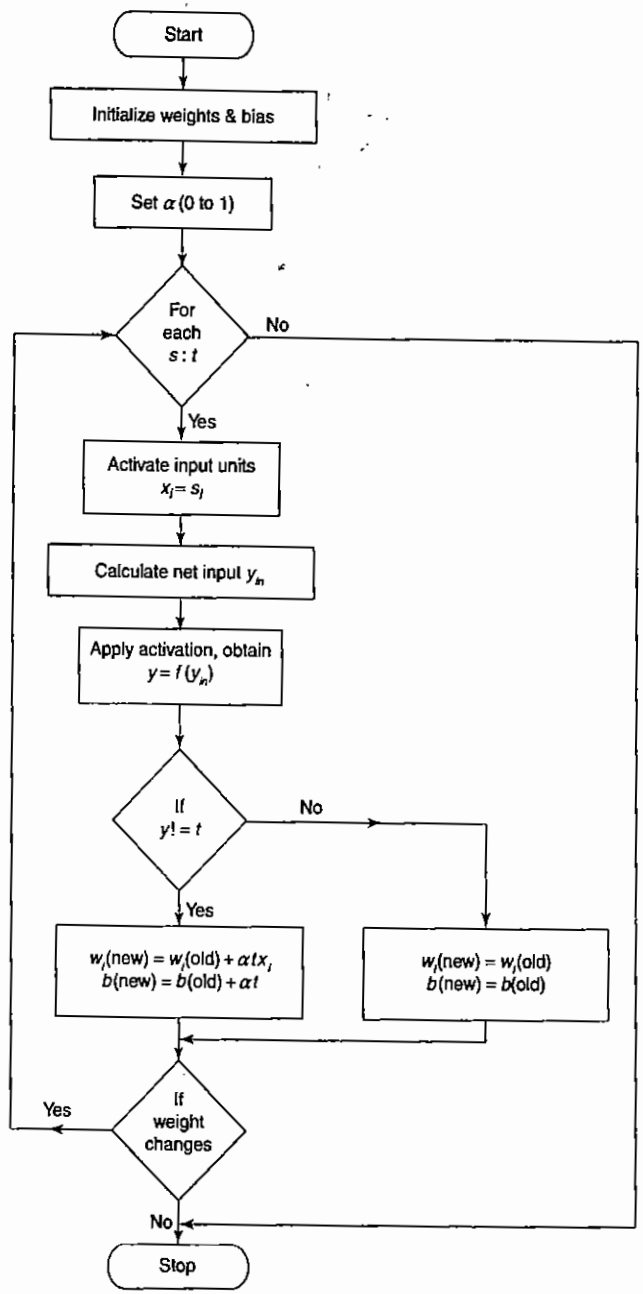


Figure 3-3 Flowchart for perceptron network with single output.

the desired output, the weight updation process is carried out. The entire network is trained based on the mentioned stopping criterion. The algorithm of a perceptron network is as follows:

Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha(0 < \alpha \leq 1)$. For simplicity α is set to 1.

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by *s:t*.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "n" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else, we have

$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

The algorithm discussed above is not sensitive to the initial values of the weights or the value of the learning rate.

3.2.6 Perceptron Training Algorithm for Multiple Output Classes

For multiple output classes, the perceptron training algorithm is as follows:

Step 0: Initialize the weights, biases and learning rate suitably.

Step 1: Check for stopping condition; if it is false, perform Steps 2–6.

Step 2: Perform Steps 3–5 for each bipolar or binary training vector pair *s:t*.

Step 3: Set activation (identity) of each input unit $i = 1$ to n:

$$x_i = s_i$$

Step 4: Calculate output response of each output unit $j = 1$ to m: First, the net input is calculated as

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}$$

Then activations are applied over the net input to calculate the output response:

$$y_j = f(y_{inj}) = \begin{cases} 1 & \text{if } y_{inj} > \theta \\ 0 & \text{if } -\theta \leq y_{inj} \leq \theta \\ -1 & \text{if } y_{inj} < -\theta \end{cases}$$

Step 5: Make adjustment in weights and bias for $j = 1$ to m and $i = 1$ to n.

If $t_j \neq y_j$, then

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha t_j x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha t_j$$

else, we have

$$w_{ij}(\text{new}) = w_{ij}(\text{old})$$
$$b_j(\text{new}) = b_j(\text{old})$$

Step 6: Test for the stopping condition, i.e., if there is no change in weights then stop the training process, else start again from Step 2.

It can be noticed that after training, the net classifies each of the training vectors. The above algorithm is suited for the architecture shown in Figure 3-4.

3.2.7 Perceptron Network Testing Algorithm

It is best to test the network performance once the training process is complete. For efficient performance of the network, it should be trained with more data. The testing algorithm (application procedure) is as follows:

Step 0: The initial weights to be used here are taken from the training algorithms (the final weights obtained during training).

Step 1: For each input vector X to be classified, perform Steps 2–3.

Step 2: Set activations of the input unit.

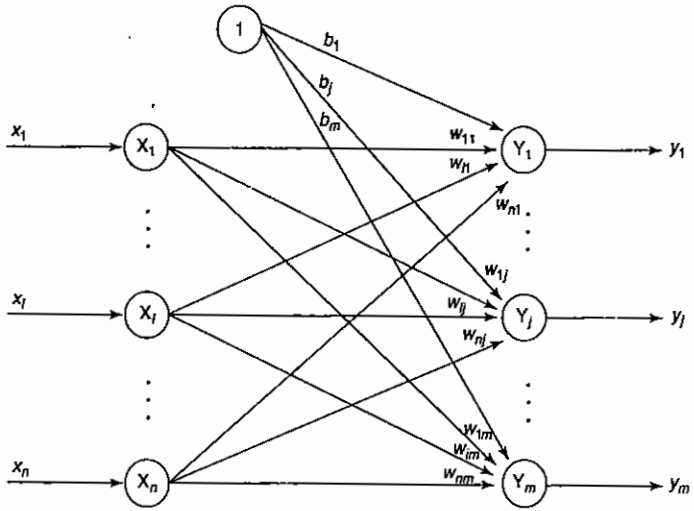


Figure 3-4 Network architecture for perceptron network for several output classes.

Step 3: Obtain the response of output unit.

$$y_{in} = \sum_{i=1}^{n} x_i w_i$$

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Thus, the testing algorithm tests the performance of network.

Note: In the case of perceptron network, it can be used for linear separability concept. Here the separating line may be based on the value of threshold, i.e., the threshold used in activation function must be a non-negative value.

The condition for separating the response from region of positive to region of zero is

$$w_1 x_1 + w_2 x_2 + b > \theta$$

The condition for separating the response from region of zero to region of negative is

$$w_1 x_1 + w_2 x_2 + b < -\theta$$

The conditions above are stated for a single-layer perceptron network with two input neurons and one output neuron and one bias.

3.3 Adaptive Linear Neuron (Adaline)

3.3.1 Theory

The units with linear activation function are called linear units. A network with a single linear unit is called an *Adaline* (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output. The weights between the input and the output are adjustable. The bias in Adaline acts like an adjustable weight, whose connection is from a unit with activations being always 1. Adaline is a net which has only one output unit. The Adaline network may be trained using delta rule. The delta rule may also be called as *least mean square* (LMS) rule or Widrow-Hoff rule. This learning rule is found to minimize the mean-squared error between the activation and the target value.

3.3.2 Delta Rule for Single Output Unit

The Widrow-Hoff rule is very similar to perceptron learning rule. However, their origins are different. The perceptron learning rule originates from the Hebbian assumption while the delta rule is derived from the gradient-descent method (it can be generalized to more than one layer). Also, the perceptron learning rule stops after a finite number of learning steps, but the gradient-descent approach continues forever, converging only asymptotically to the solution. The delta rule updates the weights between the connections so as to minimize the difference between the net input to the output unit and the target value. The major aim is to minimize the error over all training patterns. This is done by reducing the error for each pattern, one at a time.

The delta rule for adjusting the weight of ith pattern ($i = 1$ to n) is

$$\Delta w_i = \alpha(t - y_{in})x_i$$

where Δw_i is the weight change; α the learning rate; x the vector of activation of input unit; y_{in} the net input to output unit, i.e., $Y = \sum_{i=1}^{n} x_i w_i$; t the target output. The delta rule in case of several output units for adjusting the weight from ith input unit to the jth output unit (for each pattern) is

$$\Delta w_{ij} = \alpha(t_j - y_{inj})x_i$$

3.3.3 Architecture

As already stated, Adaline is a single-unit neuron, which receives input from several units and also from one unit called bias. An Adaline model is shown in Figure 3-5. The basic Adaline model consists of trainable weights. Inputs are either of the two values (+1 or −1) and the weights have signs (positive or negative). Initially, random weights are assigned. The net input calculated is applied to a quantizer transfer function (possibly activation function) that restores the output to +1 or −1. The Adaline model compares the actual output with the target output and on the basis of the training algorithm, the weights are adjusted.

3.3.4 Flowchart for Training Process

The flowchart for the training process is shown in Figure 3-6. This gives a pictorial representation of the network training. The conditions necessary for weight adjustments have to be checked carefully. The weights and other required parameters are initialized. Then the net input is calculated, output is obtained and compared with the desired output for calculation of error. On the basis of the error factor, weights are adjusted.

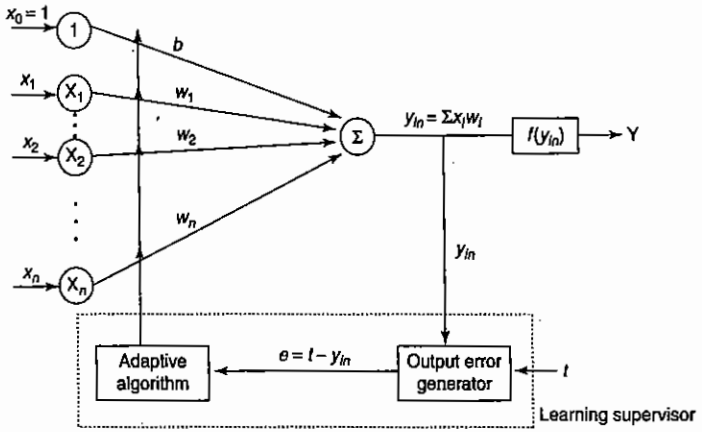


Figure 3-5 Adaline model.

3.3.5 Training Algorithm

The Adaline network training algorithm is as follows:

Step 0: Weights and bias are set to some random values but not zero. Set the learning rate parameter α.

Step 1: Perform Steps 2–6 when stopping condition is false.

Step 2: Perform Steps 3–5 for each bipolar training pair *s:t*.

Step 3: Set activations for input units $i = 1$ to n.

$$x_i = s_i$$

Step 4: Calculate the net input to the output unit.

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

Step 5: Update the weights and bias for $i = 1$ to n:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha (t - y_{in}) x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha (t - y_{in})$$

Step 6: If the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for stopping condition of a network.

The range of learning rate can be between 0.1 and 1.0.

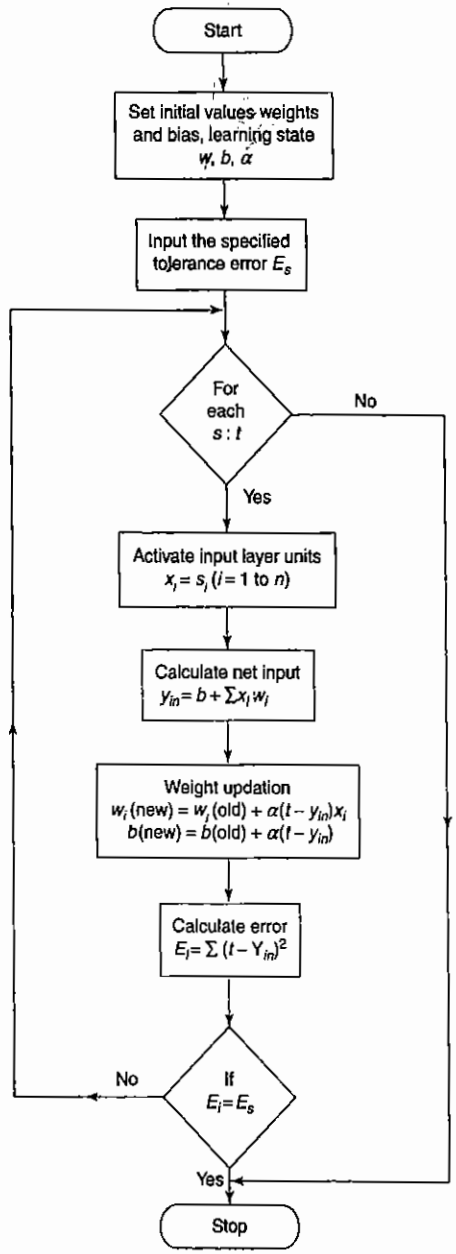


Figure 3-6 Flowchart for Adaline training process.

3.3.6 Testing Algorithm

It is essential to perform the testing of a network that has been trained. When training is completed, the Adaline can be used to classify input patterns. A step function is used to test the performance of the network. The testing procedure for the Adaline network is as follows:

Step 0: Initialize the weights. (The weights are obtained from the training algorithm.)

Step 1: Perform Steps 2–4 for each bipolar input vector x.

Step 2: Set the activations of the input units to x.

Step 3: Calculate the net input to the output unit:

$$y_{in} = b + \sum x_i w_i$$

Step 4: Apply the activation function over the net input calculated:

$$y = \begin{cases} 1 & \text{if } y_{in} \geq 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

3.4 Multiple Adaptive Linear Neurons

3.4.1 Theory

The multiple adaptive linear neurons (Madaline) model consists of many Adalines in parallel with a single output unit whose value is based on certain selection rules. It may use majority vote rule. On using this rule, the output would have as answer either true or false. On the other hand, if AND rule is used, the output is true if and only if both the inputs are true, and so on. The weights that are connected from the Adaline layer to the Madaline layer are fixed, positive and possess equal values. The weights between the input layer and the Adaline layer are adjusted during the training process. The Adaline and Madaline layer neurons have a bias of excitation "1" connected to them. The training process for a Madaline system is similar to that of an Adaline.

3.4.2 Architecture

A simple Madaline architecture is shown in Figure 3-7, which consists of "n" units of input layer, "m" units of Adaline layer and "1" unit of the Madaline layer. Each neuron in the Adaline and Madaline layers has a bias of excitation 1. The Adaline layer is present between the input layer and the Madaline (output) layer; hence, the Adaline layer can be considered a hidden layer. The use of the hidden layer gives the net computational capability which is not found in single-layer nets, but this complicates the training process to some extent.

The Adaline and Madaline models can be applied effectively in communication systems of adaptive equalizers and adaptive noise cancellation and other cancellation circuits.

3.4.3 Flowchart of Training Process

The flowchart of the training process of the Madaline network is shown in Figure 3-8. In case of training, the weights between the input layer and the hidden layer are adjusted, and the weights between the hidden layer

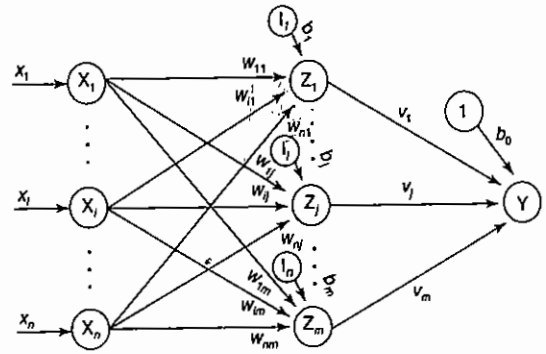


Figure 3-7 Architecture of Madaline layer.

and the output layer are fixed. The time taken for the training process in the Madaline network is very high compared to that of the Adaline network.

3.4.4 Training Algorithm

In this training algorithm, only the weights between the hidden layer and the input layer are adjusted, and the weights for the output units are fixed. The weights $v_1, v_2, \ldots, v_m$ and the bias b_0 that enter into output unit Y are determined so that the response of unit Y is 1. Thus, the weights entering Y unit may be taken as

$$v_1 = v_2 = \cdots = v_m = \tfrac{1}{2}$$

and the bias can be taken as

$$b_0 = \tfrac{1}{2}$$

The activation for the Adaline (hidden) and Madaline (output) units is given by

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{if } x < 0 \end{cases}$$

Step 0: Initialize the weights. The weights entering the output unit are set as above. Set initial small random values for Adaline weights. Also set initial learning rate α.

Step 1: When stopping condition is false, perform Steps 2–3.

Step 2: For each bipolar training pair $s:t$, perform Steps 3–7.

Step 3: Activate input layer units. For $i = 1$ to n,

$$x_i = s_i$$

Step 4: Calculate net input to each hidden Adaline unit:

$$z_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

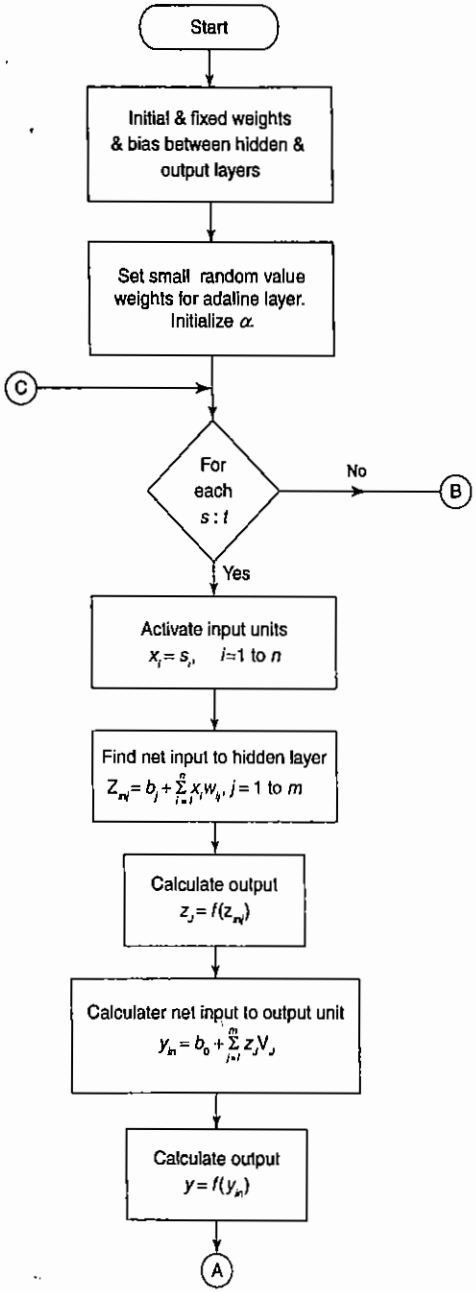


Figure 3-8 Flowchart for training of Madaline.

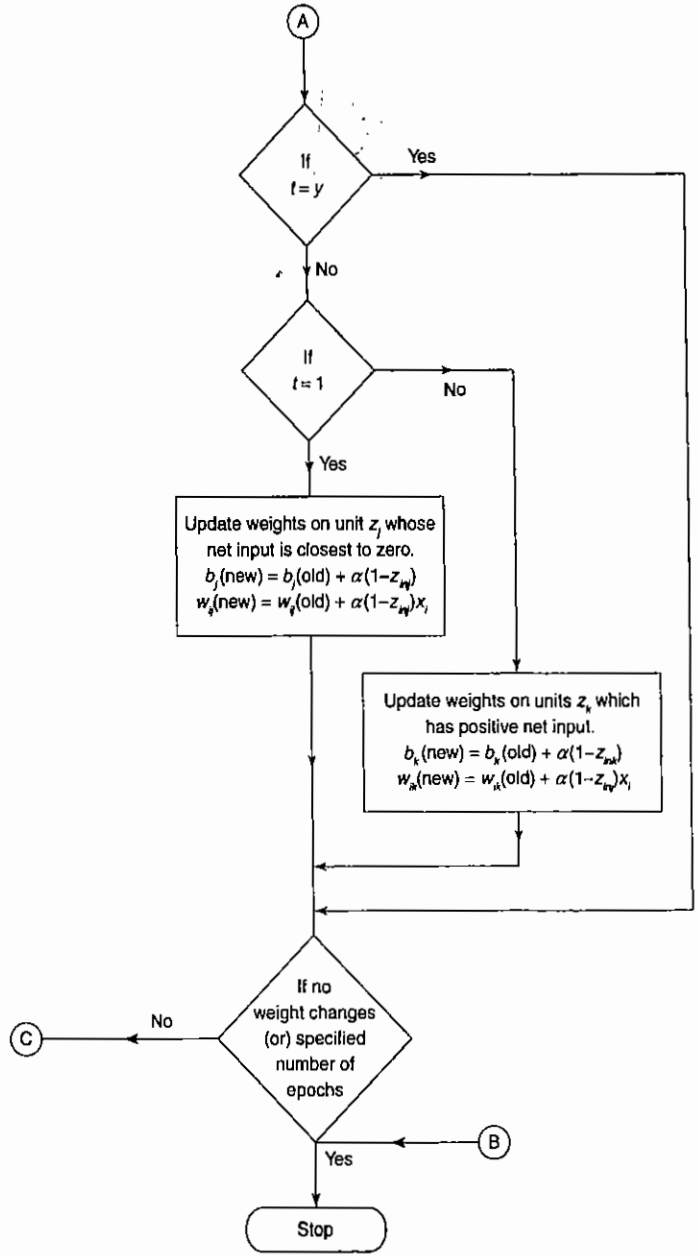


Figure 3-8 *(Continued).*

Step 5: Calculate output of each hidden unit:

$$z_j = f(z_{inj})$$

Step 6: Find the output of the net:

$$y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$$

$$y = f(y_{in})$$

Step 7: Calculate the error and update the weights.

1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha(1 - z_{inj})$$
$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(1 - z_{inj})x_i$$

3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z_{ink})x_i$$
$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z_{ink})$$

Step 8: Test for the stopping condition. (If there is no weight change or weight reaches a satisfactory level, or if a specified maximum number of iterations of weight updation have been performed then stop, or else continue).

Madalines can be formed with the weights on the output unit set to perform some logic functions. If there are only two hidden units present, or if there are more than two hidden units, then the "majority vote rule" function may be used.

3.5 Back-Propagation Network

3.5.1 Theory

The back-propagation learning algorithm is one of the most important developments in neural networks (Bryson and Ho, 1969; Werbos, 1974; Lecun, 1985; Parker, 1985; Rumelhart, 1986). This network has re-awakened the scientific and engineering community to the modeling and processing of numerous quantitative phenomena using neural networks. This learning algorithm is applied to multilayer feed-forward networks consisting of processing elements with continuous differentiable activation functions. The networks associated with back-propagation learning algorithm are also called *back-propagation networks* (BPNs). For a given set of training input-output pair, this algorithm provides a procedure for changing the weights in a BPN to classify the given input patterns correctly. The basic concept for this weight update algorithm is simply the gradient-descent method as used in the case of simple perceptron networks with differentiable units. This is a method where the error is propagated back to the hidden unit. The aim of the neural network is to train the net to achieve a balance between the net's ability to respond (memorization) and its ability to give reasonable responses to the input that is similar but not identical to the one that is used in training (generalization).

The back-propagation algorithm is different from other networks in respect to the process by which the weights are calculated during the learning period of the network. The general difficulty with the multilayer perceptrons is calculating the weights of the hidden layers in an efficient way that would result in a very small or zero output error. When the hidden layers are increased the network training becomes more complex. To update weights, the error must be calculated. The error, which is the difference between the actual (calculated) and the desired (target) output, is easily measured at the output layer. It should be noted that at the hidden layers, there is no direct information of the error. Therefore, other techniques should be used to calculate an error at the hidden layer, which will cause minimization of the output error, and this is the ultimate goal.

The training of the BPN is done in three stages – the feed-forward of the input training pattern, the calculation and back-propagation of the error, and updation of weights. The testing of the BPN involves the computation of feed-forward phase only. There can be more than one hidden layer (more beneficial) but one hidden layer is sufficient. Even though the training is very slow, once the network is trained it can produce its outputs very rapidly.

3.5.2 Architecture

A back-propagation neural network is a multilayer, feed-forward neural network consisting of an input layer, a hidden layer and an output layer. The neurons present in the hidden and output layers have biases, which are the connections from the units whose activation is always 1. The bias terms also acts as weights. Figure 3-9 shows the architecture of a BPN, depicting only the direction of information flow for the feed-forward phase. During the back-propagation phase of learning, signals are sent in the reverse direction.

The inputs are sent to the BPN and the output obtained from the net could be either binary (0, 1) or bipolar (−1, +1). The activation function could be any function which increases monotonically and is also differentiable.

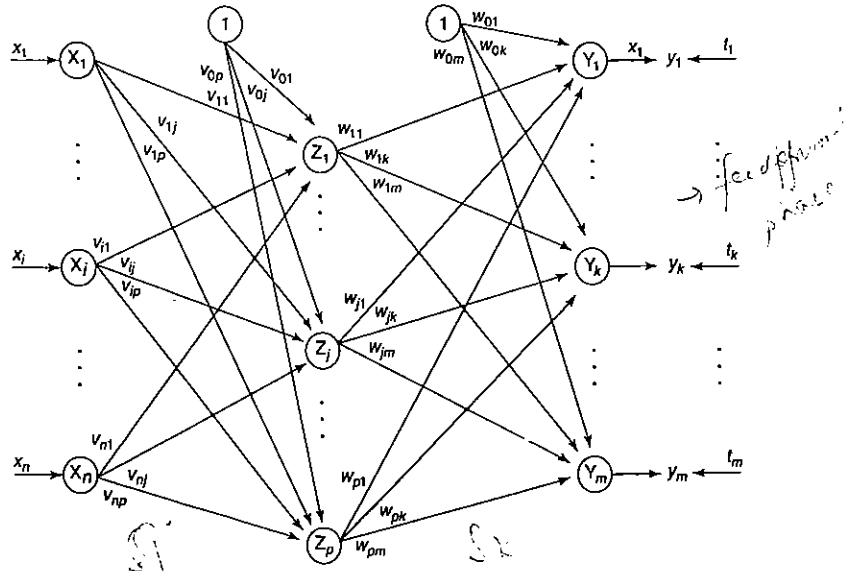


Figure 3-9 Architecture of a back-propagation network.

3.5.3 Flowchart for Training Process

The flowchart for the training process using a BPN is shown in Figure 3-10. The terminologies used in the flowchart and in the training algorithm are as follows:

x = input training vector $(x_1, \ldots, x_i, \ldots, x_n)$
t = target output vector $(t_1, \ldots, t_k, \ldots, t_m)$
α = learning rate parameter
x_i = input unit i. (Since the input layer uses identity activation function, the input and output signals here are same.)
v_{0j} = bias on jth hidden unit
w_{0k} = bias on kth output unit
z_j = hidden unit j. The net input to z_j is

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

and the output is

$$z_j = f(z_{inj})$$

y_k = output unit k. The net input to y_k is

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and the output is

$$y_k = f(y_{ink})$$

δ_k = error correction weight adjustment for w_{jk} that is due to an error at output unit y_k, which is back-propagated to the hidden units that feed into unit y_k
δ_j = error correction weight adjustment for v_{ij} that is due to the back-propagation of error to the hidden unit z_j.

Also, it should be noted that the commonly used activation functions are binary sigmoidal and bipolar sigmoidal activation functions (discussed in Section 2.3.3). These functions are used in the BPN because of the following characteristics: (i) continuity; (ii) differentiability; (iii) nondecreasing monotony.

The range of binary sigmoid is from 0 to 1, and for bipolar sigmoid it is from -1 to $+1$.

3.5.4 Training Algorithm

The error back-propagation learning algorithm can be outlined in the following algorithm:

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2–9 when stopping condition is false.

Step 2: Perform Steps 3–8 for each training pair.

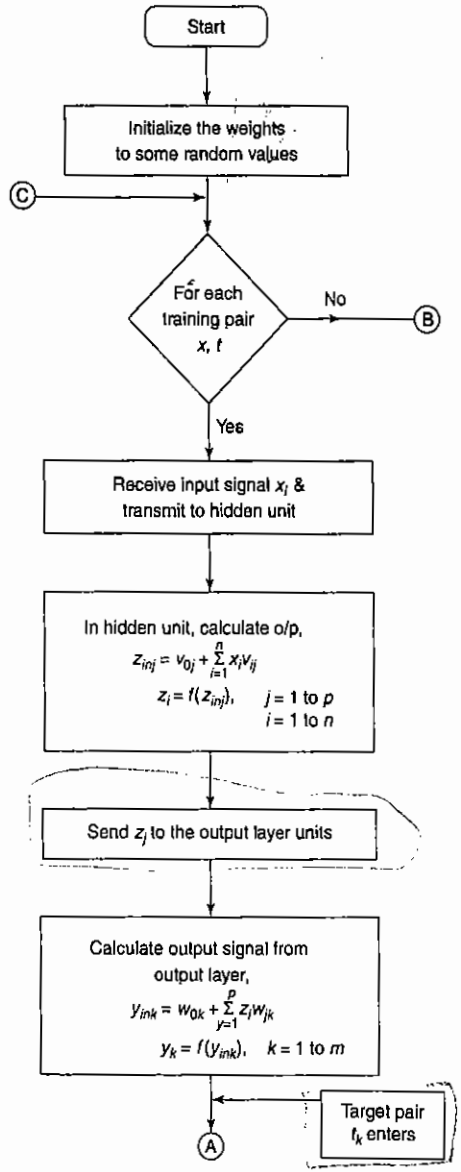


Figure 3-10 Flowchart for back-propagation network training.

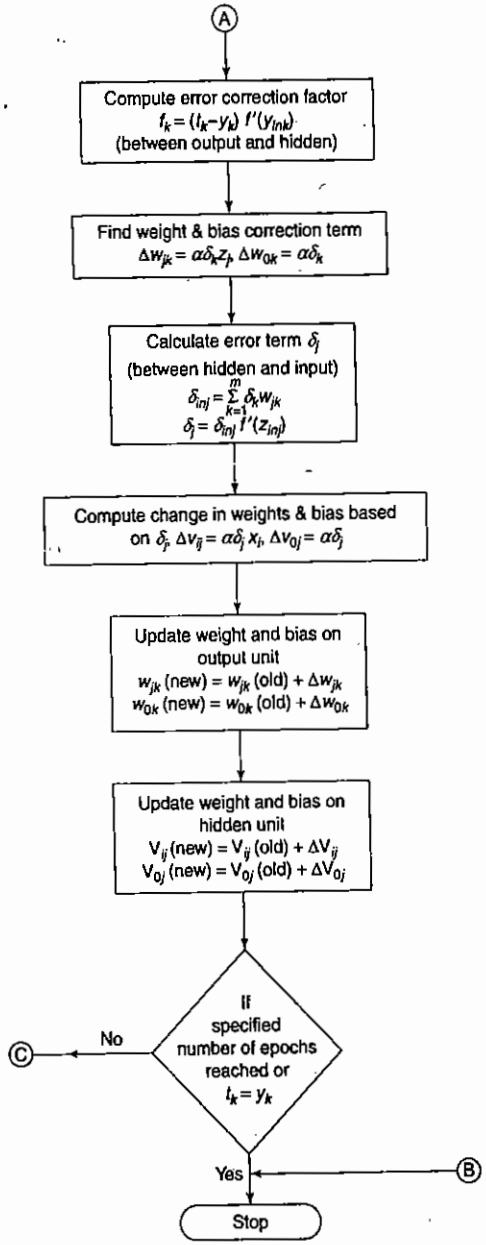


Figure 3-10 (*Continued*).

Feed-forward phase (Phase I)

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit $z_j (j = 1 \text{ to } p)$ sums its weighted input signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units.

Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

Back-propagation of error (Phase II):

Step 6: Each output unit $y_k (k = 1 \text{ to } m)$ receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit ($z_j, j = 1 \text{ to } p$) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$

Weight and bias updation (Phase III):

Step 8: Each output unit (y_k, $k = 1$ to m) updates the bias and weights:

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

Each hidden unit (z_j, $j = 1$ to p) updates its bias and weights:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$$
$$v_{0j}(\text{new}) = v_{0j}(\text{old}) + \Delta v_{0j}$$

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

The above algorithm uses the incremental approach for updation of weights, i.e., the weights are being changed immediately after a training pattern is presented. There is another way of training called *batch-mode training*, where the weights are changed only after all the training patterns are presented. The effectiveness of two approaches depends on the problem, but batch-mode training requires additional local storage for each connection to maintain the immediate weight changes. When a BPN is used as a classifier, it is equivalent to the optimal Bayesian discriminant function for asymptotically large sets of statistically independent training patterns.

The problem in this case is whether the back-propagation learning algorithm can always converge and find proper weights for network even after enough learning. It will converge since it implements a gradient-descent on the error surface in the weight space, and this will roll down the error surface to the nearest minimum error and will stop. This becomes true only when the relation existing between the input and the output training patterns is deterministic and the error surface is deterministic. This is not the case in real world because the produced square-error surfaces are always at random. This is the stochastic nature of the back-propagation algorithm, which is purely based on the stochastic gradient-descent method. The BPN is a special case of stochastic approximation.

If the BPN algorithm converges at all, then it may get stuck with local minima and may be unable to find satisfactory solutions. The randomness of the algorithm helps it to get out of local minima. The error functions may have large number of global minima because of permutations of weights that keep the network input–output function unchanged. This causes the error surfaces to have numerous troughs.

3.5.5 Learning Factors of Back-Propagation Network

The training of a BPN is based on the choice of various parameters. Also, the convergence of the BPN is based on some important learning factors such as the initial weights, the learning rate, the updation rule, the size and nature of the training set, and the architecture (number of layers and number of neurons per layer).

3.5.5.1 Initial Weights

The ultimate solution may be affected by the initial weights of a multilayer feed-forward network. They are initialized at small random values. The choice of the initial weight determines how fast the network converges. The initial weights cannot be very high because the sigmoidal activation functions used here may get saturated

from the beginning itself and the system may be stuck at a local minima or at a very flat plateau at the starting point itself. One method of choosing the weight w_{ij} is choosing it in the range

$$w_{ij} = \left[\frac{-3}{\sqrt{o_i}}, \frac{3}{\sqrt{o_i}}\right]$$

where o_i is the number of processing elements j that feed-forward to processing element i. The initialization can also be done by a method called Nyugen–Widrow initialization. This type of initialization leads to faster convergence of network. The concept here is based on the geometric analysis of the response of hidden neurons to a single input. The method is used for improving the learning ability of the hidden units. The random initialization of weights connecting input neurons to the hidden neurons is obtained by the equation

$$v_{ij}(\text{new}) = \gamma \frac{v_{ij}(\text{old})}{\left\| v_j(\text{old}) \right\|}$$

where $\overline{v_j}$ is the average weight calculated for all values of i, and the scale factor $\gamma = 0.7(P)^{1/n}$ ("n" is the number of input neurons and "P" is the number of hidden neurons).

3.5.5.2 Learning Rate α

The learning rate (α) affects the convergence of the BPN. A larger value of α may speed up the convergence but might result in overshooting, while a smaller value of α has vice-versa effect. The range of α from 10^{-3} to 10 has been used successfully for several back-propagation algorithmic experiments. Thus, a large learning rate leads to rapid learning but there is oscillation of weights, while the lower learning rate leads to slower learning.

3.5.5.3 Momentum Factor

The gradient descent is very slow if the learning rate α is small and oscillates widely if α is too large. One very efficient and commonly used method that allows a larger learning rate without oscillations is by adding a momentum factor to the normal gradient descent method.

The momentum factor is denoted by $\eta \in [0, 1]$ and the value of 0.9 is often used for the momentum factor. Also, this approach is more useful when some training data are very different from the majority of data. A momentum factor can be used with either pattern by pattern updating or batch-mode updating. In case of batch mode, it has the effect of complete averaging over the patterns. Even though the averaging is only partial in the pattern-by-pattern mode, it leaves some useful information for weight updation.

The weight updation formulas used here are

$$w_{jk}(t+1) = w_{jk}(t) + \underbrace{\alpha \delta_k z_j + \eta \left[w_{jk}(t) - w_{jk}(t-1) \right]}_{\Delta w_{jk}(t+1)}$$

and

$$v_{ij}(t+1) = v_{ij}(t) + \underbrace{\alpha \delta_j x_i + \eta \left[v_{ij}(t) - v_{ij}(t-1) \right]}_{\Delta v_{ij}(t+1)}$$

The momentum factor also helps in faster convergence.

3.5.5.4 Generalization

The best network for generalization is BPN. A network is said to be generalized when it sensibly interpolates with input networks that are new to the network. When there are many trainable parameters for the given amount of training data, the network learns well but does not generalize well. This is usually called *overfitting* or *overtraining*. One solution to this problem is to monitor the error on the test set and terminate the training when the error increases. With small number of trainable parameters, the network fails to learn the training data and performs very poorly on the test data. For improving the ability of the network to generalize from a training data set to a test data set, it is desirable to make small changes in the input space of a pattern, without changing the output components. This is achieved by introducing variations in the input space of training patterns as part of the training set. However, computationally, this method is very expensive. Also, a net with large number of nodes is capable of memorizing the training set at the cost of generalization. As a result, smaller nets are preferred than larger ones.

3.5.5.5 Number of Training Data

The training data should be sufficient and proper. There exists a rule of thumb, which states that the training data should cover the entire expected input space, and while training, training-vector pairs should be selected randomly from the set. Assume that the input space as being linearly separable into "L" disjoint regions with their boundaries being part of hyper planes. Let "T" be the lower bound on the number of training patterns. Then, choosing T such that $T/L \gg 1$ will allow the network to discriminate pattern classes using fine piecewise hyperplane partitioning. Also in some cases, scaling or normalization has to be done to help learning.

3.5.5.6 Number of Hidden Layer Nodes

If there exists more than one hidden layer in a BPN, then the calculations performed for a single layer are repeated for all the layers and are summed up at the end. In case of all multilayer feed-forward networks, the size of a hidden layer is very important. The number of hidden units required for an application needs to be determined separately. The size of a hidden layer is usually determined experimentally. For a network of a reasonable size, the size of hidden nodes has to be only a relatively small fraction of the input layer. For example, if the network does not converge to a solution, it may need more hidden nodes. On the other hand, if the network converges, the user may try a very few hidden nodes and then settle finally on a size based on overall system performance.

3.5.6 Testing Algorithm of Back-Propagation Network

The testing procedure of the BPN is as follows:

Step 0: Initialize the weights. The weights are taken from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector.

Step 2: Set the activation of input unit for x_i ($i = 1$ to n).

Step 3: Calculate the net input to hidden unit x and its output. For $j = 1$ to p,

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

$$z_i = f(z_{inj})$$

Step 4: Now compute the output of the output layer unit. For $k = 1$ to m,

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

$$y_k = f(y_{ink})$$

Use sigmoidal activation functions for calculating the output.

3.6 Radial Basis Function Network

3.6.1 Theory

The radial basis function (RBF) is a classification and functional approximation neural network developed by M.J.D. Powell. The network uses the most common nonlinearities such as sigmoidal and Gaussian kernel functions. The Gaussian functions are also used in regularization networks. The response of such a function is positive for all values of y; the response decreases to 0 as $|y| \to 0$. The Gaussian function is generally defined as

$$f(y) = e^{-y^2}$$

The derivative of this function is given by

$$f'(y) = -2ye^{-y^2} = -2yf(y)$$

The graphical representation of this Gaussian function is shown in Figure 3-11 below.

When the Gaussian potential functions are being used, each node is found to produce an identical output for inputs existing within the fixed radial distance from the center of the kernel, they are found to be radically symmetric, and hence the name radial basis function network. The entire network forms a linear combination of the nonlinear basis function.

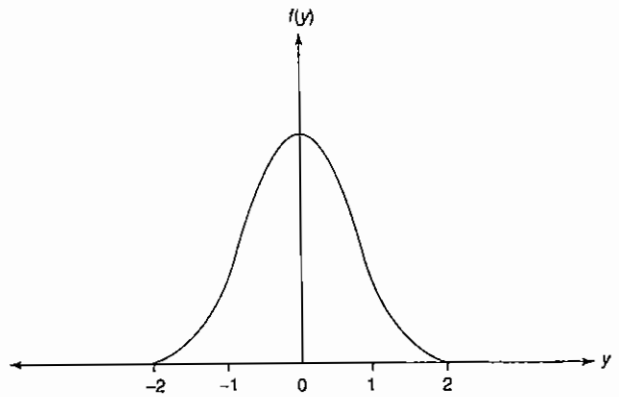


Figure 3-11 Gaussian kernel function.

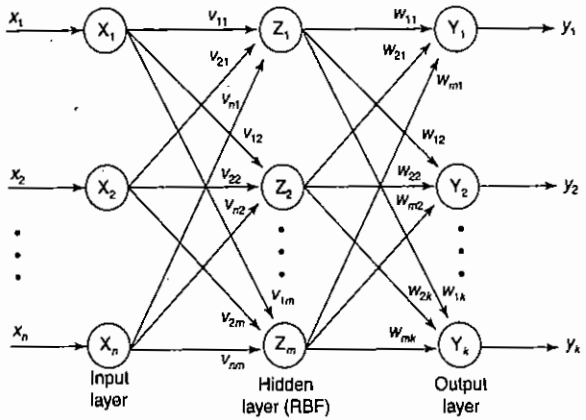


Figure 3-12 Architecture of RBF.

3.6.2 Architecture

The architecture for the radial basis function network (RBFN) is shown in Figure 3-12. The architecture consists of two layers whose output nodes form a linear combination of the kernel (or basis) functions computed by means of the RBF nodes or hidden layer nodes. The basis function (nonlinearity) in the hidden layer produces a significant nonzero response to the input stimulus it has received only when the input of it falls within a small localized region of the input space. This network can also be called as *localized receptive field network*.

3.6.3 Flowchart for Training Process

The flowchart for the training process of the RBF is shown in Figure 3-13 below. In this case, the center of the RBF functions has to be chosen and hence, based on all parameters, the output of network is calculated.

3.6.4 Training Algorithm

The training algorithm describes in detail all the calculations involved in the training process depicted in the flowchart. The training is started in the hidden layer with an unsupervised learning algorithm. The training is continued in the output layer with a supervised learning algorithm. Simultaneously, we can apply supervised learning algorithm to the hidden and output layers for fine-tuning of the network. The training algorithm is given as follows.

Step 0: Set the weights to small random values.

Step 1: Perform Steps 2–8 when the stopping condition is false.

Step 2: Perform Steps 3–7 for each input.

Step 3: Each input unit (x_i for all $i = 1$ to n) receives input signals and transmits to the next hidden layer unit.

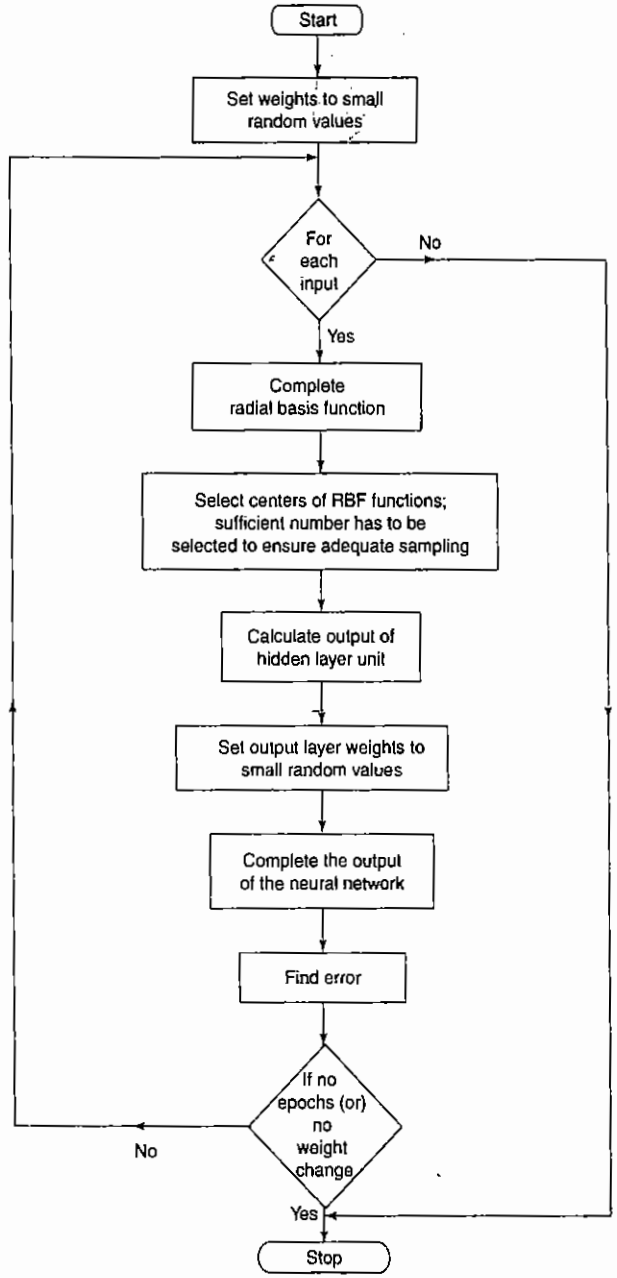


Figure 3-13 Flowchart for the training process of RBF.

Step 4: Calculate the radial basis function.

Step 5: Select the centers for the radial basis function. The centers are selected from the set of input vectors. It should be noted that a sufficient number of centers have to be selected to ensure adequate sampling of the input vector space.

Step 6: Calculate the output from the hidden layer unit:

$$v_i(x_i) = \frac{\exp\left[-\sum_{j=1}^{r}(x_{ji} - \hat{x}_{ji})^2\right]}{\sigma_i^2}$$

where $\hat{x}_{ji}$ is the center of the RBF unit for input variables; σ_i the width of ith RBF unit; x_{ji} the jth variable of input pattern.

Step 7: Calculate the output of the neural network:

$$y_{net} = \sum_{i=1}^{k} w_{im} v_i(x_i) + w_0$$

where k is the number of hidden layer nodes (RBF function); y_{net} the output value of mth node in output layer for the nth incoming pattern; w_{im} the weight between ith RBF unit and mth output node; w_0 the biasing term at nth output node.

Step 8: Calculate the error and test for the stopping condition. The stopping condition may be number of epochs or to a certain extent weight change.

Thus, a network can be trained using RBFN.

3.7 Time Delay Neural Network

The neural network has to respond to a sequence of patterns. Here the network is required to produce a particular output sequence in response to a particular sequence of inputs. A shift register can be considered as a tapped delay line. Consider a case of a multilayer perceptron where the tapped outputs of the delay line are applied to its inputs. This type of network constitutes a *time delay neural network* (TDNN). The output consists of a finite temporal dependence on its inputs, given as

$$U(t) = F[x(t), x(t-1), \ldots, x(t-n)]$$

where F is any nonlinearity function. The multilayer perceptron with delay line is shown in Figure 3-14.

When the function $U(t)$ is a weighted sum, then the TDNN is equivalent to a finite impulse response filter (FIR). In TDNN, when the output is being fed back through a unit delay into the input layer, then the net computed here is equivalent to an infinite impulse response (IIR) filter. Figure 3-15 shows TDNN with output feedback.

Thus, a neuron with a tapped delay line is called a TDNN unit, and a network which consists of TDNN units is called a TDNN. A specific application of TDNNs is speech recognition. The TDNN can be trained using the back-propagation-learning rule with a momentum factor.

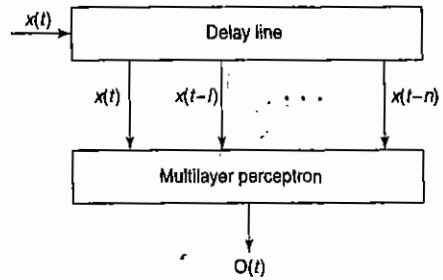


Figure 3-14 Time delay neural network (FIR filter).

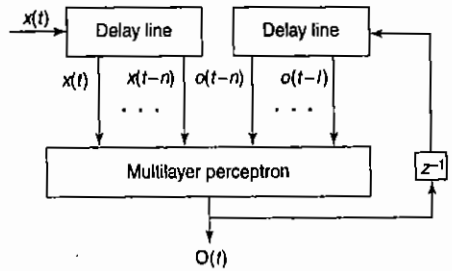


Figure 3-15 TDNN with output feedback (IIR filter).

3.8 Functional Link Networks

These networks are specifically designed for handling linearly non-separable problems using appropriate input representation. Thus, suitable enhanced representation of the input data has to be found out. This can be achieved by increasing the dimensions of the input space. The input data which is expanded is used for training instead of the actual input data. In this case, higher order input terms are chosen so that they are linearly independent of the original pattern components. Thus, the input representation has been enhanced and linear separability can be achieved in the extended space. One of the functional link model networks is shown in Figure 3-16. This model is helpful for learning continuous functions. For this model, the higher-order input terms are obtained using the orthogonal basis functions such as $\sin \pi x$, $\cos \pi x$, $\sin 2\pi x$, $\cos 2\pi x$, etc.

The most common example of linear nonseparability is XOR problem. The functional link networks help in solving this problem. The inputs now are

x_1	x_2	x_1x_2	t
−1	−1	1	1
−1	1	−1	−1
1	−1	−1	−1
1	1	1	1

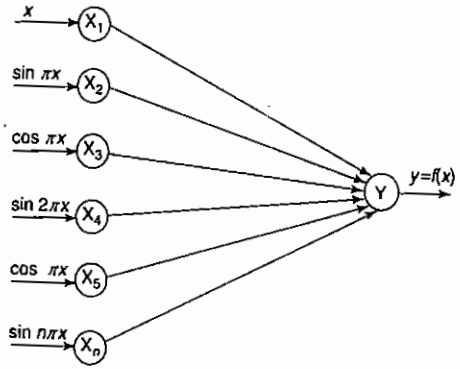


Figure 3-16 Functional line network model.

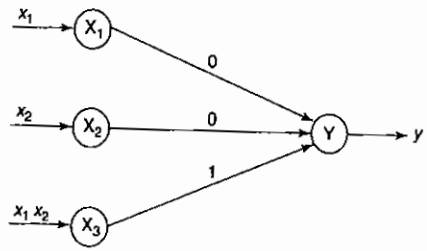


Figure 3-17 The XOR problem.

Thus, it can be easily seen that the functional link network in Figure 3-17 is used for solving this problem. The functional link network consists of only one layer, therefore, it can be trained using delta learning rule instead of the generalized delta learning rule used in BPN. As a result, the learning speed of the functional link network is faster than that of the BPN.

3.9 Tree Neural Networks

The tree neural networks (TNNs) are used for the pattern recognition problem. The main concept of this network is to use a small multilayer neural network at each decision-making node of a binary classification tree for extracting the non-linear features. TNNs completely extract the power of tree classifiers for using appropriate local features at the different levels and nodes of the tree. A binary classification tree is shown in Figure 3-18.

The decision nodes are present as circular nodes and the terminal nodes are present as square nodes. The terminal node has class label denoted by $\hat{c}$ associated with it. The rule base is formed in the decision node (splitting rule in the form of $f(x) < \theta$). The rule determines whether the pattern moves to the right or to the left. Here, $f(x)$ indicates the associated feature of pattern and "θ" is the threshold. The pattern will be given the class label of the terminal node on which it has landed. The classification here is based on the fact that the appropriate features can be selected at different nodes and levels in the tree. The output feature $y = f(x)$

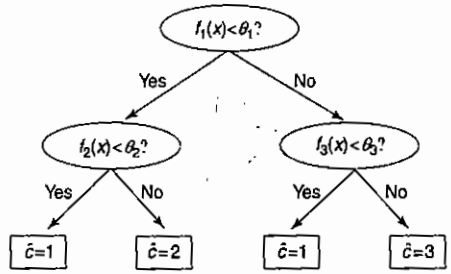


Figure 3-18 Binary classification tree.

obtained by a multilayer network at a particular decision node is used in the following way:

x directed to left child node t_L, if $y < 0$

x directed to right child node t_R, if $y \geq 0$

The algorithm for a TNN consists of two phases:

1. *Tree growing phase*: In this phase, a large tree is grown by recursively finding the rules for splitting until all the terminal nodes have pure or nearly pure class membership, else it cannot split further.
2. *Tree pruning phase*: Here a smaller tree is being selected from the pruned subtree to avoid the overfilling of data.

The training of TNN involves two nested optimization problems. In the inner optimization problem, the BPN algorithm can be used to train the network for a given pair of classes. On the other hand, in outer optimization problem, a heuristic search method is used to find a good pair of classes. The TNN when tested on a character recognition problem decreases the error rate and size of the tree relative to that of the standard classification tree design methods. The TNN can be implemented for waveform recognition problem. It obtains comparable error rates and the training here is faster than the large BPN for the same application. Also, TNN provides a structured approach to neural network classifier design problems.

3.10 Wavelet Neural Networks

The wavelet neural network (WNN) is based on the wavelet transform theory. This network helps in approximating arbitrary nonlinear functions. The powerful tool for function approximation is wavelet decomposition.

Let $f(x)$ be a piecewise continuous function. This function can be decomposed into a family of functions, which is obtained by dilating and translating a single wavelet function $\phi : R^n \rightarrow R$ as

$$f(x) = \sum_{i=1}^{n} w_i \det\left[D_i^{1/2}\right] \phi\left[D_i(x - t_i)\right]$$

where D_i is the diag(d_i), $d_i \in R_n^+$ are dilation vectors; D_i and t_i are the translational vectors; det [ ] is the determinant operator. The wavelet function ϕ selected should satisfy some properties. For selecting $\phi : R^n \rightarrow R$, the condition may be

$$\phi(x) = \phi_1(x_1) \cdots \phi_1(x_n) \quad \text{for } x = (x_1, x_2, \ldots, x_n)$$

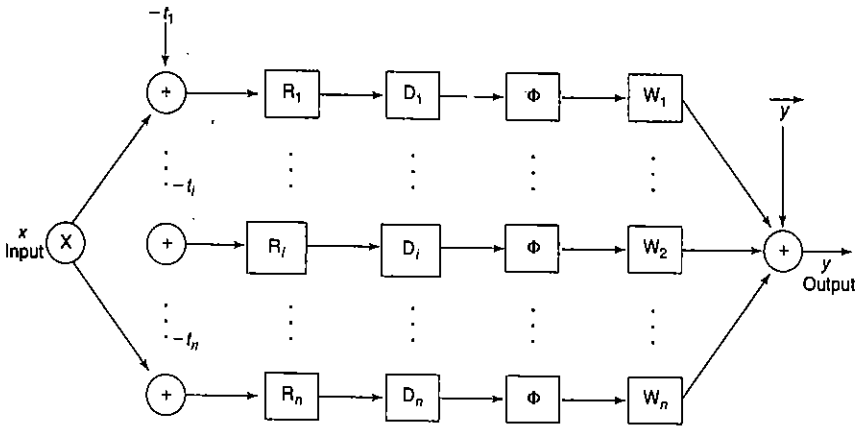


Figure 3-19 Wavelet neural network.

where

$$\phi_t(x) = -x\exp\left(\frac{-x^2}{2}\right)$$

is called scalar wavelet. The network structure can be formed based on the wavelet decomposition as

$$y(x) = \sum_{i=1}^{n} w_i \phi\,[D_i(x - t_i)] + \bar{y}$$

where $\bar{y}$ helps to deal with nonzero mean functions on finite domains. For proper dilation, a rotation can be made for better network operation:

$$y(x) = \sum_{i=1}^{n} w_i \phi\,[D_i R_i(x - t_i)] + \bar{y}$$

where R_i are the rotation matrices. The network which performs according to the above equation is called wavelet neural network. This is a combination of translation, rotation and dilation; and if a wavelet is lying on the same line, then it is called *wavelon* in comparison to the neurons in neural networks. The wavelet neural network is shown in Figure 3-19.

3.11 Summary

In this chapter we have discussed the supervised learning networks. In most of the classification and recognition problems, the widely used networks are the supervised learning networks. The architecture, the learning rule, flowchart for training process and training algorithm are discussed in detail for perceptron network, Adaline, Madaline, back-propagation network and radial basis function network. The perceptron network can be trained for single output classes as well as multioutput classes. Also, many Adaline networks combine together

to form a Madaline network. These networks are trained using delta learning rule. Back-propagation network is the most commonly used network in the real time applications. The error is back-propagated here and is fine tuned for achieving better performance. The basic difference between the back-propagation network and radial basis function network is the activation function used. The radial basis function network mostly uses Gaussian activation function. Apart from these networks, some special supervised learning networks such as time delay neural networks, functional link networks, tree neural networks and wavelet neural networks have also been discussed.

3.12 Solved Problems

1. Implement AND function using perceptron networks for bipolar inputs and targets.

Solution: Table 1 shows the truth table for AND function with bipolar inputs and targets:

Table 1

x_1	x_2	t
1	1	1
1	−1	−1
−1	1	−1
−1	−1	−1

The perceptron network, which uses perceptron learning rule, is used to train the AND function. The network architecture is as shown in Figure 1. The input patterns are presented to the network one by one. When all the four input patterns are presented, then one epoch is said to be completed. The initial weights and threshold are set to zero, i.e., $w_1 = w_2 = b = 0$ and $\theta = 0$. The learning rate α is set equal to 1.

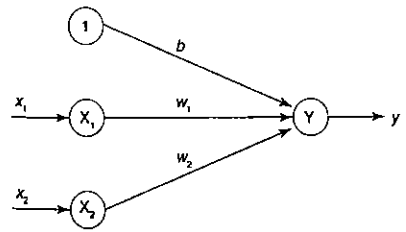


Figure 1 Perceptron network for AND function.

For the first input pattern, $x_1 = 1, x_2 = 1$ and $t = 1$, with weights and bias, $w_1 = 0$, $w_2 = 0$ and $b = 0$:

- Calculate the net input

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0 + 1 \times 0 + 1 \times 0 = 0$$

- The output y is computed by applying activations over the net input calculated:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } y_{in} = 0 \\ -1 & \text{if } y_{in} < 0 \end{cases}$$

Here we have taken $\theta = 0$. Hence, when, $y_{in} = 0$, $y = 0$.

- Check whether $t = y$. Here, $t = 1$ and $y = 0$, so $t \neq y$, hence weight updation takes place:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times 1 \times 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times 1 \times 1 = 1$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times 1 = 1$$

Here, the change in weights are

$$\Delta w_1 = \alpha t x_1;$$
$$\Delta w_2 = \alpha t x_2;$$
$$\Delta b = \alpha t$$

The weights $w_1 = 1$, $w_2 = 1$, $b = 1$ are the final weights after first input pattern is presented. The same process is repeated for all the input patterns. The process can be stopped when all the targets become equal to the calculated output or when a separating line is obtained using the final weights for separating the positive responses from negative responses. Table 2 shows the training of perceptron network until its

Table 2

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	−1	1	−1	1	1	−1	1	−1	0	2	0
−1	1	1	−1	2	1	+1	−1	−1	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1
EPOCH-2											
1	1	1	1	1	1	0	0	0	1	1	−1
1	−1	1	−1	−1	−1	0	0	0	1	1	−1
−1	1	1	−1	−1	−1	0	0	0	1	1	−1
−1	−1	1	−1	−3	−1	0	0	0	1	1	−1

target and calculated output converge for all the patterns.

The final weights and bias after second epoch are

$$w_1 = 1, w_2 = 1,\ b = -1$$

Since the threshold for the problem is zero, the equation of the separating line is

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

Here

$$w_1x_1 + w_2x_2 + b > \theta$$
$$w_1x_1 + w_2x_2 + b > 0$$

Thus, using the final weights we obtain

$$x_2 = -\frac{1}{1}x_1 - \frac{(-1)}{1}$$
$$x_2 = -x_1 + 1$$

It can be easily found that the above straight line separates the positive response and negative response region, as shown in Figure 2.

The same methodology can be applied for implementing other logic functions such as OR, AND-NOT, NAND, etc. If there exists a threshold value $\theta \neq 0$, then two separating lines have to be obtained, i.e., one to separate positive response from zero and the other for separating zero from the negative response.

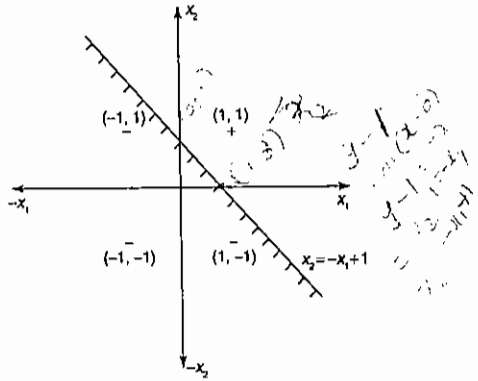


Figure 2 Decision boundary for AND function in perceptron training ($\theta = 0$).

2. Implement OR function with binary inputs and bipolar targets using perceptron training algorithm upto 3 epochs.

Solution: The truth table for OR function with binary inputs and bipolar targets is shown in Table 3.

Table 3

x_1	x_2	t
1	1	1
1	0	1
0	1	1
0	0	−1

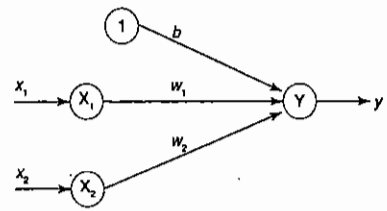


Figure 3 Perceptron network for OR function.

The perceptron network, which uses perceptron learning rule, is used to train the OR function. The network architecture is shown in Figure 3. The initial values of the weights and bias are taken as zero, i.e.,

$$w_1 = w_2 = b = 0$$

Also the learning rate is 1 and threshold is 0.2. So, the activation function becomes

$$f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \end{cases}$$

The network is trained as per the perceptron training algorithm and the steps are as in problem 1 (given for first pattern). Table 4 gives the network training for 3 epochs.

The final weights at the end of third epoch are

$$w_1 = 2, w_2 = 1, b = -1$$

Further epochs have to be done for the convergence of the network.

3. Find the weights using perceptron network for ANDNOT function when all the inputs are presented only one time. Use bipolar inputs and targets.

Solution: The truth table for ANDNOT function is shown in Table 5.

Table 5

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

The network architecture of ANDNOT function is shown as in Figure 4. Let the initial weights be zero and $\alpha = 1, \theta = 0$. For the first input sample, we compute the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$
$$= 0 + 1 \times 0 + 1 \times 0 = 0$$

Table 4

Input			Target	Net input	Calculated output	Weight changes			Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	Δw_1	Δw_2	Δb	w_1 (0	w_2 0	b 0)
EPOCH-1											
1	1	1	1	0	0	1	1	1	1	1	1
1	0	1	1	2	1	0	0	0	1	1	1
0	1	1	1	2	1	0	0	0	1	1	0
0	0	1	−1	1	1	0	0	−1	1	1	0
EPOCH-2											
1	1	1	1	2	1	0	0	0	1	1	0
1	0	1	1	1	1	0	0	0	1	1	0
0	1	1	1	1	1	0	0	0	1	1	0
0	0	1	−1	0	0	0	0	0	1	1	−1
EPOCH-3											
1	1	1	1	1	1	0	0	0	1	1	−1
1	0	1	1	0	0	1	0	1	2	1	0
0	1	1	1	1	1	0	0	0	2	1	0
0	0	1	−1	0	0	0	0	−1	2	1	−1

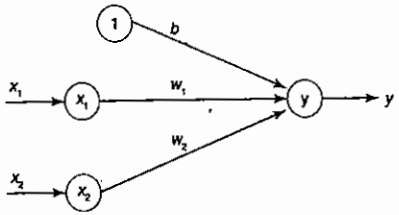


Figure 4 Network for ANDNOT function.

Applying the activation function over the net input, we obtain

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } -0 \le y_{in} \le 0 \\ -1 & \text{if } y_{in} < -0 \end{cases}$$

Hence, the output $y = f(y_{in}) = 0$. Since $t \neq y$, the new weights are computed as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times -1 \times 1 = -1$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times -1 \times 1 = -1$$

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times -1 = -1$$

The weights after presenting the first sample are

$$w = [-1 \ -1 \ -1]$$

For the second input sample, we calculate the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= -1 + 1 \times -1 + (-1 \times -1)$$

$$= -1 - 1 + 1 = -1$$

The output $y = f(y_{in})$ is obtained by applying activation function, hence $y = -1$.

Since $t \neq y$, the new weights are calculated as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = -1 + 1 \times 1 \times 1 = 0$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = -1 + 1 \times 1 \times -1 = -2$$

$$b(\text{new}) = b(\text{old}) + \alpha t = -1 + 1 \times 1 = 0$$

The weights after presenting the second sample are

$$w = [0 \ -2 \ 0]$$

For the third input sample, $x_1 = -1$, $x_2 = 1$, $t = -1$, the net input is calculated as,

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= 0 + -1 \times 0 + 1 \times -2 = 0 + 0 - 2 = -2$$

The output is obtained as $y = f(y_{in}) = -1$. Since $t = y$, no weight changes. Thus, even after presenting the third input sample, the weights are

$$w = [0 \ -2 \ 0]$$

For the fourth input sample, $x_1 = -1, x_2 = -1$, $t = -1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + x_1 w_1 + x_2 w_2$$

$$= 0 + -1 \times 0 + (-1 \times -2)$$

$$= 0 + 0 + 2 = 2$$

The output is obtained as $y = f(y_{in}) = 1$. Since $t \neq y$, the new weights on updating are given as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times -1 \times -1 = 1$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = -2 + 1 \times -1 \times -1 = -1$$

$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times -1 = -1$$

The weights after presenting fourth input sample are

$$w = [1 \ -1 \ -1]$$

One epoch of training for ANDNOT function using perceptron network is tabulated in Table 6.

Table 6

Input			Target	Net input	Calculated output	Weights		
x_1	x_2	1	(t)	(y_{in})	(y)	w_1 (0	w_2 0	b 0)
1	1	1	−1	0	0	−1	−1	−1
1	−1	1	1	−1	−1	0	−2	0
−1	1	1	−1	−2	−1	0	−2	0
−1	−1	1	−1	2	1	1	−1	−1

4. Find the weights required to perform the following classification using perceptron network. The vectors (1, 1, 1, 1) and (−1, 1 −1, −1) are belonging to the class (so have target value 1), vectors (1, 1, 1, −1) and (1, −1, −1, 1) are not belonging to the class (so have target value −1). Assume learning rate as 1 and initial weights as 0.

Solution: The truth table for the given vectors is given in Table 7.

Let $w_1 = w_2 = w_3 = w_4 = b = 0$ and the learning rate $\alpha = 1$. Since the threshold $\theta = 0.2$, so the activation function is

$$y = \begin{cases} 1 & \text{if } y_{in} > 0.2 \\ 0 & \text{if } -0.2 \le y_{in} \le 0.2 \\ -1 & \text{if } y_{in} < -0.2 \end{cases}$$

The net input is given by

$$y_{in} = b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4$$

The training is performed and the weights are tabulated in Table 8.

Table 7

Input					
x_1	x_2	x_3	x_4	b	Target (t)
1	1	1	1	1	1
−1	1	−1	−1	1	1
1	1	1	−1	1	−1
1	−1	−1	1	1	−1

Thus, in the third epoch, all the calculated outputs become equal to targets and the network has converged. The network convergence can also be checked by forming separating line equations for separating positive response regions from zero and zero from negative response region.

The network architecture is shown in Figure 5.

5. Classify the two-dimensional input pattern shown in Figure 6 using perceptron network. The symbol "*" indicates the data representation to be +1 and "•" indicates data to be −1. The patterns are I-F. For pattern I, the target is +1, and for F, the target is −1.

Table 8

Inputs					Target	Net input	Output	Weight changes					Weights				
$(x_1$	x_2	x_3	x_4	$b)$	(t)	(y_{in})	(y)	$(\Delta w_1$	Δw_2	Δw_3	Δw_4	$\Delta b)$	$(w_1$ (0	w_2 0	w_3 0	w_4 0	$b)$ 0)
EPOCH-1																	
(1	1	1	1	1)	1	0	0	1	1	1	1	1	1	1	1	1	1
(−1	1	−1	−1	1)	1	−1	−1	−1	1	−1	−1	1	0	2	0	0	2
(1	1	1	−1	1)	−1	4	1	−1	−1	−1	1	−1	−1	1	−1	1	1
(1	−1	−1	1	1)	−1	1	1	−1	1	1	−1	−1	−2	2	0	0	0
EPOCH-2																	
(1	1	1	1	1)	1	0	0	1	1	1	1	1	−1	3	1	1	1
(−1	1	−1	−1	1)	1	3	1	0	0	0	0	0	−1	3	1	1	1
(1	1	1	−1	1)	−1	4	1	−1	−1	−1	1	−1	−2	2	0	2	0
(1	−1	−1	1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0
EPOCH-3																	
(1	1	1	1	1)	1	2	1	0	0	0	0	0	−2	2	0	2	0
(−1	1	−1	−1	1)	1	2	1	0	0	0	0	0	−2	2	0	2	0
(1	1	1	−1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0
(1	−1	−1	1	1)	−1	−2	−1	0	0	0	0	0	−2	2	0	2	0

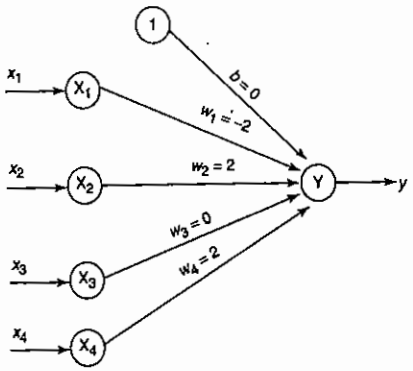


Figure 5 Network architecture.

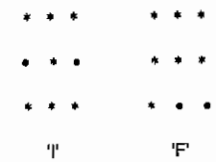


Figure 6 I-F data representation.

Solution: The training patterns for this problem are tabulated in Table 9.

Table 9

	Input									
Pattern	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	1	Target (t)
I	1	1	1	−1	1	−1	1	1	1	1	1
F	1	1	1	1	1	1	1	−1	−1	1	−1

The initial weights are all assumed to be zero, i.e., $\theta = 0$ and $\alpha = 1$. The activation function is given by

$$y = \begin{cases} 1 & \text{if } y_{in} > 0 \\ 0 & \text{if } -0 \le y_{in} \le 0 \\ -1 & \text{if } y_{in} < -0 \end{cases}$$

For the first input sample, $x_1 = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1]$, $t = 1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{9} x_i w_i$$

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + x_5 w_5 + x_6 w_6 + x_7 w_7 + x_8 w_8 + x_9 w_9$$
$$= 0 + 1 \times 0 + 1 \times 0 + 1 \times 0 + (-1) \times 0 + 1 \times 0 + (-1) \times 0 + 1 \times 0 + 1 \times 0$$
$$y_{in} = 0$$

Therefore, by applying the activation function the output is given by $y = f(y_{in}) = 0$. Now since $t \neq y$, the new weights are computed as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 0 + 1 \times 1 \times 1 = 1$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 0 + 1 \times 1 \times 1 = 1$$
$$w_3(\text{new}) = w_3(\text{old}) + \alpha t x_3 = 0 + 1 \times 1 \times 1 = 1$$
$$w_4(\text{new}) = w_4(\text{old}) + \alpha t x_4 = 0 + 1 \times 1 \times -1 = -1$$
$$w_5(\text{new}) = w_5(\text{old}) + \alpha t x_5 = 0 + 1 \times 1 \times 1 = 1$$
$$w_6(\text{new}) = w_6(\text{old}) + \alpha t x_6 = 0 + 1 \times 1 \times -1 = -1$$
$$w_7(\text{new}) = w_7(\text{old}) + \alpha t x_7 = 0 + 1 \times 1 \times 1 = 1$$
$$w_8(\text{new}) = w_8(\text{old}) + \alpha t x_8 = 0 + 1 \times 1 \times 1 = 1$$
$$w_9(\text{new}) = w_9(\text{old}) + \alpha t x_9 = 0 + 1 \times 1 \times 1 = 1$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 0 + 1 \times 1 = 1$$

The weights after presenting first input sample are

$$w = [1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ 1\ 1]$$

For the second input sample, $x_2 = [1\ 1\ 1\ 1\ 1\ 1\ 1\ -1\ -1\ 1]$, $t = -1$, the net input is calculated as

$$y_{in} = b + \sum_{i=1}^{9} x_i w_i$$

$$= b + x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4 + x_5 w_5 + x_6 w_6 + x_7 w_7 + x_8 w_8 + x_9 w_9$$
$$= 1 + 1 \times 1 + 1 \times 1 + 1 \times 1 + 1 \times -1 + 1 \times 1 + 1 \times -1 + 1 \times 1 + (-1) \times 1 + (-1) \times 1$$
$$y_{in} = 2$$

Therefore the output is given by $y = f(y_{in}) = 1$. Since $t \neq y$, the new weights are

$$w_1(\text{new}) = w_1(\text{old}) + \alpha t x_1 = 1 + 1 \times -1 \times 1 = 0$$
$$w_2(\text{new}) = w_2(\text{old}) + \alpha t x_2 = 1 + 1 \times -1 \times 1 = 0$$
$$w_3(\text{new}) = w_3(\text{old}) + \alpha t x_3 = 1 + 1 \times -1 \times 1 = 0$$
$$w_4(\text{new}) = w_4(\text{old}) + \alpha t x_4 = -1 + 1 \times -1 \times 1 = -2$$
$$w_5(\text{new}) = w_5(\text{old}) + \alpha t x_5 = 1 + 1 \times -1 \times 1 = 0$$
$$w_6(\text{new}) = w_6(\text{old}) + \alpha t x_6 = -1 + 1 \times -1 \times 1 = -2$$
$$w_7(\text{new}) = w_7(\text{old}) + \alpha t x_7 = 1 + 1 \times -1 \times 1 = 0$$
$$w_8(\text{new}) = w_8(\text{old}) + \alpha t x_8 = 1 + 1 \times -1 \times -1 = 2$$
$$w_9(\text{new}) = w_9(\text{old}) + \alpha t x_9 = 1 + 1 \times -1 \times -1 = 2$$
$$b(\text{new}) = b(\text{old}) + \alpha t = 1 + 1 \times -1 = 0$$

The weights after presenting the second input sample are

$$w = [0\ 0\ 0\ -2\ 0\ -2\ 0\ 2\ 2\ 0]$$

The network architecture is as shown in Figure 7. The network can be further trained for its convergence.

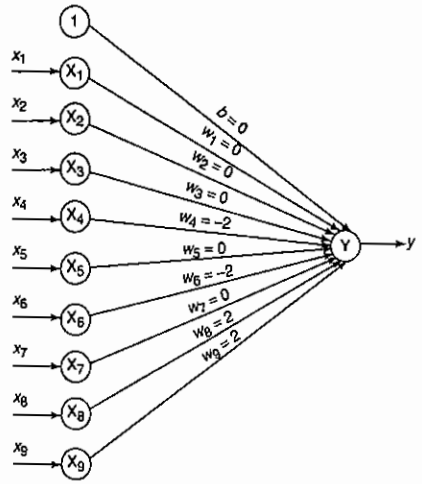


Figure 7 Network architecture.

6. Implement OR function with bipolar inputs and targets using Adaline network.

Solution: The truth table for OR function with bipolar inputs and targets is shown in Table 10.

Table 10

x_1	x_2	1	t
1	1	1	1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

Initially all the weights and links are assumed to be small random values, say 0.1, and the learning rate is also set to 0.1. Also here the least mean square error may be set. The weights are calculated until the least mean square error is obtained.

The initial weights are taken to be $w_1 = w_2 = b = 0.1$ and the learning rate $\alpha = 0.1$. For the first input sample, $x_1 = 1$, $x_2 = 1$, $t = 1$, we calculate the net input as

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i = b + \sum_{i=1}^{2} x_i w_i$$
$$= b + x_1 w_1 + x_2 w_2$$
$$= 0.1 + 1 \times 0.1 + 1 \times 0.1 = 0.3$$

Now compute $(t - y_{in}) = (1 - 0.3) = 0.7$. Updating the weights we obrain,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$$

where $\alpha(t - y_{in})x_i$ is called as weight change Δw_i. The new weights are obtained as

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.1 + 0.1 \times 0.7 \times 1$$
$$= 0.1 + 0.07 = 0.17$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1$$
$$+ 0.1 \times 0.7 \times 1 = 0.17$$
$$b(\text{new}) = b(\text{old}) + \Delta b = 0.1 + 0.1 \times 0.7 = 0.17$$

where

$$\Delta w_1 = \alpha(t - y_{in})x_1$$
$$\Delta w_2 = \alpha(t - y_{in})x_2$$
$$\Delta b = \alpha(t - y_{in})$$

Now we calculate the error:

$$E = (t - y_{in})^2 = (0.7)^2 = 0.49$$

The final weights after presenting first input sample are

$$w = [0.17\ \ 0.17\ \ 0.17]$$

and error $E = 0.49$.

These calculations are performed for all the input samples and the error is calculated. One epoch is completed when all the input patterns are presented. Summing up all the errors obtained for each input sample during one epoch will give the total mean square error of that epoch. The network training is continued until this error is minimized to a very small value.

Adopting the method above, the network training is done for OR function using Adaline network and is tabulated below in Table 11 for $\alpha = 0.1$.

The total mean square error after each epoch is given as in Table 12.

Thus from Table 12, it can be noticed that as training goes on, the error value gets minimized. Hence, further training can be continued for further minimization of error. The network architecture of Adaline network for OR function is shown in Figure 8.

Table 12

Epoch	Total mean square error
Epoch 1	3.02
Epoch 2	1.938
Epoch 3	1.5506
Epoch 4	1.417
Epoch 5	1.377

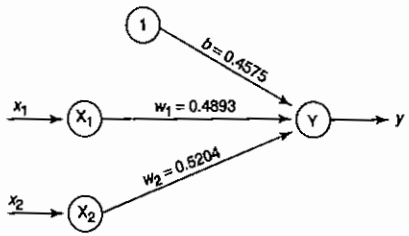


Figure 8 Network architecture of Adaline.

Table 11

Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t-y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.1	w_2 0.1	b 0.1)	$(t-y_{in})^2$
EPOCH-1												
1	1	1	1	0.3	0.7	0.07	0.07	0.07	0.17	0.17	0.17	0.49
1	−1	1	1	0.17	0.83	0.083	−0.083	0.083	0.253	0.087	0.253	0.69
−1	1	1	1	0.087	0.913	−0.0913	0.0913	0.0913	0.1617	0.1783	0.3443	0.83
−1	−1	1	−1	0.0043	−1.0043	0.1004	0.1004	−0.1004	0.2621	0.2787	0.2439	1.01
EPOCH-2												
1	1	1	1	0.7847	0.2153	0.0215	0.0215	0.0215	0.2837	0.3003	0.2654	0.046
1	−1	1	1	0.2488	0.7512	0.7512	−0.0751	0.0751	0.3588	0.2251	0.3405	0.564
−1	1	1	1	0.2069	0.7931	−0.7931	0.0793	0.0793	0.2795	0.3044	0.4198	0.629
−1	−1	1	−1	−0.1641	−0.8359	0.0836	0.0836	−0.0836	0.3631	0.388	0.336	0.699
EPOCH-3												
1	1	1	1	1.0873	−0.0873	−0.087	−0.087	−0.087	0.3543	0.3793	0.3275	0.0076
1	−1	1	1	0.3025	+0.6975	0.0697	−0.0697	0.0697	0.4241	0.3096	0.3973	0.487
−1	1	1	1	0.2827	0.7173	−0.0717	0.0717	0.0717	0.3523	0.3813	0.469	0.515
−1	−1	1	−1	−0.2647	−0.7353	0.0735	0.0735	−0.0735	0.4259	0.4548	0.3954	0.541
EPOCH-4												
1	1	1	1	1.2761	−0.2761	−0.0276	−0.0276	−0.0276	0.3983	0.4272	0.3678	0.076
1	−1	1	1	0.3389	0.6611	0.0661	−0.0661	0.0661	0.4644	0.3611	0.4339	0.437
−1	1	1	1	0.3307	0.6693	−0.0669	0.0669	0.0699	0.3974	0.428	0.5009	0.448
−1	−1	1	−1	−0.3246	−0.6754	0.0675	0.0675	−0.0675	0.465	0.4956	0.4333	0.456
EPOCH-5												
1	1	1	1	1.3939	−0.3939	−0.0394	−0.0394	−0.0394	0.4256	0.4562	0.393	0.155
1	−1	1	1	0.3634	0.6366	0.0637	−0.0637	0.0637	0.4893	0.3925	0.457	0.405
−1	1	1	1	0.3609	0.6391	−0.0639	0.0639	0.0639	0.4253	0.4654	0.5215	0.408
−1	−1	1	−1	−0.3603	−0.6397	0.064	0.064	−0.064	0.4893	0.5204	0.4575	0.409

7. Use Adaline network to train ANDNOT function with bipolar inputs and targets. Perform 2 epochs of training.

Solution: The truth table for ANDNOT function with bipolar inputs and targets is shown in Table 13.

Table 13

x_1	x_2	1	t
1	1	1	−1
1	−1	1	1
−1	1	1	−1
−1	−1	1	−1

Initially the weights and bias have assumed a random value say 0.2. The learning rate is also set to 0.2. The weights are calculated until the least mean square error is obtained. The initial weights are $w_1 = w_2 = b = 0.2$, and $\alpha = 0.2$. For the first input sample $x_1 = 1$, $x_2 = 1$, $t = -1$, we calculate the net input as

$$y_{in} = b + x_1 w_1 + x_2 w_2$$
$$= 0.2 + 1 \times 0.2 + 1 \times 0.2 = 0.6$$

Now compute $(t - y_{in}) = (-1 - 0.6) = -1.6$. Updating the weights we obtain

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y_{in})x_i$$

The new weights are obtained as

$$w_1(\text{new}) = w_1(\text{old}) + \alpha(t - y_{in})x_1$$
$$= 0.2 + 0.2 \times (-1.6) \times 1 = -0.12$$

$$w_2(\text{new}) = w_2(\text{old}) + \alpha(t - y_{in})x_2$$
$$= 0.2 + 0.2 \times (-1.6) \times 1 = -0.12$$

$$b(\text{new}) = b(\text{old}) + \alpha(t - y_{in})$$
$$= 0.2 + 0.2 \times (-1.6) = -0.12$$

Now we compute the error,

$$E = (t - y_{in})^2 = (-1.6)^2 = 2.56$$

The final weights after presenting first input sample are $w = [-0.12 - 0.12 - 0.12]$ and error $E = 2.56$.

The operational steps are carried for 2 epochs of training and network performance is noted. It is tabulated as shown in Table 14.

The total mean square error at the end of two epochs is summation of the errors of all input samples as shown in Table 15.

Table 15

Epoch	Total mean square error
Epoch 1	5.71
Epoch 2	2.43

Hence from Table 15, it is clearly understood that the mean square error decreases as training progresses. Also, it can be noted that at the end of the sixth epoch, the error becomes approximately equal to 1. The network architecture for ANDNOT function using Adaline network is shown in Figure 9.

Table 14

Inputs			Target	Net input		Weight changes			Weights			Error
x_1	x_2	1	t	y_{in}	$(t-y_{in})$	Δw_1	Δw_2	Δb	w_1 (0.2	w_2 0.2	b 0.2)	$(t-y_{in})^2$
EPOCH-1												
1	1	1	−1	0.6	−1.6	−0.32	−0.32	−0.32	−0.12	−0.12	−0.12	2.56
1	−1	1	1	−0.12	1.12	0.22	−0.22	0.22	0.10	−0.34	0.10	1.25
−1	1	1	−1	−0.34	−0.66	0.13	−0.13	−0.13	0.24	−0.48	−0.03	0.43
−1	−1	1	−1	0.21	−1.2	0.24	0.24	−0.24	0.48	−0.23	−0.27	1.47
EPOCH-2												
1	1		−1	−0.02	−0.98	−0.195	−0.195	−0.195	0.28	−0.43	−0.46	0.95
1	−1	1	1	0.25	0.76	0.15	−0.15	0.15	0.43	−0.58	−0.31	0.57
−1	1	1	−1	−1.33	0.33	−0.065	0.065	0.065	0.37	−0.51	−0.25	0.106
−1	−1	1	−1	−0.11	−0.90	0.18	0.18	−0.18	0.55	−0.38	0.43	0.8

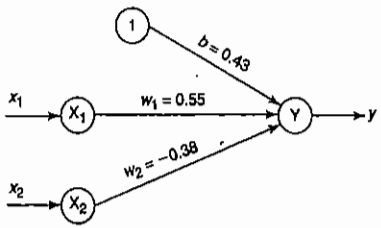


Figure 9 Network architecture for ANDNOT function using Adaline network.

8 Using Madaline network, implement XOR function with bipolar inputs and targets. Assume the required parameters for training of the network.

Solution: The training pattern for XOR function is given in Table 16.

Table 16

x_1	x_2	1	t
1	1	1	−1
1	−1	1	1
−1	1	1	1
−1	−1	1	−1

The Madaline Rule I (MRI) algorithm in which the weights between the hidden layer and output layer remain fixed is used for training the network. Initializing the weights to small random values, the network architecture is as shown in Figure 10, with initial weights. From Figure 10, the initial weights and bias are $[w_{11}\ w_{21}\ b_1] = [0.05\ 0.2\ 0.3]$, $[w_{12}\ w_{22}\ b_2] = [0.1\ 0.2\ 0.15]$ and $[v_1\ v_2\ b_3] = [0.5\ 0.5\ 0.5]$. For first

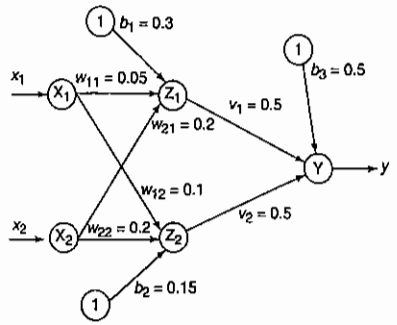


Figure 10 Network architecture of Madaline for XOR functions (initial weights given).

input sample, $x_1 = 1$, $x_2 = 1$, target $t = -1$, and learning rate α equal to 0.5:

- Calculate net input to the hidden units:

$$z_{in1} = b_1 + x_1 w_{11} + x_2 w_{21}$$
$$= 0.3 + 1 \times 0.05 + 1 \times 0.2 = 0.55$$
$$z_{in2} = b_2 + x_1 w_{12} + x_2 w_{22}$$
$$= 0.15 + 1 \times 0.1 + 1 \times 0.2 = 0.45$$

- Calculate the output z_1, z_2 by applying the activations over the net input computed. The activation function is given by

$$f(z_{in}) = \begin{cases} 1 & \text{if } z_{in} \geq 0 \\ -1 & \text{if } z_{in} < 0 \end{cases}$$

Hence,

$$z_1 = f(z_{in1}) = f(0.55) = 1$$
$$z_2 = f(z_{in2}) = f(0.45) = 1$$

- After computing the output of the hidden units, then find the net input entering into the output unit:

$$y_{in} = b_3 + z_1 v_1 + z_2 v_2$$
$$= 0.5 + 1 \times 0.5 + 1 \times 0.5 = 1.5$$

- Apply the activation function over the net input y_{in} to calculate the output y:

$$y = f(y_{in}) = f(1.5) = 1$$

- Since $t \neq y$, weight updation has to be performed. Also since $t = -1$, the weights are updated on z_1 and z_2 that have positive net input. Since here both net inputs z_{in1} and z_{in2} are positive, updating the weights and bias on both hidden units, we obtain

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(t - z_{inj})x_i$$
$$b_j(\text{new}) = b_j(\text{old}) + \alpha(t - z_{inj})$$

This implies:

$$w_{11}(\text{new}) = w_{11}(\text{old}) + \alpha(t - z_{in1})x_1$$
$$= 0.05 + 0.5(-1 - 0.55) \times 1 = -0.725$$
$$w_{12}(\text{new}) = w_{12}(\text{old}) + \alpha(t - z_{in2})x_1$$
$$= 0.1 + 0.5(-1 - 0.45) \times 1 = -0.625$$
$$b_1(\text{new}) = b_1(\text{old}) + \alpha(t - z_{in1})$$
$$= 0.3 + 0.5(-1 - 0.55) = -0.475$$

$$w_{21}(\text{new}) = w_{21}(\text{old}) + \alpha(t - z_{in1})x_2$$
$$= 0.2 + 0.5(-1 - 0.55) \times 1 = -0.575$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + \alpha(t - z_{in2})x_2$$
$$= 0.2 + 0.5(-1 - 0.45) \times 1 = -0.525$$
$$b_2(\text{new}) = b_2(\text{old}) + \alpha(t - z_{in2})$$
$$= 0.15 + 0.5(-1 - 0.45) = -0.575$$

All the weights and bias between the input layer and hidden layer are adjusted. This completes the training for the first epoch. The same process is repeated until the weight converges. It is found that the weight converges at the end of 3 epochs. Table 17 shows the training performance of Madaline network for XOR function.

The network architecture for Madaline network with final weights for XOR function is shown in Figure 11.

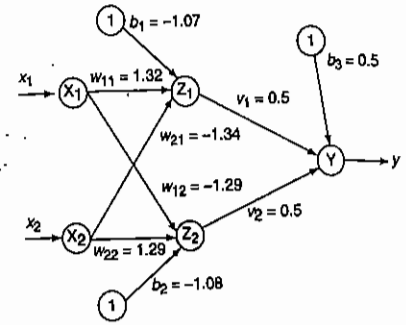


Figure 11 Madaline network for XOR function (final weights given).

9. Using back-propagation network, find the new weights for the net shown in Figure 12. It is presented with the input pattern [0, 1] and the target output is 1. Use a learning rate $\alpha = 0.25$ and binary sigmoidal activation function.

Solution: The new weights are calculated based on the training algorithm in Section 3.5.4. The initial weights are $[v_{11}\ v_{21}\ v_{01}] = [0.6\ -0.1\ 0.3]$,

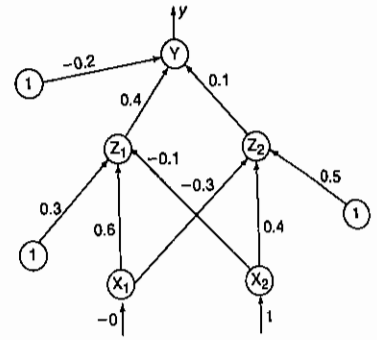


Figure 12 Network.

Table 17

Inputs			Target											
x_1	x_2	1	(t)	z_{in1}	z_{in2}	z_1	z_2	y_{in}	y	w_{11}	w_{21}	b_1	w_{12}	w_{22}	b_2
EPOCH-1															
1	1	1	−1	0.55	0.45	1	1	1.5	1	−0.725	−0.58	−0.475	−0.625	−0.525	−0.575
1	−1	1	1	−0.625	−0.675	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−0.625	−0.525	−0.575
−1	1	1	1	−1.1375	−0.475	−1	−1	−0.5	−1	0.0875	−1.39	0.34	−1.3625	0.2125	0.1625
−1	−1	1	−1	1.6375	1.3125	1	1	1.5	1	1.4065	−0.069	−0.98	−0.207	1.369	−0.994
EPOCH-2															
1	1	1	−1	0.3565	0.168	1	1	1.5	1	0.7285	−0.75	−1.66	−0.791	−0.207	−1.58
1	−1	1	1	−0.1845	−3.154	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−0.791	0.785	−1.58
−1	1	1	1	−3.728	−0.002	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	0.785	−1.08
−1	−1	1	−1	−1.0495	−1.071	−1	−1	−0.5	−1	1.3205	−1.34	−1.068	−1.29	1.29	−1.08
EPOCH-3															
1	1	1	−1	−1.0865	−1.083	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
1	−1	1	1	1.5915	−3.655	1	−1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	1	1	1	−3.728	1.501	−1	1	0.5	1	1.32	−1.34	−1.07	−1.29	1.29	−1.08
−1	−1	1	−1	−1.0495	−1.701	−1	−1	−0.5	−1	1.32	−1.34	−1.07	−1.29	1.29	−1.08

$[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$. Activation function used is binary sigmoidal activation function and is given by

$$f(x) = \frac{1}{1+e^{-x}}$$

Given the output sample $[x_1, x_2] = [0, 1]$ and target $t = 1$,

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + 0 \times 0.6 + 1 \times -0.1 = 0.2$$

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + 0 \times -0.3 + 1 \times 0.4 = 0.9$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1}{1+e^{-z_{in1}}} = \frac{1}{1+e^{-0.2}} = 0.5498$$

$$z_2 = f(z_{in2}) = \frac{1}{1+e^{-z_{in2}}} = \frac{1}{1+e^{-0.9}} = 0.7109$$

- Calculate the net input entering the output layer. For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + 0.5498 \times 0.4 + 0.7109 \times 0.1$$
$$= 0.09101$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1}{1+e^{-y_{in}}} = \frac{1}{1+e^{-0.09101}} = 0.5227$$

- Compute the error portion δ_k:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = f(y_{in})[1 - f(y_{in})] = 0.5227[1 - 0.5227]$$
$$f'(y_{in}) = 0.2495$$

This implies

$$\delta_1 = (1 - 0.5227)\ (0.2495) = 0.1191$$

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.1191 \times 0.5498$$
$$= 0.0164$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.1191 \times 0.7109$$
$$= 0.02117$$

$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.1191 = 0.02978$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj})$$

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$
$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.1191 \times 0.4 = 0.04764$$
$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.1191 \times 0.1 = 0.01191$$

Error, $\delta_1 = \delta_{in1} f'(z_{in1})$

$$f'(z_{in1}) = f(z_{in1})[1 - f(z_{in1})]$$
$$= 0.5498[1 - 0.5498] = 0.2475$$
$$\delta_1 = \delta_{in1} f'(z_{in1})$$
$$= 0.04764 \times 0.2475 = 0.0118$$

Error, $\delta_2 = \delta_{in2} f'(z_{in2})$

$$f'(z_{in2}) = f(z_{in2})[1 - f(z_{in2})]$$
$$= 0.7109[1 - 0.7109] = 0.2055$$
$$\delta_2 = \delta_{in2} f'(z_{in2})$$
$$= 0.01191 \times 0.2055 = 0.00245$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.0118 \times 0 = 0$$
$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.0118 \times 1 = 0.00295$$
$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.0118 = 0.00295$$
$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.00245 \times 0 = 0$$
$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.00245 \times 1 = 0.0006125$$
$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.00245 = 0.0006125$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 + 0 = 0.6$$
$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 + 0 = -0.3$$
$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21}$$
$$= -0.1 + 0.00295 = -0.09705$$
$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22}$$
$$= 0.4 + 0.0006125 = 0.4006125$$
$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 + 0.0164$$
$$= 0.4164$$
$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.02117$$
$$= 0.12117$$
$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.00295$$
$$= 0.30295$$
$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02}$$
$$= 0.5 + 0.0006125 = 0.5006125$$
$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.02978$$
$$= -0.17022$$

Thus, the final weights have been computed for the network shown in Figure 12.

10. Find the new weights, using back-propagation network for the network shown in Figure 13. The network is presented with the input pattern $[-1, 1]$ and the target output is $+1$. Use a learning rate of $\alpha = 0.25$ and bipolar sigmoidal activation function.

Solution: The initial weights are $[v_{11} v_{21} v_{01}] = [0.6\ -0.1\ 0.3]$, $[v_{12}\ v_{22}\ v_{02}] = [-0.3\ 0.4\ 0.5]$ and $[w_1\ w_2\ w_0] = [0.4\ 0.1\ -0.2]$, and the learning rate is $\alpha = 0.25$.

Activation function used is binary sigmoidal activation function and is given by

$$f(x) = \frac{2}{1+e^{-x}} - 1 = \frac{1-e^{-x}}{1+e^{-x}}$$

Given the input sample $[x_1, x_2] = [-1, 1]$ and target $t = 1$:

- Calculate the net input: For z_1 layer

$$z_{in1} = v_{01} + x_1 v_{11} + x_2 v_{21}$$
$$= 0.3 + (-1) \times 0.6 + 1 \times -0.1 = -0.4$$

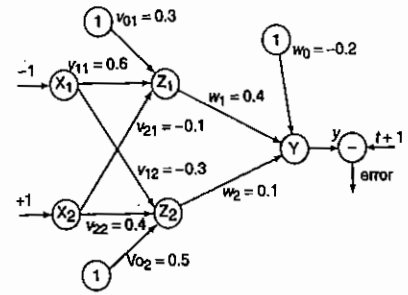


Figure 13 Network.

For z_2 layer

$$z_{in2} = v_{02} + x_1 v_{12} + x_2 v_{22}$$
$$= 0.5 + (-1) \times -0.3 + 1 \times 0.4 = 1.2$$

Applying activation to calculate the output, we obtain

$$z_1 = f(z_{in1}) = \frac{1-e^{-z_{in1}}}{1+e^{-z_{in1}}} = \frac{1-e^{0.4}}{1+e^{0.4}} = -0.1974$$

$$z_2 = f(z_{in2}) = \frac{1-e^{-z_{in1}}}{1+e^{-z_{in2}}} = \frac{1-e^{-1.2}}{1+e^{-1.2}} = 0.537$$

- Calculate the net input entering the output layer. For y layer

$$y_{in} = w_0 + z_1 w_1 + z_2 w_2$$
$$= -0.2 + (-0.1974) \times 0.4 + 0.537 \times 0.1$$
$$= -0.22526$$

Applying activations to calculate the output, we obtain

$$y = f(y_{in}) = \frac{1-e^{-y_{in}}}{1+e^{-y_{in}}} = \frac{1-e^{0.22526}}{1+e^{0.22526}} = -0.1122$$

- Compute the error portion δ_k:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

Now

$$f'(y_{in}) = 0.5[1 + f(y_{in})][1 - f(y_{in})]$$
$$= 0.5[1 - 0.1122][1 + 0.1122] = 0.4937$$

This implies

$$\delta_1 = (1 + 0.1122)\,(0.4937) = 0.5491$$

Find the changes in weights between hidden and output layer:

$$\Delta w_1 = \alpha \delta_1 z_1 = 0.25 \times 0.5491 \times -0.1974 = -0.0271$$

$$\Delta w_2 = \alpha \delta_1 z_2 = 0.25 \times 0.5491 \times 0.537 = 0.0737$$

$$\Delta w_0 = \alpha \delta_1 = 0.25 \times 0.5491 = 0.1373$$

- Compute the error portion δ_j between input and hidden layer ($j = 1$ to 2):

$$\delta_j = \delta_{inj} f'(z_{inj})$$

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

$$\delta_{inj} = \delta_1 w_{j1} \quad [\because \text{only one output neuron}]$$

$$\Rightarrow \delta_{in1} = \delta_1 w_{11} = 0.5491 \times 0.4 = 0.21964$$

$$\Rightarrow \delta_{in2} = \delta_1 w_{21} = 0.5491 \times 0.1 = 0.05491$$

$$\text{Error, } \delta_1 = \delta_{in1} f'(z_{in1}) = 0.21964 \times 0.5 \times (1 + 0.1974)(1 - 0.1974) = 0.1056$$

$$\text{Error, } \delta_2 = \delta_{in2} f'(z_{in2}) = 0.05491 \times 0.5 \times (1 - 0.537)(1 + 0.537) = 0.0195$$

Now find the changes in weights between input and hidden layer:

$$\Delta v_{11} = \alpha \delta_1 x_1 = 0.25 \times 0.1056 \times -1 = -0.0264$$

$$\Delta v_{21} = \alpha \delta_1 x_2 = 0.25 \times 0.1056 \times 1 = 0.0264$$

$$\Delta v_{01} = \alpha \delta_1 = 0.25 \times 0.1056 = 0.0264$$

$$\Delta v_{12} = \alpha \delta_2 x_1 = 0.25 \times 0.0195 \times -1 = -0.0049$$

$$\Delta v_{22} = \alpha \delta_2 x_2 = 0.25 \times 0.0195 \times 1 = 0.0049$$

$$\Delta v_{02} = \alpha \delta_2 = 0.25 \times 0.0195 = 0.0049$$

- Compute the final weights of the network:

$$v_{11}(\text{new}) = v_{11}(\text{old}) + \Delta v_{11} = 0.6 - 0.0264 = 0.5736$$

$$v_{12}(\text{new}) = v_{12}(\text{old}) + \Delta v_{12} = -0.3 - 0.0049 = -0.3049$$

$$v_{21}(\text{new}) = v_{21}(\text{old}) + \Delta v_{21} = -0.1 + 0.0264 = -0.0736$$

$$v_{22}(\text{new}) = v_{22}(\text{old}) + \Delta v_{22} = 0.4 + 0.0049 = 0.4049$$

$$w_1(\text{new}) = w_1(\text{old}) + \Delta w_1 = 0.4 - 0.0271 = 0.3729$$

$$w_2(\text{new}) = w_2(\text{old}) + \Delta w_2 = 0.1 + 0.0737 = 0.1737$$

$$v_{01}(\text{new}) = v_{01}(\text{old}) + \Delta v_{01} = 0.3 + 0.0264 = 0.3264$$

$$v_{02}(\text{new}) = v_{02}(\text{old}) + \Delta v_{02} = 0.5 + 0.0049 = 0.5049$$

$$w_0(\text{new}) = w_0(\text{old}) + \Delta w_0 = -0.2 + 0.1373 = -0.0627$$

Thus, the final weight has been computed for the network shown in Figure 13.

3.13 Review Questions

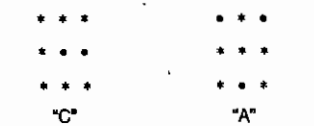
1. What is supervised learning and how is it different from unsupervised learning?
2. How does learning take place in supervised learning?
3. From a mathematical point of view, what is the process of learning in supervised learning?
4. What is the building block of the perceptron?
5. Does perceptron require supervised learning? If no, what does it require?
6. List the limitations of perceptron.
7. State the activation function used in perceptron network.
8. What is the importance of threshold in perceptron network?
9. Mention the applications of perceptron network.
10. What are feature detectors?
11. With a neat flowchart, explain the training process of perceptron network.
12. What is the significance of error signal in perceptron network?

13. State the testing algorithm used in perceptron algorithm.
14. How is the linear separability concept implemented using perceptron network training?
15. Define perceptron learning rule.
16. Define delta rule.
17. State the error function for delta rule.
18. What is the drawback of using optimization algorithm?
19. What is Adaline?
20. Draw the model of an Adaline network.
21. Explain the training algorithm used in Adaline network.
22. How is a Madaline network formed?
23. Is it true that Madaline network consists of many perceptrons?
24. State the characteristics of weighted interconnections between Adaline and Madaline.
25. How is training adopted in Madaline network using majority vote rule?
26. State few applications of Adaline and Madaline.
27. What is meant by epoch in training process?
28. What is meant by gradient descent method?
29. State the importance of back-propagation algorithm.
30. What is called as memorization and generalization?
31. List the stages involved in training of back-propagation network.
32. Draw the architecture of back-propagation algorithm.
33. State the significance of error portions δ_k and δ_j in BPN algorithm.
34. What are the activations used in back-propagation network algorithm?
35. What is meant by local minima and global minima?
36. Derive the generalized delta learning rule.
37. Derive the derivations of the binary and bipolar sigmoidal activation function.
38. What are the factors that improve the convergence of learning in BPN network?
39. What is meant by incremental learning?
40. Why is gradient descent method adopted to minimize error?
41. What are the methods of initialization of weights?
42. What is the necessity of momentum factor in weight updation process?
43. Define "over fitting" or "over training."
44. State the techniques for proper choice of learning rate.
45. What are the limitations of using momentum factor?
46. How many hidden layers can there be in a neural network?
47. What is the activation function used in radial basis function network?
48. Explain the training algorithm of radial basis function network.
49. By what means can an IIR and an FIR filter be formed in neural network?
50. What is the importance of functional link network?
51. Write a short note on binary classification tree neural network.
52. Explain in detail about wavelet neural network.

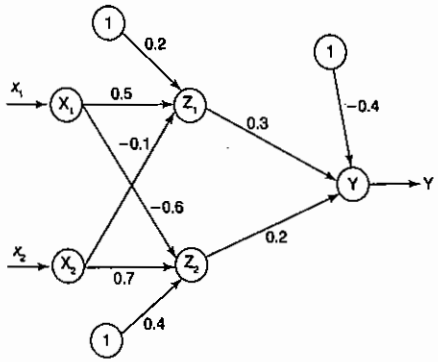
3.14 Exercise Problems

1. Implement NOR function using perceptron network for bipolar inputs and targets.
2. Find the weights required to perform the following classifications using perceptron network. The vectors (1, 1, −1, −1) and (1, −1, 1, −1) are belonging to the class (so have target value 1), vector (−1, −1, −1, 1) and (−1, −1, 1 1) are not belonging to the class (so have target value −1). Assume learning rate 1 and initial weights as 0.

3. Classify the two-dimensional pattern shown in figure below using perceptron network.



4. Implement AND function using Adaline network.
5. Using the delta rule, find the weights required to perform following classifications: Vectors (1, 1, −1, −1) and (−1, −1, −1, −1) are belonging to the class having target value 1; vectors (1, 1, 1, 1) and (−1, −1, 1, −1) are not belonging to the class having target value −1. Use a learning rate of 0.5 and assume random value of weights. Also, using each of the training vectors as input, test the response of the net.
6. Implement AND function using Madaline network.
7. With suitable example, discuss the perceptron network training with and without bias.
8. Using back-propagation network, find the new weights for the network shown in the following figure. The network is presented with the input pattern [1, 0] and target output 1. Use learning rate of $\alpha = 0.3$ and binary sigmoidal activation function.



9. Find the new weights for the network given in the above problem using back-propagation network. The network is presented the input pattern [1, −1] and target output +1. Use learning rate of $\alpha = 0.3$ and bipolar sigmoidal activation function.
10. Find the new weights for the activation function with the network shown in problem 8 using BPN. The network is presented with the input pattern [−1, 1] and target output −1. Use learning rate of $\alpha = 0.45$ and suitable activation function.

3.15 Projects

1. Classify upper case letters and lower case letters using perceptron network. Use as many output units based on training set as possible. Test the network with noisy pattern as well.
2. Write a suitable computer program to classify the numbers between 0–9 using Adaline network.
3. Write a computer program to train a Madaline to perform AND function using MRI algorithm.
4. Write a program for implementing BPN for training a single hidden layer back-propagation network with bipolar sigmoidal units ($x = 1$) to achieve the following two-to-one mappings.

   - $y = 6\sin(\pi x_1) + \cos(\pi x_2)$
   - $y = \sin(\pi x_1) + \cos(0.2\pi x_2)$

   Set up two sets of data, each consisting of 10 input–output pairs, one for training and other for testing. The input–output data are obtained by varying input variables (x_1, x_2) within [−1, +1] randomly. Also the output data is normalized within [−1, 1]. Apply training to find proper weights in the network.

4 Associative Memory Networks

Learning Objectives

- Gives details on associative memories.
- Discusses the training algorithm used for pattern association networks – Hebb rule and outer products rule.
- The architecture, flowchart for training process, training algorithm and testing algorithm of autoassociative, heteroassociative and bidirectional associative memory are discussed in detail.
- Variants of BAM – continuous BAM and discrete BAM are included.
- Hopfield network with its electrical model is described with training algorithm.
- Analysis of energy function was performed for BAM, discrete and continuous Hopfield networks.
- An overview is given on the iterative autoassociative network – linear autoassociator memory brain-in-the-box network and autoassociator with threshold unit.
- Also temporal associative memory is discussed in brief.

4.1 Introduction

An associative memory network can store a set of patterns as memories. When the associative memory is being presented with a key pattern, it responds by producing one of the stored patterns, which closely resembles or relates to the key pattern. Thus, the recall is through association of the key pattern, with the help of information memorized. These types of memories are also called as *content-addressable memories* (CAM) in contrast to that of traditional *address-addressable memories* in digital computers where stored pattern (in bytes) is recalled by its address. It is also a matrix memory as in RAM/ROM. The CAM can also be viewed as associating data to address, i.e., for every data in the memory there is a corresponding unique address. Also, it can be viewed as data correlator. Here input data is correlated with that of the stored data in the CAM. It should be noted that the stored patterns must be unique, i.e., different patterns in each location. If the same pattern exists in more than one location in the CAM, then, even though the correlation is correct, the address is noted to be ambiguous. The basic structure of CAM is given in Figure 4-1.

Associative memory makes a parallel search within a stored data file. The concept behind this search is to output any one or all stored items which match the given search argument and to retrieve the stored data either completely or partially.

Two types of associative memories can be differentiated. They are *autoassociative memory* and *heteroassociative memory*. Both these nets are single-layer nets in which the weights are determined in a manner that the net stores a set of pattern associations. Each of this association is an input–output vector pair, say, s:t. If each of the output vectors is same as the input vectors with which it is associated, then the net is a said to

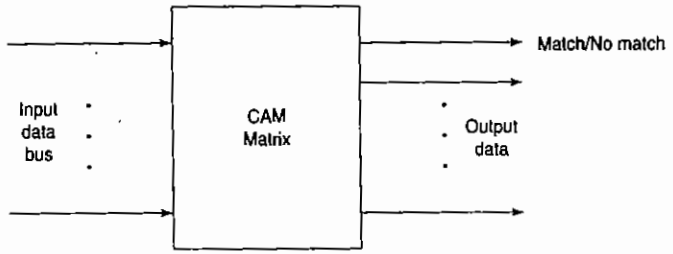


Figure 4-1 CAM architecture.

be autoassociative memory net. On the other hand, if the output vectors are different from the input vectors then the net is said to be heteroassociative memory net.

If there exist vectors, say, $x = (x_1, x_2, \ldots, x_n)^T$ and $x' = (x_1', x_2', \ldots, x_n')^T$, then the hamming distance (HD) is defined as the number of mismatched components of x and x' vectors, i.e.,

$$\text{HD}(x, x') = \begin{cases} \sum_{i=1}^{n} |x_i - x_i'| \text{ if } x_i, x_i' \in [0, 1] \\ \frac{1}{2} \sum_{i=1}^{n} |x_i - x_i'| \text{ if } x_i, x_i' \in [-1, 1] \end{cases}$$

The architecture of an associative net may be either feed-forward or iterative (recurrent). As is already known, in a feed-forward net the information flows from the input units to the output units; on the other hand, in a recurrent neural net, there are connections among the units to form a closed-loop structure. In the forthcoming sections, we will discuss the training algorithms used for pattern association and various types of association nets in detail.

4.2 Training Algorithms for Pattern Association

There are two algorithms developed for training of pattern association nets. These are discussed below.

4.2.1 Hebb Rule

The Hebb rule is widely used for finding the weights of an associative memory neural net. The training vector pairs here are denoted as $s:t$. The flowchart for the training algorithm of pattern association is as shown in Figure 4-2. The weights are updated until there is no weight change. The algorithmic steps followed are given below:

Step 0: Set all the initial weights to zero, i.e.,

$$w_{ij} = 0 \quad (i = 1 \text{ to } n, j = 1 \text{ to } m)$$

Step 1: For each training target input output vector pairs $s:t$, perform Steps 2–4.

Step 2: Activate the input layer units to current training input,

$$x_i = s_i \quad (\text{for } i = 1 \text{ to } n)$$

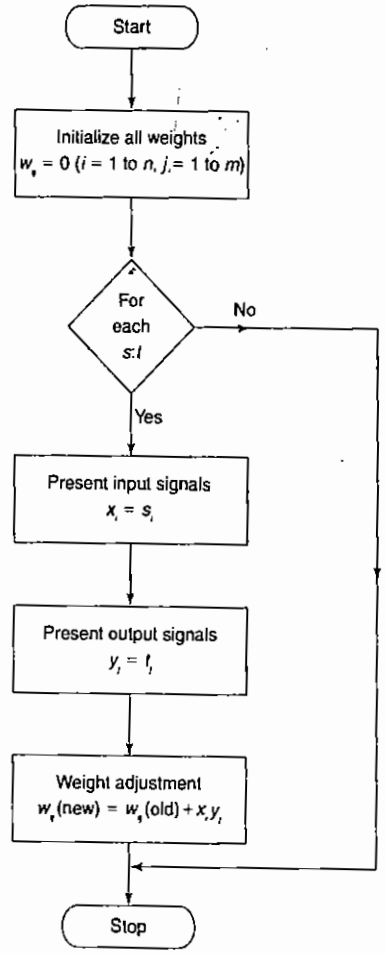


Figure 4-2 Flowchart for Hebb rule.

Step 3: Activate the output layer units to current target output,

$$y_j = t_j \quad (\text{for } j = 1 \text{ to } m)$$

Step 4: Start the weight adjustment

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j \quad (\text{for } i = 1 \text{ to } n, j = 1 \text{ to } m)$$

This algorithm is used for the calculation of the weights of the associative nets. Also, it can be used with patterns that are being represented as either binary or bipolar vectors.

4.2.2 Outer Products Rule

Outer products rule is an alternative method for finding weights of an associative net. This is depicted as follows:

$$\text{Input} \Rightarrow s = (s_1, \ldots, s_i, \ldots, s_n)$$
$$\text{Output} \Rightarrow t = (t_1, \ldots, t_j, \ldots, t_m)$$

The outer product of the two vectors is the product of the matrices $S = s^T$ and $T = t$, i.e., between $[n \times 1]$ matrix and $[1 \times m]$ matrix. The transpose is to be taken for the input matrix given.

The matrix multiplication is done as follows:

$$ST = s^T t$$

$$= \begin{bmatrix} s_1 \\ \cdot \\ \cdot \\ \cdot \\ s_i \\ \cdot \\ \cdot \\ \cdot \\ s_n \end{bmatrix}_{n \times 1} [t_1 \ldots t_j \ldots t_m]_{1 \times m}$$

$$W = \begin{bmatrix} s_1t_1 & \ldots & s_1t_j & \ldots & s_1t_m \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ s_it_1 & \ldots & s_it_j & \ldots & s_it_m \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ \cdot & & & & \cdot \\ s_nt_1 & \ldots & s_nt_j & \ldots & s_nt_m \end{bmatrix}_{n \times m}$$

This weight matrix is same as the weight matrix obtained by Hebb rule to store the pattern association $s:t$. For storing a set of associations, $s(p):t(p)$, $p = 1$ to P, wherein,

$$s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$
$$t(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p))$$

the weight matrix $W = \{w_{ij}\}$ can be given as

$$w_{ij} = \sum_{p=1}^{P} s_i^T(p)\, t_j(p)$$

This can also be rewritten as

$$W = \sum_{p=1}^{P} s^T(p)\, t(p)$$

for finding the weights of the net using Hebbian learning. Similar to the Hebb rule, even the delta rule discussed in Chapter 2 can be used for storing weights of pattern association nets.

4.3 Autoassociative Memory Network

4.3.1 Theory

In the case of an autoassociative neural net, the training input and the target output vectors are the same. The determination of weights of the association net is called storing of vectors. This type of memory net needs suppression of the output noise at the memory output. The vectors that have been stored can be retrieved from distorted (noisy) input if the input is sufficiently similar to it. The net's performance is based on its ability to reproduce a stored pattern from a noisy input. It should be noted, that in the case of autoassociative net, the weights on the diagonal can be set to zero. This can be called as auto associative net with no self-connection. The main reason behind setting the weights to zero is that it improves the net's ability to generalize or increase the biological plausibility of the net. This may be more suited for iterative nets and when delta rule is being used.

4.3.2 Architecture

The architecture of an autoassociative neural net is shown in Figure 4-3. It shows that for an autoassociative net, the training input and target output vectors are the same. The input layer consists of n input units and the output layer also consists of n output units. The input and output layers are connected through weighted interconnections. The input and output vectors are perfectly correlated with each other component by component.

4.3.3 Flowchart for Training Process

The flowchart here is the same as discussed in Section 4.2.1, but it may be noted that the number of input units and output units are the same. The flowchart is shown in Figure 4-4.

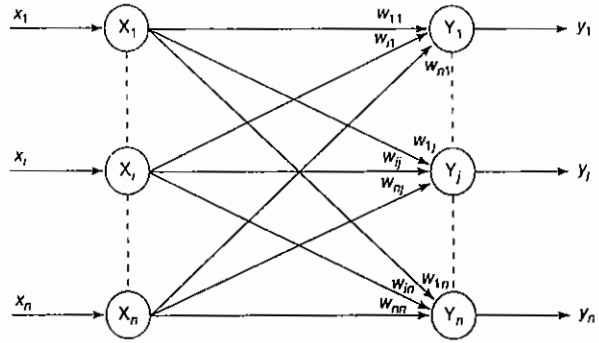


Figure 4-3 Architecture of autoassociative net.

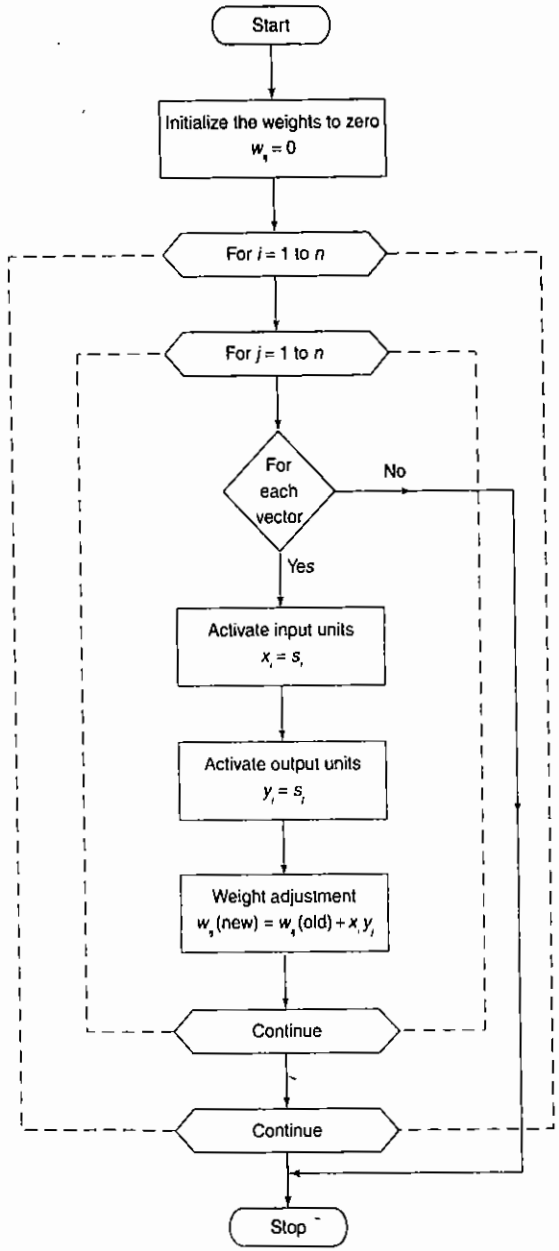


Figure 4-4 Flowchart for training of autoassociative net.

4.3.4 Training Algorithm

The training algorithm discussed here is similar to that discussed in Section 4.2.1 but there are same numbers of output units as that of the input units.

Step 0: Initialize all the weights to zero,

$$w_{ij} = 0 \ (i = 1 \text{ to } n, \ j = 1 \text{ to } n)$$

Step 1: For each of the vector that has to be stored perform Steps 2–4.

Step 2: Activate each of the input unit,

$$x_i = s_i \ (i = 1 \text{ to } n)$$

Step 3: Activate each of the output unit,

$$y_j = s_j \ (j = 1 \text{ to } n)$$

Step 4: Adjust the weights,

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j$$

The weights can also be determined by the formula

$$W = \sum_{p=1}^{P} s^T(p) s(p)$$

4.3.5 Testing Algorithm

An autoassociative memory neural network can be used to determine whether the given input vector is a "known" vector or an "unknown" vector. The net is said to recognize a "known" vector if the net produces a pattern of activation on the output units which is same as one of the vectors stored in it. The testing procedure of an autoassociative neural net is as follows:

Step 0: Set the weights obtained for Hebb's rule or outer products.

Step 1: For each of the testing input vector presented perform Steps 2–4.

Step 2: Set the activations of the input units equal to that of input vector.

Step 3: Calculate the net input to each output unit $j = 1$ to n:

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Step 4: Calculate the output by applying the activation over the net input:

$$y_j = f(y_{inj}) = \begin{cases} +1 & \text{if } y_{inj} > 0 \\ -1 & \text{if } y_{inj} \le 0 \end{cases}$$

This type of network can be used in speech processing, image processing, pattern classification, etc.

4.4 Heteroassociative Memory Network

4.4.1 Theory

In case of a heteroassociative neural net, the training input and the target output vectors are different. The weights are determined in a way that the net can store a set of pattern associations. The association here is a pair of training input target output vector pairs $(s(p), t(p))$, with $p = 1, \ldots, P$. Each vector $s(p)$ has n components and each vector $t(p)$ has m components. The determination of weights is done either by using Hebb rule or delta rule. The net finds an appropriate output vector, which corresponds to an input vector x, that may be either one of the stored patterns or a new pattern.

4.4.2 Architecture

The architecture of a heteroassociative net is shown in Figure 4-5. From the figure, it can be noticed that for a heteroassociative net, the training input and target output vectors are different. The input layer consists of n number of input units and the output layer consists of m number of output units. There exist weighted interconnections between the input and output layers. The input and output layer units are not correlated with each other. The flowchart of the training process and the training algorithm are the same as discussed in Section 4.2.1.

4.4.3 Testing Algorithm

The testing algorithm used for testing the heteroassociative net with either noisy input or with known input is as follows:

Step 0: Initialize the weights from the training algorithm.

Step 1: Perform Steps 2–4 for each input vector presented.

Step 2: Set the activation for input layer units equal to that of the current input vector given, x_i.

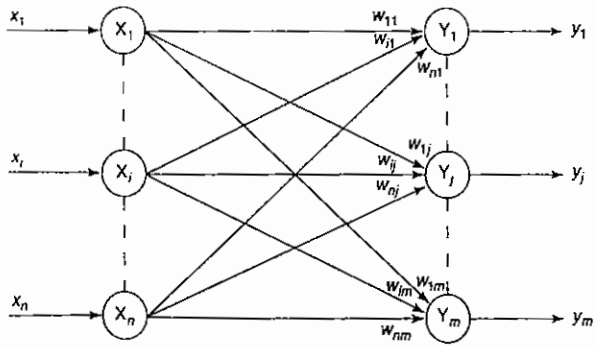


Figure 4-5 Architecture of heteroassociative net.

Step 3: Calculate the net input to the output units:

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij} \quad (j = 1 \text{ to } m)$$

Step 4: Determine the activations of the output units over the calculated net input:

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ 0 & \text{if } y_{inj} = 0 \\ -1 & \text{if } y_{inj} < 0 \end{cases}$$

Thus, the output vector y obtained gives the pattern associated with the input vector x.

Note: Heteroassociative memory is not an iterative memory network. If the responses of the net are binary, then the activation function to be used is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} \geq 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

4.5 Bidirectional Associative Memory (BAM)

4.5.1 Theory

The BAM was developed by Kosko in the year 1988. The BAM network performs forward and backward associative searches for stored stimulus responses. The BAM is a recurrent heteroassociative pattern-matching network that encodes binary or bipolar patterns using Hebbian learning rule. It associates patterns, say from set A to patterns from set B and vice versa is also performed. BAM neural nets can respond to input from either layers (input layer and output layer). There exist two types of BAM, called *discrete* and *continuous* BAM. These two types of BAM are discussed in the following sections.

4.5.2 Architecture

The architecture of BAM network is shown in Figure 4-6. It consists of two layers of neurons which are connected by directed weighted path interconnections. The network dynamics involve two layers of interaction. The BAM network iterates by sending the signals back and forth between the two layers until all the neurons reach equilibrium. The weights associated with the network are bidirectional. Thus, BAM can respond to the inputs in either layer. Figure 4-6 shows a single layer BAM network consisting of n units in X layer and m units in Y layer. The layers can be connected in both directions (bidirectional) with the result the weight matrix sent from the X layer to the Y layer is W and the weight matrix for signals sent from the Y layer to the X layer is W^T. Thus, the weight matrix is calculated in both directions.

4.5.3 Discrete Bidirectional Associative Memory

The structure of discrete BAM is same as shown in Figure 4-6. When the memory neurons are being activated by putting an initial vector at the input of a layer, the network evolves a two-pattern stable state with each pattern at the output of one layer. Thus, the network involves two layers of interaction between each other

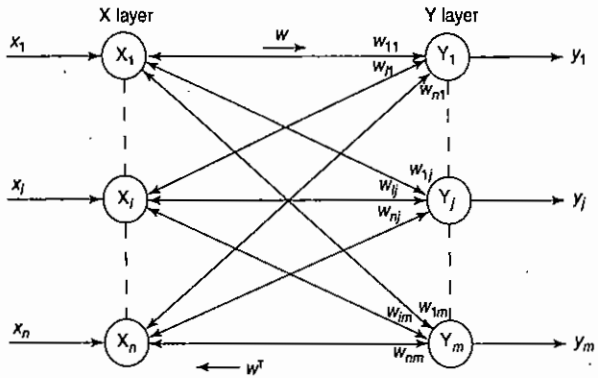


Figure 4-6 Bidirectional associative memory net.

The two bivalent forms of BAM are found to be related with each other, i.e., binary and bipolar. The weights in both the cases are found as the sum of the outer products of the bipolar form of the given training vector pairs. In case of BAM, a definite nonzero threshold is assigned. Thus, the activation function is a step function, with the defined nonzero threshold. When compared, to the binary vectors, bipolar vectors improve the performance of the net to a large extent.

4.5.3.1 Determination of Weights

Let the input vectors be denoted by $s(p)$ and target vectors by $t(p)$, $p = 1, \ldots, P$. Then the weight matrix to store a set of input and target vectors, where

$$s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$$
$$t(p) = (t_1(p), \ldots, t_j(p), \ldots, t_m(p))$$

can be determined by Hebb rule training algorithm discussed in Section 4.2.1. In case of input vectors being binary, the weight matrix $W = \{w_{ij}\}$ is given by

$$w_{ij} = \sum_{p=1}^{P} [2s_i(p) - 1][2t_j(p) - 1]$$

On the other hand, when the input vectors are bipolar, the weight matrix $W = \{w_{ij}\}$ can be defined as

$$w_{ij} = \sum_{p=1}^{P} s_i(p)t_j(p)$$

The weights matrix in both the cases is going to be in bipolar form neither the input vectors are in binary or not. The formulas mentioned above can be directly applied to the determination of weights of a BAM.

4.5.3.2 Activation Functions for BAM

The step activation function with a nonzero threshold is used as the activation function for discrete BAM network. The activation function is based on whether the input target vector pairs used are binary or bipolar. The activation function for the Y layer

1. with binary input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ y_j & \text{if } y_{inj} = 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

2. with bipolar input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > \theta_j \\ y_j & \text{if } y_{inj} = \theta_j \\ -1 & \text{if } y_{inj} < \theta_j \end{cases}$$

The activation function for the X layer

1. with binary input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > 0 \\ x_i & \text{if } x_{ini} = 0 \\ 0 & \text{if } x_{ini} < 0 \end{cases}$$

2. with bipolar input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > \theta_i \\ x_i & \text{if } x_{ini} = \theta_i \\ -1 & \text{if } x_{ini} < \theta_i \end{cases}$$

It may be noted that if the threshold value is equal to that of the net input calculated, then the previous output value calculated is left as the activation of that unit. At a particular time instant, signals are sent only from one layer to the other and not in both the directions.

4.5.3.3 Testing Algorithm for Discrete BAM

The testing algorithm is used to test the noisy patterns entering into the network. Based on the training algorithm, weights are determined, by means of which net input is calculated for the given test pattern and activations is applied over it, to recognize the test patterns. The testing algorithm for the net is as follows:

Step 0: Initialize the weights to store p vectors. Also initialize all the activations to zero.

Step 1: Perform Steps 2–6 for each testing input.

Step 2: Set the activations of X layer to current input pattern, i.e., presenting the input pattern x to X layer and similarly presenting the input pattern y to Y layer. Even though, it is bidirectional memory, at one time step, signals can be sent from only one layer. So, either of the input patterns may be the zero vector.

Step 3: Perform Steps 4–6 when the activations are not converged.

Step 4: Update the activations of units in Y layer. Calculate the net input,

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Applying the activations (as in Section 4.5.3.2), we obtain

$$y_j = f(y_{inj})$$

Send this signal to the X layer.

Step 5: Update the activations of units in X layer. Calculate the net input,

$$x_{ini} = \sum_{j=1}^{m} y_j w_{ij}$$

Apply the activations over the net input,

$$x_i = f(x_{ini})$$

Send this signal to the Y layer.

Step 6: Test for convergence of the net. The convergence occurs if the activation vectors x and y reach equilibrium. If this occurs then stop, otherwise, continue.

4.5.4 Continuous BAM

A continuous BAM transforms the input smoothly and continuously in the range 0–1 using logistic sigmoid functions as the activation functions for all units. The logistic sigmoidal function may be either binary sigmoidal function or bipolar sigmoidal function. When a bipolar sigmoidal function with a high gain is chosen, then the continuous BAM might converge to a state of vectors which will approach vertices of the cube. When that state of the vector approaches it acts like a discrete BAM.

If the input vectors are binary, $(s(p), t(p))$, $p = 1$ to P, the weights are determined using the formula

$$w_{ij} = \sum_{p=1}^{P} [2s_i(p) - 1][2t_j(p) - 1]$$

i.e., even though the input vectors are binary, the weight matrix is bipolar. The activation function used here is the logistic sigmoidal function. If it is binary logistic function, then the activation function is

$$f(y_{inj}) = \frac{1}{1 + e^{-y_{inj}}}$$

If the activation function used is a bipolar logistic function, then the function is defined as

$$f(y_{inj}) = \frac{2}{1 + e^{-y_{inj}}} - 1 = \frac{1 - e^{-y_{inj}}}{1 + e^{-y_{inj}}}$$

These activation functions are applied over the net input to calculate the output. The net input can be calculated with a bias included, i.e.,

$$y_{inj} = b_j + \sum_i x_i w_{ij}$$

and all these formulas apply for the units in X layer also.

4.5.5 Analysis of Hamming Distance, Energy Function and Storage Capacity

The hamming distance is defined as the number of mismatched components of two given bipolar or binary vectors. It can also be defined as the number of different bits in two binary or bipolar vectors X and X'. It is denoted as $H[X, X']$. The average hamming distance between the vectors is $(1/n)H[X, X']$, where "n" is the number of components in each vector. Consider the vectors,

$$X = [1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0] \text{ and } X' = [1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1]$$

The hamming distance between these two given vectors is equal to 5. The average hamming distance between the corresponding vectors is 5/7.

The stability analysis of a BAM is based on the definition of Lyapunov function (energy function). Consider that there are p vector association pairs to be stored in a BAM:

$$\{(x^1, y^1), (x^2, y^2), \ldots, (x^p, y^p)\}$$

where $x^k = (x_1^k, x_2^k, \ldots, x_m^k)^T$ and $y^k = (y_1^k, y_2^k, \ldots, y_n^k)^T$ are either binary or bipolar vectors. A Lyapunov function must be always bounded and decreasing. A BAM can be said to be bidirectionally stable if the state converges to a stable point, i.e., $y^k \to y^{k+1} \to y^{k+2}$ and $y^{k+2} = y^k$. This gives the minimum of the energy function. The energy function or Lynapunov function of a BAM is defined as

$$E_f(x, y) = \frac{-1}{2} x^T W^T y - \frac{1}{2} y^T W x = -y^T W x$$

The change in energy due to the single bit changes in both vectors y and x given as Δy_i and Δx_j can be found as

$$\Delta E_f(y_i) = \nabla_y E \Delta y_i = -Wx\Delta y_i = -\left(\sum_{j=1}^{m} x_j w_{ij}\right) \times \Delta y_i, \quad i = 1 \text{ to } n$$

$$\Delta E_f(x_j) = \nabla_x E \Delta x_j = -W^T y \Delta x_j = -\left(\sum_{i=1}^{n} y_i w_{ij}\right) \times \Delta x_j, \quad j = 1 \text{ to } m$$

where Δy_i and Δx_j are given as

$$\Delta x_j = \begin{cases} 2 & \text{if } \sum_{i=1}^{n} y_i w_{ji} > 0 \\ 0 & \text{if } \sum_{i=1}^{n} y_i w_{ji} = 0 \\ -2 & \text{if } \sum_{i=1}^{n} y_i w_{ji} < 0 \end{cases} \quad \text{and} \quad \Delta y_i = \begin{cases} 2 & \text{if } \sum_{j=1}^{m} x_j w_{ij} > 0 \\ 0 & \text{if } \sum_{j=1}^{m} x_j w_{ij} = 0 \\ -2 & \text{if } \sum_{j=1}^{m} x_j w_{ij} < 0 \end{cases}$$

Here the energy function is bounded below by

$$E_f(x,y) \geq -\sum_{i=1}^{n}\sum_{j=1}^{m}|w_{ij}|$$

so the discrete BAM will converge to a stable state.

The memory capaciry or the storage capacity of BAM may be given as

$$\min(m, n)$$

where "n" is the number of units in X layer and "m" is the number of units in Y layer. Also a more conservative capacity is estimated as follows:

$$\sqrt{\min(m, n)}$$

4.6 Hopfield Networks

John J. Hopfield developed a model in the year 1982 conforming to the asynchronous nature of biological neurons. The networks proposed by Hopfield are known as Hopfield networks and it is his work that promoted construction of the first analog VLSI neural chip. This network has found many useful applications in associative memory and various optimization problems. In this section, two types of network are discussed: *discrete* and *continuous Hopfield networks.*

4.6.1 Discrete Hopfield Network

The Hopfield network is an autoassociative fully interconnected single-layer feedback network. It is also a symmetrically weighted network. When this is operated in discrete line fashion it is called as *discrete Hopfield* network and its architecture as a single-layer feedback network can be called as *recurrent*. The network takes two-valued inputs: binary (0, 1) or bipolar (+1, −1); the use of bipolar inputs makes the analysis easier. The network has symmetrical weights with no self-connections, i.e.,

$$w_{ij} = w_{ji}; \quad w_{ii} = 0$$

The key points to be noted in Hopfield net are: only one unit updates its activation at a time; also each unit is found to continuously receive an external signal along with the signals it receives from the other units in the net. When a single-layer recurrent network is performing a sequential updating process, an input pattern is first applied to the network and the network's output is found to be initialized accordingly. Afterwards, the initializing pattern is removed, and the output that is initialized becomes the new updated input through the feedback connections. The first updated input forces the first updated output, which in turn acts as the second updated input through the feedback interconnections and results in second updated output. This transition process continues until no new, updated responses are produced and the network reaches its equilibrium.

The asynchronous updation of the units allows a function, called as energy functions or Lyapunov function, for the net. The existence of this function enables us to prove that the net will converge to a stable set of activations. The usefulness of content addressable memory is realized by the discrete Hopfield net.

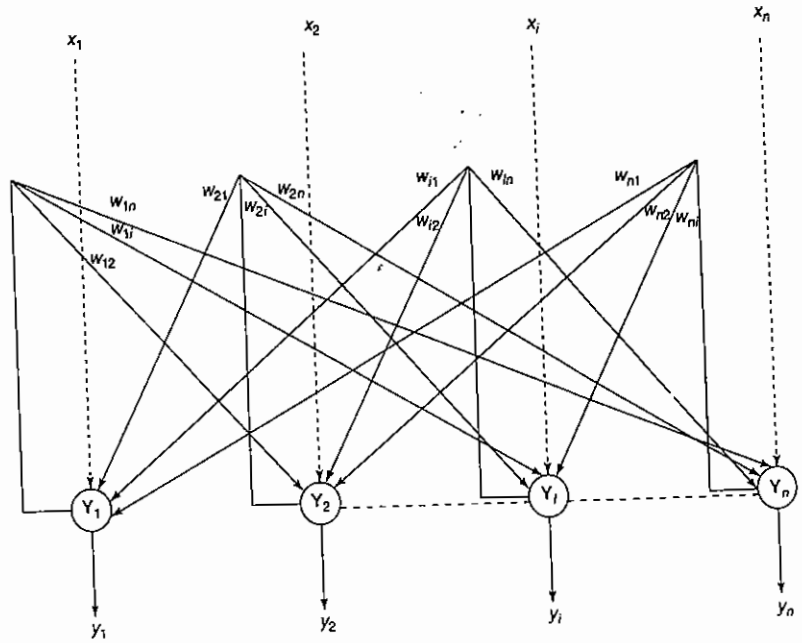


Figure 4-7 Architecture of discrete Hopfield net.

4.6.1.1 *Architecture of Discrete Hopfield Net*

The architecture of discrete Hopfield net is shown in Figure 4-7. The Hopfield's model consists of processing elements with two outputs, one inverting and the other non-inverting. The outputs from each processing element are fed back to the input of other processing elements but not to itself. The connections are found to be resistive and the connection strength over it is represented as w_{ij}. Here, as such there are no negative resistors, hence excitatory connections use positive outputs and inhibitory connections use inverted outputs. Connections are excitatory if the output of a processing element is found to be same as the input, and they are inhibitory if the inputs differ from the output of the processing element. A connection between the processing elements i and j is found to be associated with a connection strength w_{ij}. This weight is positive if units i and j are both on. On the other hand, if the connection strength is negative, it represents the situation of unit i being on and j being off. Also, the weights are symmetric, i.e., the weights w_{ij} are same as w_{ji}.

4.6.1.2 *Training Algorithm of Discrete Hopfield Net*

There exist several versions of the discrete Hopfield net. It should be noted that Hopfield's first description used binary input vectors and only later on bipolar input vectors used.

For storing a set of binary patterns $s(p)$, $p = 1$ to P, where $s(p) = (s_1(p), \ldots, s_i(p), \ldots, s_n(p))$, the weight matrix W is given as

$$w_{ij} = \sum_{p=1}^{P}[2s_i(p) - 1][2s_j(p) - 1], \quad \text{for } i \neq j$$

For storing a set of bipolar input patterns, $s(p)$ (as defined above), the weight matrix W is given as

$$w_{ij} = \sum_{p=1}^{P} s_i(p) s_j(p), \quad \text{for } i \neq j$$

and the weights here have no self-connection, i.e., $w_{ij} = 0$.

4.6.1.3 Testing Algorithm of Discrete Hopfield Net

In the case of testing, the update rule is formed and the initial weights are those obtained from the training algorithm. The testing algorithm for the discrete Hopfield net is as follows:

Step 0: Initialize the weights to store patterns, i.e., weights obtained from training algorithm using Hebb rule.

Step 1: When the activations of the net are not converged, then perform Steps 2–8.

Step 2: Perform Steps 3–7 for each input vector X.

Step 3: Make the initial activations of the net equal to the external input vector X:

$$y_i = x_i \ (i = 1 \text{ to } n)$$

Step 4: Perform Steps 5–7 for each unit Y_i. (Here, the units are updated in random order.)

Step 5: Calculate the net input of the network:

$$y_{in_i} = x_i + \sum_j y_j w_{ji}$$

Step 6: Apply the activations over the net input to calculate the output:

$$y_i = \begin{cases} 1 & \text{if } y_{in_i} > \theta_i \\ y_i & \text{if } y_{in_i} = \theta_i \\ 0 & \text{if } y_{in_i} < \theta_i \end{cases}$$

where θ_i is the threshold and is normally taken as zero.

Step 7: Now feed back (transmit) the obtained output y_i to all other units. Thus, the activation vectors are updated.

Step 8: Finally, test the network for convergence.

The updation here is carried out at random, but it should be noted that each unit may be updated at the same average rate. The asynchronous fashion of updation is carried out here. This means that for a given time only a single neural unit is allowed to update its output. The next update can be carried out on a randomly chosen node which uses the already updated output. It can also be said that under asynchronous operation of the network, each output node unit is updated separately by taking into account the most recent values that have already been updated. This type of updation is referred to as an *asynchronous stochastic recursion* of the discrete Hopfield network. By performing the analysis of the Lyapunov function, i.e., the energy function for the Hopfield net, it can be shown that the main feature for the convergence of this net is the asynchronous updation of weights and the weights with no self-connection, i.e., the zeros exist on the diagonals of the weight matrix.

A Hopfield network with binary input vectors is used to determine whether an input vector is a "known" vector or an "unknown" vector. The net has the capacity to recognize a known vector by producing a pattern of activations on the units of the net that is same as the vector stored in the net. For example, if the input vector is an unknown vector, the activation vectors resulted during iteration will converge to an activation vector which is not one of the stored patterns; such a pattern is called as *spurious stable state.*

4.6.1.4 Analysis of Energy Function and Storage Capacity on Discrete Hopfield Net

An energy function generally is defined as a function that is bounded and is a nonincreasing function of the state of the system. The energy function, also called as Lyapunov function, determines the stability property of a discrete Hopfield network. The state of a system for a neural network is the vector of activations of the units. Hence, if it is possible to find an energy function for an iterative neural net, the net will converge to a stable set of activations. An energy function E_f of a discrete Hopfield network is characterized as

$$E_f = -\frac{1}{2} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} y_i y_j w_{ij} - \sum_{i=1}^{n} x_i y_i + \sum_{i=1}^{n} \theta_i y_i$$

If the network is stable, then the above energy function decreases whenever the state of any node changes. Assuming that node i has changed its state from $y_i^{(k)}$ to $y_i^{(k+1)}$, i.e., the output has changed from $+1$ to -1 or from -1 to $+1$, the energy change ΔE_f is then given by

$$\begin{aligned} \Delta E_f &= E_f\left(y_j^{(k+1)}\right) - E_f\left(y_i^{(k)}\right) \\ &= -\left(\sum_{\substack{j=1 \\ j \neq i}}^{n} y_j^{(k)} w_{ij} + x_i - \theta_i\right)\left(y_i^{(k+1)} - y_i^{(k)}\right) \\ &= -(\text{net}_i)\, \Delta y_i \end{aligned}$$

where $\Delta y_i = y_i^{(k+1)} - y^{(k)}$. The change in energy is dependent on the fact that only one unit can update its activation at a time. The change in energy equation ΔE_f exploits the fact that $y_j^{(k+1)} = y_j^{(k)}$ for $j \neq i$ and $w_{ij} = w_{ji}$ and $w_{ii} = 0$ (symmetric weight property).

There exist two cases in which a change Δy_i will occur in the activation of neuron Y_i. If y_i is positive, then it will change to zero if

$$\left[x_i + \sum_{j=1}^{n} y_j w_{ji}\right] < \theta_i$$

This results in a negative change for y_i and $\Delta E_f < 0$. On the other hand, if y_i is zero, then it will change to positive if

$$\left[x_i + \sum_{j=1}^{n} y_j w_{ji}\right] > \theta_i$$

This results in a positive change for y_i and $\Delta E_f < 0$. Hence Δy_i is positive only if net input is positive and Δy_i is negative only if net input is negative. Therefore, the energy cannot increase in any manner. As a result,

because the energy is bounded, the net must reach a stable state equilibrium, such that the energy does not change with further iteration. From this it can be concluded that the energy change depends mainly on the change in activation of one unit and on the symmetry of weight matrix with zeros existing on the diagonal.

A Hopfield network always converges to a stable state in a finite number of node-updating steps, where every stable state is found to be at the local minima of the energy function E_f. Also, the proving process uses the well-known Lyapunov stability theorem, which is generally used to prove the stability of dynamic system defined with arbitrarily many interlocked differential equations. A positive-definite (energy) function $E_f(y)$ can be found such that:

1. $E_f(y)$ is continuous with respect to all the components y_i for $i = 1$ to n;
2. $d E_f[y(t)]/dt < 0$, which indicates that the energy function is decreasing with time and hence the origin of the state space is asymptotically stable.

Hence, a positive-definite (energy) function $E_f(y)$ satisfying the above requirements can be Lyapunov function for any given system; this function is not unique. If, at least one such function can be found for a system, then the system is asymptotically stable. According to the Lyapunov theorem, the energy function that is associated with a Hopfield network is a Lyapunov function and thus the discrete Hopfield network is asymptotically stable.

The storage capacity is another important factor. It can be found that the number of binary patterns that can be stored and recalled in a network with a reasonable accuracy is given approximately as

$$\text{Storage capacity } C \simeq 0.15n$$

where n is the number of neurons in the net. It can also be given as

$$C \cong \frac{n}{2 \log_2 n}$$

4.6.2 Continuous Hopfield Network

A discrete Hopfield net can be modified to a continuous model, in which time is assumed to be a continuous variable, and can be used for associative memory problems or optimization problems like traveling salesman problem. The nodes of this network have a continuous, graded output rather than a two-state binary output. Thus, the energy of the network decreases continuously with time. The continuous Hopfield networks can be realized as an electronic circuit, which uses non-linear amplifiers and resistors. This helps building the Hopfield network using analog VLSI technology.

4.6.2.1 Hardware Model of Continuous Hopfield Network

The continuous network build up of electrical components is shown in Figure 4-8.

The model consists of n amplifiers, mapping its input voltage u_i into an output voltage y_i over an activation function $a(u_i)$. The activation function used can be a sigmoid function, say,

$$a(\lambda u_i) = \frac{1}{1 + e^{-\lambda u_i}}$$

where λ is called the gain parameter.

The continuous model becomes a discrete one when $\lambda \to \alpha$. Each of the amplifiers consists of an input capacitance c_i and an input conductance g_{ri}. The external signals entering into the circuit are x_i. The external

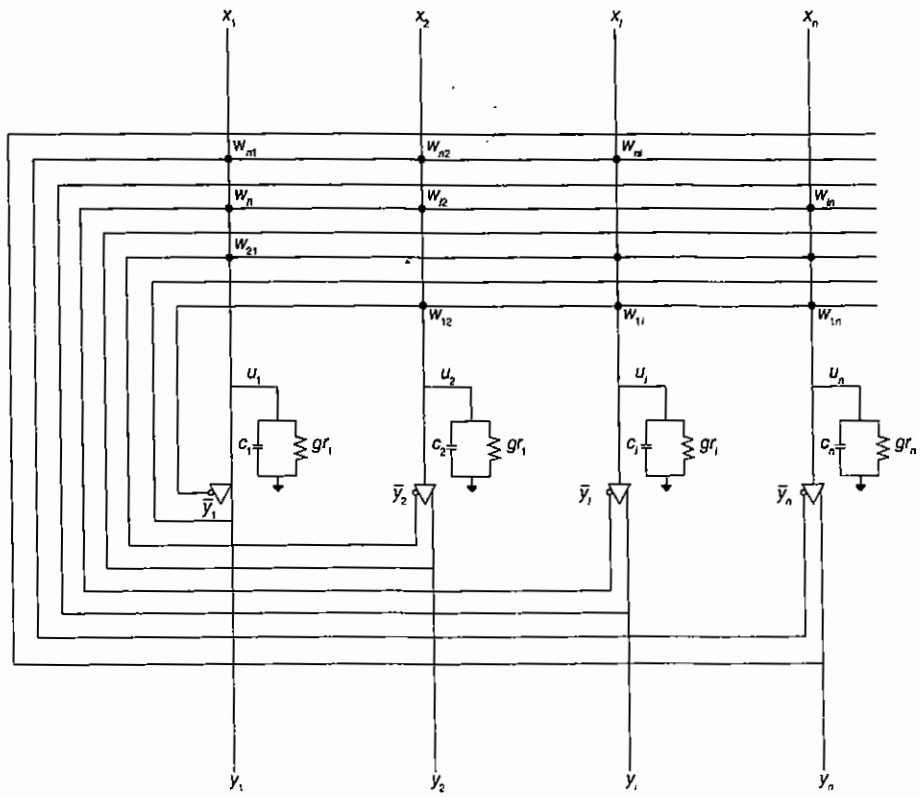


Figure 4-8 Model of Hopfield network using electrical components.

signals supply constant current to each amplifier for an actual circuit. The output of the jth node is connected to the input of the ith node through conductance w_{ij}. Since all real resistor values are positive, the inverted node outputs $\overline{y}_i$ are used to simulate the inhibitory signals. The connection is made with the signal from the noninverted output if the output of a particular node excites some other node. If the connection is inhibitory, then the connection is made with the signal from the inverted output. Here also, the important symmetric weight requirement for Hopfield network is imposed, i.e., $w_{ij} = w_{ji}$ and $w_{ii} = 0$.

The rule of each node in a continuous Hopfield network can be derived as shown in Figure 4-9. Consider the input of a single node as in Figure 4-9. Applying Kirchoff's current law (KCL), which states that the total current entering a junction is equal to that leaving the same function, we get

$$C_i \frac{du_i}{dt} = \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij}(y_j - u_i) - g_{ri} u_i + x_i = \sum_{\substack{j=1 \\ j \neq i}}^{n} w_{ij} y_j - G_i u_i + x_i$$

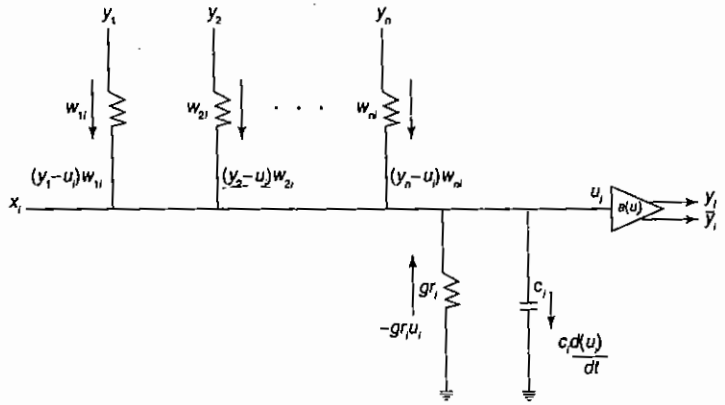


Figure 4-9 Input of a single node of continuous Hopfield network.

where

$$G_i = \sum_{\substack{j=1 \\ j\neq i}}^{n} w_{ij} + g_{ri}$$

The equation obtained using KCL describes the time evolution of the system completely. If each single node is given an initial value say, $u_i(0)$, then the value $u_i(t)$ and thus the amplifier output, $y_i(t) = a(u_i(t))$ at time t, can be known by solving the differential equation obtained using KCL.

4.6.2.2 Analysis of Energy Function of Continuous Hopfield Network

For evaluating the stability property of continuous Hopfield network, a continuous energy function is defined such that the evolution of the system is in the negative gradient of the energy function and finally converges to one of the table minima in the state space. The corresponding Lyapunov energy function for the model shown in Figure 4-8 is

$$E_f = -\frac{1}{2}\sum_{i=1}^{n}\sum_{\substack{j=1 \\ j\neq i}}^{n} w_{ij}y_iy_j - \sum_{i=1}^{n} x_iy_i + \frac{1}{\lambda}\sum_{i=1}^{n} G_i \int_0^{y_i} a^{-1}(y)dy$$

where $a^{-1}(y) = \lambda u$ is the inverse of the function $y = a(\lambda u)$. The inverse of the function $a^{-1}(y)$ is shown in Figure 4-10(A) and the integral of it in Figure 4-10(B).

To prove that E_f obtained is the Lyapunov function for the network, its time derivative is taken with weights w_{ij} symmetric:

$$\frac{dE_f}{dt} = \sum_{i=1}^{n}\frac{dE}{dy_i}\frac{dy_i}{dt} = \sum_{i=1}^{n}\left(-\sum_{\substack{j=1 \\ j\neq i}}^{n} y_jw_{ij} + G_iu_i - x_i\right)\frac{dy_i}{dt} = -\sum_{i=1}^{n} c_i\frac{dy_i}{dt}\frac{du_i}{dt}$$

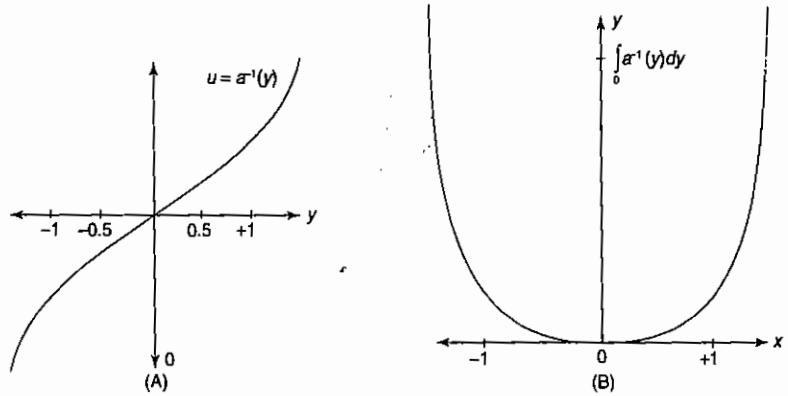


Figure 4-10 (A) Inverse and (B) integral of nonlinear activation function $a^{-1}(y)$.

As

$$u_i = \left(\frac{1}{\lambda}\right)a^{-1}(y_i)$$

we get

$$\frac{du_i}{dt} = \frac{1}{\lambda}\frac{da^{-1}(y_i)}{dy_i}\frac{dy_i}{dt} = \frac{1}{\lambda}a^{-1'}(y_i)\frac{dy_i}{dt}$$

where the derivative of $a^{-1}(y)$ is $a^{-1'}(y)$. So, the derivative of energy function equation becomes

$$\frac{dE_f}{dt} = -\sum_{i=1}^{n}\frac{1}{\lambda}c_ia^{-1'}(y_i)\left(\frac{dy_i}{dt}\right)^2$$

From Figure 4-10(A), we know that $a^{-1}(y_i)$ is a monotonically increasing function of y_i and hence its derivative is positive, all over. This shows that dE_f/dt is negative, and thus the energy function E_f must decrease as the system evolves. Therefore, if E_f is bounded, the system will eventually reach a stable state, where

$$\frac{dE_f}{dt} = \frac{dy_i}{dt} = 0$$

When the values of threshold are zero, the continuous energy function becomes equal to the discrete energy function, except for the term,

$$\frac{1}{\lambda}\sum_{i=1}^{n} G_i \int_0^{y_i} a^{-1}(y)dy$$

From Figure 4-10(B), the integral of $a^{-1}(y)$ is zero when y_i is zero and positive for all other values of y_i. The integral becomes very large as y approaches +1 or −1. Hence, the energy function E_f is bounded from below and is a Lyapunov function. The continuous Hopfield nets are best suited for the constrained optimization problems.

4.7 Iterative Autoassociative Memory Networks

There exists a situation where the net does not respond to the input signal immediately with a stored target pattern but the response may be more like the stored pattern, which suggests using the first response as input to the net again. The iterative autoassociative net should be able to recover an original stored vector when presented with a test vector close to it. These types of networks can also be called as *recurrent autoassociative networks* and Hopfield networks discussed in Section 4.6 come under this category.

4.7.1 Linear Autoassociative Memory (LAM)

In 1977, James Anderson focused on the development of the LAM. This was based on Hebbian rule, which states that connections between neuron like elements are strengthened every time when they are activated. Linear algebra is used to analyze the performance of the net.

Consider an $m \times m$ non singular symmetric matrix having "m" mutually orthogonal eigen vectors. The eigen vectors satisfy the property of orthogonality. A recurrent linear autoassociator network is trained using a set of P orthogonal unit vector $u_1, \ldots, u_P$, where the number of times each vector going to be presented is not the same.

The weight matrix can be determined using Hebb learning rule, but this allows the repetition of some of the stored vectors. Each of these stored vectors is an eigen vector of the weight matrix. Here, eigen values represent the number of times the vector was presented.

When the input vector X is presented, the output response of the net is XW, where W is the weight matrix. From the concepts of linear algebra, we know that we obtain the largest value of $\|XW\|$ when X is the eigen vector for the largest eigen value; the next largest value of $\|XW\|$ occurs when X is the eigen vector for the next largest eigen value, and so on. Thus, a recurrent linear autoassociator produces its response as the stored vector for which the input vector is most similar. This may perhaps take several iterations. The linear combination of vectors may be used to represent an input pattern. When an input vector is presented, the response of the net is the linear combination of its corresponding eigen values. The eigen vector with largest value in this linear expansion is the one which is most similar to that of the input vectors. Although, the net increases its response corresponding to components of the input pattern over which it is trained most extensively, the overall output response of the system may grow without bound.

The main conditions of linearity between the associative memories is that the set of input vector pairs and output vector pairs (since, autoassociative, both are same) should be mutually orthogonal with each other, i.e., if "A_p" is the input pattern pair, for $p = 1$ to P, then

$$A_i A_j^{\mathrm{T}} = 0, \qquad \text{for all } i \neq j$$

Also if all the vectors A_p are normalized to unit length, i.e.,

$$\sum_{i=1}^{n} (a_i)_p^2 = 1, \qquad \text{for all } p = 1 \text{ to } P$$

then the output $Y_j = A_p$, i.e., the desired output has been recalled.

4.7.2 Brain-in-the-Box Network

An extension to the linear associator is the brain-in-the-box model. This model was described by Anderson, 1972, as follows: an activity pattern inside the box receives positive feedback on certain components, which has the effect of forcing it outward. When its element start to limit (when it hits the wall of the box), it moves to corner of the box where it remains as such. The box resides in the state-space (each neuron occupies one axis) of the network and represents the saturation limits for each state. Each component here is being restricted between -1 and $+1$. The updation of activations of the units in brain-in-the-box model is done simultaneously.

The brain-in-the-box model consists of n units, each being connected to every other unit. Also, there is a trained weight on the self-connection, i.e., the diagonal elements are set to zero. There also exists a self-connection with weight 1. The algorithm for brain-in-the-box model is given in Section 4.7.2.1.

4.7.2.1 Training Algorithm for Brain-in-the-Box Model

Step 0: Initialize the weights to very small random values. Initialize the learning rates α and β.

Step 1: Perform Steps 2–6 for each training input vector.

Step 2: The initial activations of the net are made equal to the external input vector X:

$$y_i = x_i$$

Step 3: Perform Steps 4 and 5 when the activations continue to change.

Step 4: Calculate the net input:

$$y_{inj} = y_i + \alpha \sum_{j=1}^{n} y_j w_{ji}$$

Step 5: Calculate the output of each unit by applying its activations:

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 1 \\ y_{inj} & \text{if } -1 \leq y_{inj} \leq 1 \\ -1 & \text{if } y_{inj} < -1 \end{cases}$$

The vertex of the box will be a stable state for the activation vector.

Step 6: Update the weights:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \beta\, y_i y_j$$

4.7.3 Autoassociator with Threshold Unit

If a threshold unit is set, then a threshold function can be used as the activation function for an iterative autoassociator net. The testing algorithm of autoassociator with specified threshold for bipolar vectors and activations with symmetric weights and no self-connections, i.e., $w_{ij} = w_{ji}$ and $w_{ii} = 0$ is given in the following section.

4.7.3.1 Testing Algorithm

Step 0: The weights are initialized from the training algorithm to store patterns (use Hebbian learning).

Step 1: Perform Steps 2-5 for each testing input vector.

Step 2: Set the activations of X.

Step 3: Perform Steps 4 and 5 when the stopping condition is false.

Step 4: Update the activations of all units:

$$x_i = \begin{cases} 1 & \text{if } \sum_{j=1}^{n} x_j w_{ij} > \theta_i \\ x_i & \text{if } \sum_{j=1}^{n} x_j w_{ij} = \theta_i \\ -1 & \text{if } \sum_{j=1}^{n} x_j w_{ij} > \theta_i \end{cases}$$

The threshold θ_i may be taken as zero.

Step 5: Test for the stopping condition.

The network performs iteration until the correct vector X matches a stored vector or the testing input matches a previous vector or the maximum number of iterations allowed is reached.

4.8 Temporal Associative Memory Network

The associative memories discussed so far evolve a stable state and stay there. All are acting as content addressable memories for a set of static patterns. But there is also a possibility of storing the sequences of patterns in the form of dynamic transitions. These types of patterns are called as *temporal patterns* and an associative memory with this capability is called as a *temporal associative memory*. In this section, we shall learn how the BAM act as temporal associative memories. Assume all temporal patterns as bipolar or binary vectors given by an ordered set S with p vectors:

$$S = \{s_1, s_2, \ldots, s_i, \ldots, s_p\} \quad (p = 1 \text{ to } P)$$

where column vectors are n-dimensional. The neural network can memorize the sequence S in its dynamic state transitions such that the recalled sequence is $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i \rightarrow \ldots \rightarrow s_p \rightarrow s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_i \rightarrow \ldots$ or in reverse order.

A BAM can be used to generate the sequence $S = \{s_1, s_2, \ldots, s_i, \ldots, s_p\}$. The pair of consecutive vectors s_k and s_{k+1} are taken as heteroassociative. From this point of view, s_1 is associated with s_2, s_2 is associated with $s_3, \ldots$ and s_p is again associated with s_1. The weight matrix is then given as

$$W = \sum_{k=1}^{p} (s_{k+1})(s_k)^T$$

A BAM for temporal patterns can be modified so that both layers X and Y are described by identical weight matrices W. Hence, the recalling is based on

$$x = f(W_y); \quad y = f(W_x)$$

where $f(\cdot)$ is the activation function of the network. Also a reverse order recall can be implemented using the transposed weight matrices in both layers X and Y. In case of temporal BAM, layers X and Y update nonsimultaneously and in an alternate circular fashion.

The energy function for a temporal BAM can be defined as

$$E_f = -\sum_{k=1}^{p} s_{k+1} W s_k$$

The energy function E_f decreases during the temporal sequence retrieval $s_1 \rightarrow s_2 \rightarrow \ldots \rightarrow s_p$. The energy is found to increase stepwise at the transition $s_p \rightarrow s_1$ and then it continues to decrease in the following cycle of $(p-1)$ retrievals. The storage capacity of the BAM is estimated using $p \leq \min(m, n)$. Hence, the maximum length sequence is bounded by $p < n$, where n is number of components in input vector and m is number of components in output vector.

4.9 Summary

Pattern association is carried out efficiently by associative memory networks. The two main algorithms used for training a pattern association network are the Hebb rule and the outer products rule. The basic architecture, flowchart for training process and the training algorithm are discussed in detail for autoassociative net, heteroassociative memory net, BAM, Hopfield net and iterative nets. Also, in all cases suitable testing algorithm is included. The variations in BAM, discrete BAM and continuous BAM, are discussed in this chapter. The analysis of hamming distance, energy function and storage capacity is done for few networks such as BAM, discrete Hopfield network and continuous Hopfield network. In case of iterative autoassociative memory network, the linear autoassociative memory, brain-in-the-box model and an autoassociator with a threshold unit are discussed. Also temporal associative memory network is discussed briefly.

4.10 Solved Problems

1. Train a heteroassociative memory network using Hebb rule to store input row vector $s = (s_1, s_2, s_3, s_4)$ to the output row vector $t = (t_1, t_2)$. The vector pairs are given in Table 1.

Table 1

Input targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	1	0	1	0
2nd	1	0	0	1	1	0
3rd	1	1	0	0	0	1
4th	0	0	1	1	0	1

Solution: The network for the given problem is as shown in Figure 1. The training algorithm based on Hebb rule is used to determine the weights.

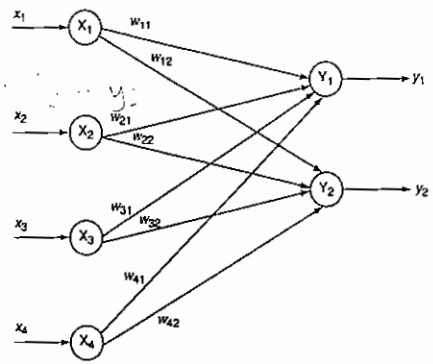


Figure 1 Neural net.

For 1st input vector:

Step 0: Initialize the weights, the initial weights are taken as zero.

Step 1: For first pair (1, 0, 1, 0):(1, 0)

Step 2: Set the activations of input units:

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 1, \quad x_4 = 0$$

Step 3: Set the activations of output unit:

$$y_1 = 1, \quad y_2 = 0$$

Step 4: Update the weights,

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + x_i y_j$$
$$w_{11}(\text{new}) = w_{11}(\text{old}) + x_1 y_1 = 0 + 1 \times 1 = 1$$
$$w_{21}(\text{new}) = w_{21}(\text{old}) + x_2 y_1 = 0 + 0 \times 1 = 0$$
$$w_{31}(\text{new}) = w_{31}(\text{old}) + x_3 y_1 = 0 + 1 \times 1 = 1$$
$$w_{41}(\text{new}) = w_{41}(\text{old}) + x_4 y_1 = 0 + 0 \times 1 = 0$$
$$w_{12}(\text{new}) = w_{12}(\text{old}) + x_1 y_2 = 0 + 1 \times 0 = 0$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + x_2 y_2 = 0 + 0 \times 0 = 0$$
$$w_{32}(\text{new}) = w_{32}(\text{old}) + x_3 y_2 = 0 + 1 \times 0 = 0$$
$$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 0 \times 0 = 0$$

For 2nd input vector:

The input–output vector pair is (1, 0, 0, 1):(1, 0)

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 0, \quad x_4 = 1,$$
$$y_1 = 1, \quad y_2 = 0$$

The final weights obtained for the input vector pair is used as initial weight here:

$$w_{11}(\text{new}) = w_{11}(\text{old}) + x_1 y_1 = 1 + 1 \times 1 = 2$$
$$w_{41}(\text{new}) = w_{41}(\text{old}) + x_4 y_1 = 0 + 1 \times 1 = 1$$

Since $x_2 = x_3 = y_2 = 0$, the other weights remains the same.

The final weights after second input vector is presented are

$$w_{11} = 2, \; w_{21} = 0, \; w_{31} = 1, \; w_{41} = 1$$
$$w_{12} = 0, \; w_{22} = 0, \; w_{32} = 0, \; w_{42} = 0$$

For 3rd input vector:

The input–output vector pair is (1, 1, 0, 0):(0, 1)

$$x_1 = 1, \; x_2 = 1, \; x_3 = 0, \; x_4 = 0, \; y_1 = 0, \; y_2 = 1$$

Training, using Hebb rule, evolves the final weights as follows:

Since $y_1 = 0$, the weights of y_1 are going to the same.

Computing the weights of y_2 unit, we obtain

$$w_{12}(\text{new}) = w_{12}(\text{old}) + x_1 y_2 = 0 + 1 \times 1 = 1$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + x_2 y_2 = 0 + 1 \times 1 = 1$$
$$w_{32}(\text{new}) = w_{32}(\text{old}) + x_3 y_2 = 0 + 0 \times 1 = 0$$
$$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 0 \times 1 = 0$$

The final weights after presenting third input vector are

$$w_{11} = 2, \; w_{21} = 0, \; w_{31} = 1, \; w_{41} = 1$$
$$w_{12} = 1, \; w_{22} = 1, \; w_{32} = 0, \; w_{42} = 0$$

For 4th input vector:

The input–output vector pair is (0, 0, 1, 1):(0, 1)

$$x_1 = 0, \; x_2 = 0, \; x_3 = 1, \; x_4 = 1, \; y_1 = 0, \; y_2 = 1$$

The weights are given by

$$w_{32}(\text{new}) = w_{32}(\text{old}) + x_3 y_2 = 0 + 1 \times 1 = 1$$
$$w_{42}(\text{new}) = w_{42}(\text{old}) + x_4 y_2 = 0 + 1 \times 1 = 1$$

Since, $x_1 = x_2 = y_1 = 0$, the other weights remains the same. The final weights after presenting the fourth input vector are

$$w_{11} = 2, w_{21} = 0, w_{31} = 1, w_{41} = 1$$
$$w_{12} = 1, w_{22} = 1, w_{32} = 1, w_{42} = 1$$

Thus, the weight matrix in matrix form is

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

2. Train the heteroassociative memory network using outer products rule to store input row vectors $s = (s_1, s_2, s_3, s_4)$ to the output row vectors $t = (t_1, t_2)$. Use the vector pairs as given in Table 2.

Table 2

Input and targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	1	0	1	0
2nd	1	0	0	1	1	0
3rd	1	1	0	0	0	1
4th	0	0	1	1	0	1

Solution: Use outer products rule to determine the weight matrix:

$$W = \sum_{p=1}^{P} s^T(p)\, t(p)$$

For 1st pair: The input and output vectors are $s = (1\ 0\ 1\ 0)$, $t = (1\ 0)$. For $p = 1$,

$$s^T(p)\, t(p) = s^T(1)\, t(1)$$
$$= \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}_{4\times 1} \begin{bmatrix} 1 & 0 \end{bmatrix}_{1\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix}_{4\times 2}$$

For 2nd pair: The input and output vectors are $s = (1\ 0\ 0\ 1)$, $t = (1\ 0)$. For $p = 2$,

$$s^T(p)\, t(p) = s^T(2)\, t(2)$$
$$= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}_{4\times 1} \begin{bmatrix} 1 & 0 \end{bmatrix}_{1\times 2} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix}_{4\times 2}$$

For 3rd pair: The input and output vectors are $s = (1\ 1\ 0\ 0)$, $t = (0\ 1)$. For $p = 3$,

$$s^T(p)\, t(p) = s^T(3)\, t(3)$$
$$= \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix}_{4\times 1} \begin{bmatrix} 0 & 1 \end{bmatrix}_{1\times 2} = \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}_{4\times 2}$$

For 4th pair: The input and output vectors are $s = (0\ 0\ 1\ 1)$, $t = (0\ 1)$. For $p = 4$,

$$s^T(p)\, t(p) = s^T(4)\, t(4)$$
$$= \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}_{4\times 1} \begin{bmatrix} 0 & 1 \end{bmatrix}_{1\times 2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}_{4\times 2}$$

The final weight matrix is the summation of all the individual weight matrices obtained for each pair.

$$W = \sum_{p=1}^{4} s^T(p)\, t(p)$$
$$= s^T(1)t(1) + s^T(2)t(2) + s^T(3)t(3) + s^T(4)t(4)$$
$$= \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}$$
$$W = \begin{bmatrix} 2 & 1 \\ 0 & 1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix}$$

3. Train a heteroassociative memory network to store the input vectors $s = (s_1, s_2, s_3, s_4)$ to the output vectors $t = (t_1, t_2)$. The vector pairs are given in Table 3. Also test the performance of the network using its training input as testing input.

Table 3

Input and targets	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	0	0	0	1
2nd	1	1	0	0	0	1
3rd	0	0	0	1	1	0
4th	0	0	1	1	1	0

Solution: The network architecture for the given input–target vector pair is shown in Figure 2. Training the network means the determination of weights of the network. Here outer products rule is used to determine the weight.

The weight matrix W using outer products rule is given by

$$W = \sum_{p=1}^{P} s^T(p)\, t(p)$$

For $p = 1$ to 4,

$$W = \sum_{p=1}^{4} s^T(p)\, t(p)$$

$$= s^T(1)t(1) + s^T(2)t(2) + s^T(3)t(3) + s^T(4)t(4)$$

$$= \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 1 & 0 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

This is the final weight of the matrix.

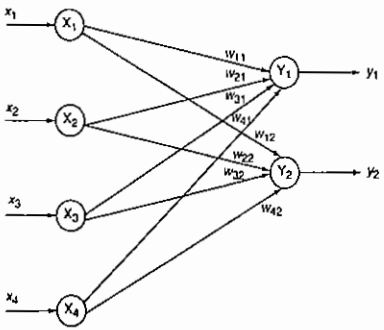


Figure 2 Network architecture.

Testing the Network

Method I

The testing algorithm for a heteroassociative memory network is used to test the performance of the net. The weight obtained from training algorithm is the initial weight in testing algorithm.

For 1st testing input

Step 0: Initialize the weights:

$$W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

Step 1: Performs Steps 2–4 for each testing input–output vector.

Step 2: Set the activations, $x = [1\ 0\ 0\ 0]$.

Step 3: Compute the net input, $n = 4$, $m = 2$. For $i = 1$ to 4 and $j = 1$ to 2:

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

$$y_{in1} = \sum_{i=1}^{n} x_i w_{i1}$$

$$= x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$

$$= 1 \times 0 + 0 \times 0 + 0 \times 1 + 0 \times 2 = 0$$

$$y_{in2} = \sum_{i=1}^{n} x_i w_{i2}$$

$$= x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$

$$= 1 \times 2 + 0 \times 1 + 0 \times 0 + 0 \times 0 = 2$$

Step 4: Applying activation over the net input to calculate the output.

$$y_1 = f(y_{in1}) = f(0) = 0$$
$$y_2 = f(y_{in2}) = f(2) = 1$$

The output is [0, 1] which is correct response for first input pattern.

For 2nd testing input

Set the activation $x = [1\ 1\ 0\ 0]$. Computing the net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 0 + 0 = 0$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 2 + 1 + 0 + 0 = 3$$

Compute the output by applying activations over net input,

$$y_1 = f(y_{in1}) = f(0) = 0$$
$$y_2 = f(y_{in2}) = f(3) = 1$$

The output is [0, 1] which is correct response for second input pattern.

For 3rd testing input

Set the activation $x = [0\ 0\ 0\ 1]$. Computing net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 0 + 2 = 2$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 0 + 0 + 0 + 0 = 0$$

Calculate output of the network,

$$y_1 = f(y_{in1}) = f(2) = 1$$
$$y_2 = f(y_{in2}) = f(0) = 0$$

The output is [1 0] which is correct response for third testing input pattern.

For 4th testing input

Set the activation $x = [0\ 0\ 1\ 1]$. Calculating the net input, we obtain

$$y_{in1} = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41}$$
$$= 0 + 0 + 1 + 2 = 3$$
$$y_{in2} = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42}$$
$$= 0 + 0 + 0 + 0 = 0$$

Calculate the output of the network,

$$y_1 = f(y_{in1}) = f(3) = 1$$
$$y_2 = f(y_{in2}) = f(0) = 0$$

The output is [1 0] which is correct response for fourth testing input pattern.

Method II

Since net input is the dot product of the input row vector with the column of weight matrix, hence a method using matrix multiplication can be used to test performance of network. The initial weights for the network are

$$W = \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

The binary activations are used, i.e.,

$$f(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \le 0 \end{cases}$$

For 1st testing input

Set the activation $x = [1\ 0\ 0\ 0]$. The net input is given by $y_{in} = xW$ (in vector form):

$$[y_{in1}\ y_{in2}] = [1\ 0\ 0\ 0]_{1\times 4} \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}_{4\times 2}$$

$$= [0+0+0+0 \quad 2+0+0+0]$$
$$= [0 \quad 2]$$

Applying activations over the net input, we get

$$[y_1\ y_2] = [0 \quad 1]$$

The correct response is obtained for first testing input pattern.

For 2nd testing input

Set the activation $x = [1\ 1\ 0\ 0]$. The net input is obtained by

$$[y_{in1}\ y_{in2}] = [1\ 1\ 0\ 0] \begin{bmatrix} 0 & 2 \\ 0 & 1 \\ 1 & 0 \\ 2 & 0 \end{bmatrix}$$

$$= [0+0+0+0 \quad 2+1+0+0]$$
$$= [0 \quad 3]$$

Apply activations over the net input to obtain output, we get

$$[y_{in1}\ y_{in2}] = [0 \quad 1]$$

The correct response is obtained for second testing input.

For 3rd testing input

Set the activation $x = [0\ 0\ 0\ 1]$. The net input is obtained by

$$[y_{in1}\ y_{in2}] = [0\ 0\ 0\ 1]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$
$$= [0+0+0+2 \quad 0+0+0+0]$$
$$= [2 \quad 0]$$

Applying activations to calculate the output, we get

$$[y_1 \quad y_2] = [1 \quad 0]$$

Thus, correct response is obtained for third testing input.

For 4th testing input

Set the activation $x = [0 \quad 0 \quad 1 \quad 1]$. The net input is calculated as

$$[y_{in1}\ y_{in2}] = [0 \quad 0 \quad 1 \quad 1]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$
$$= [0+0+1+2 \quad 0+0+0+0]$$
$$= [3 \quad 0]$$

The output is obtained by applying activations over the net input:

$$[y_1 \quad y_2] = [1 \quad 0]$$

The correct response is obtained for fourth testing input. Thus, training and testing of a hetero associative network is done here.

4. For Problem 3, test a heteroassociative network with a similar test vector and unsimilar test vector.

Solution: The heteroassociative network has to be tested with similar and unsimilar test vector.

With similar test vector: From Problem 3, the second input vector is $x = [1\ 1\ 0\ 0]$ with target $y = [0\ 1]$. To test the network with a similar vector, making a change in one component of the input vector, we get

$$x = [0\ 1\ 0\ 0]$$

The weight matrix is

$$W = \begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

The net input is calculated for the similar vector,

$$[y_{in1}\ y_{in2}] = [0\ 1\ 0\ 0]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$
$$= [0+0+0+0 \quad 0+1+0+0]$$
$$= [0 \quad 1]$$

The output is obtained by applying activations over the net input

$$[y_1 \quad y_2] = [0 \quad 1]$$

The correct response same as the target is found, hence the vector similar to the input vector is recognized by the network.

With unsimilar input vector: The second input vector is $x = [1 \quad 1 \quad 0 \quad 0]$ with target $y = [0 \quad 1]$. To test the network with unsimilar vectors by making a change in two components of the input vector, we get

$$x = [0 \quad 1 \quad 1 \quad 0]$$

The weight matrix is

$$W = \begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$

The net input is calculated for unsimilar vector,

$$[y_{in1}\ y_{in2}] = [0\ 1\ 1\ 0]\begin{bmatrix}0 & 2\\ 0 & 1\\ 1 & 0\\ 2 & 0\end{bmatrix}$$
$$= [0+0+1+0 \quad 0+1+0+0]$$
$$= [1 \quad 1]$$

The output is obtained by applying activations over the net input

$$[y_1 \quad y_2] = [1 \quad 1]$$

The correct response is not obtained when the vector unsimilar to the input network is presented to the network.

5. Train a heteroassociative network to store the input vectors $s = (s_1\ s_2\ s_3\ s_4)$ to the output vector $t = (t_1\ t_2)$. The training input–target output vector pairs are in binary form. Obtain the weight vector in bipolar form. The binary vector pairs are as given in Table 4.

Table 4

	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	0	0	0	0	1
2nd	1	1	0	0	0	1
3rd	0	0	0	1	1	0
4th	0	0	1	1	1	0

Solution: In this case, the hybrid representation of the network is adopted to find the weight matrix in bipolar form. The weight matrix can be formed using

$$w_{11} = (2\times1-1)(2\times0-1) + (2\times1-1)(2\times0-1) + (2\times0-1)(2\times1-1) + (2\times0-1)(2\times1-1)$$
$$= -1-1-1-1 = -4$$
$$w_{12} = (2\times1-1)(2\times1-1) + (2\times1-1)(2\times1-1) + (2\times0-1)(2\times0-1) + (2\times0-1)(2\times0-1)$$
$$= 1+1+1+1 = 4$$
$$w_{21} = -1\times-1+1\times-1+-1\times1+-1\times1$$
$$= 1-1-1-1 = -2$$
$$w_{22} = -1\times1+1\times1+-1\times-1+-1\times-1$$
$$= -1+1+1+1 = 2$$
$$w_{31} = -1\times-1+-1\times-1+-1\times1+1\times1$$
$$= 1+1-1+1 = 2$$
$$w_{32} = -1\times1+-1\times1+-1\times-1+1\times-1$$
$$= -1-1+1-1 = -2$$
$$w_{41} = -1\times-1+-1\times-1+1\times1+1\times1$$
$$= 1+1+1+1 = 4$$
$$w_{42} = -1\times1+-1\times1+1\times-1+1\times-1$$
$$= -1-1-1-1 = -4$$

The weight matrix W is given by

$$W = \begin{bmatrix}w_{11} & w_{12}\\ w_{21} & w_{22}\\ w_{31} & w_{32}\\ w_{41} & w_{42}\end{bmatrix} = \begin{bmatrix}-4 & 4\\ -2 & 2\\ 2 & -2\\ 4 & -4\end{bmatrix}$$

6. Train a heteroassociative network to store the given bipolar input vectors $s = (s_1\ s_2\ s_3\ s_4)$ to the output vector $t = (t_1\ t_2)$. The bipolar vector pairs are as given in Table 5.

Table 5

	s_1	s_2	s_3	s_4	t_1	t_2
1st	1	−1	−1	−1	−1	1
2nd	1	1	−1	−1	−1	1
3rd	−1	−1	−1	1	1	−1
4th	−1	−1	1	1	1	−1

Solution: To store a bipolar vector pair, the weight matrix is

$$w_{ij} = \sum_{p=1}^{P} s_i(p)t_j(p)$$

If the outer products rule is used, then

$$W = \sum_{p} s^T(p)t(p)$$

For 1st pair

$$s = [1 \quad -1 \quad -1 \quad -1], \quad t = [-1 \quad 1]$$

$$s^T(1)t(1) = \begin{bmatrix}1\\ -1\\ -1\\ -1\end{bmatrix}\begin{bmatrix}-1 & 1\end{bmatrix} = \begin{bmatrix}-1 & 1\\ 1 & -1\\ 1 & -1\\ 1 & -1\end{bmatrix}$$

For 2nd pair

$$s = [1 \quad 1 \quad -1 \quad -1], \quad t = [-1 \quad 1]$$

$$s^T(2)t(2) = \begin{bmatrix}1\\ 1\\ -1\\ -1\end{bmatrix}\begin{bmatrix}-1 & 1\end{bmatrix} = \begin{bmatrix}-1 & 1\\ -1 & 1\\ 1 & -1\\ 1 & -1\end{bmatrix}$$

For 3rd pair

$$s = [-1 \quad -1 \quad -1 \quad 1], \quad t = [1 \quad -1]$$

$$s^T(3)s(3) = \begin{bmatrix} -1 \\ -1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix}$$

For 4th pair

$$s = [-1 \quad -1 \quad 1 \quad 1], \quad t = [1 \quad -1]$$

$$s^T(4)s(4) = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

The final weight matrix is

$$W = \sum_{p=1}^{4} s^T(p)s(p) = \begin{bmatrix} -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

7. For Problem 6, test the performance of the network with missing and mistaken data in the test vector.

Solution: *With missing data*

Let the test vector be $x = [0 \quad 1 \quad 0 \quad -1]$ with changes made in two components of second input vector $[1 \quad 1 \quad -1 \quad -1]$. Computing the net input, we get

$$[y_{in1} \quad y_{in2}] = [0 \quad 1 \quad 0 \quad -1] \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

$$= [0-2+0-4 \quad 0+2+0+4]$$

$$= [-6 \quad 6]$$

Applying activations to compute the output, we get

$$[y_1 \quad y_2] = [-1 \quad 1]$$

Thus, the net has recognized the missing data.

With mistaken data: Let the test vector be $x = [-1 \quad 1 \quad 1 \quad -1]$ with changes made in two components of second input vector $[1 \quad 1 \quad -1 \quad -1]$. Computing the net input for the test vector, using the final weights obtained in Problem 6, as initial weight to test the test vector, we get

$$[y_{in1} \quad y_{in2}] = [-1 \quad 1 \quad 1 \quad -1] \begin{bmatrix} -4 & 4 \\ -2 & 2 \\ 2 & -2 \\ 4 & -4 \end{bmatrix}$$

$$= [0 \quad 0]$$

Applying the activations over the net input to calculate the output, we obtain

$$[y_1 \quad y_2] = [0 \quad 0]$$

Thus, the net does not recognize the mistaken data because the output obtained [0, 0] has a mismatch with the target vector $[-1 \quad 1]$.

8. Train the autoassociative network for input vector $[-1 \quad 1 \quad 1 \quad 1]$ and also test the network for the same input vector. Test the autoassociative network with one missing, one mistake, two missing and two mistake entries in test vector.

Solution: The input vector is $x = [-1 \quad 1 \quad 1 \quad 1]$. The weight vector is

$$W = \sum s^T(p)s(p) = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}_{4\times 1} [-1 \quad 1 \quad 1 \quad 1]_{1\times 4}$$

$$= \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}_{4\times 4}$$

Testing the network with same input vector: The test input is $[-1 \quad 1 \quad 1 \quad 1]$. The weight obtained above is used as the initial weight here. Computing the input, we get

$$y_{inj} = x \cdot W = [-1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-4 \quad 4 \quad 4 \quad 4]$$

Applying activations over the net input to calculate the output, we have

$$y_j = f(y_{inj}) = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ -1 & \text{if } y_{inj} \leq 0 \end{cases}$$

$$= [-1 \quad 1 \quad 1 \quad 1]$$

Hence the correct response is obtained.

Testing the network with one missing entry

- Test input $x = [0 \quad 1 \quad 1 \quad 1]$. Computing the input, we get

$$y_{inj} = x \cdot W = [0 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-3 \quad 3 \quad 3 \quad 3]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

Test input $x = [-1 \quad 1 \quad 0 \quad 1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W$$

$$= [-1 \quad 1 \quad 0 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-3 \quad 3 \quad 3 \quad 3]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

Testing the network with one mistake entry

- Test input $x = [-1 \quad -1 \quad 1 \quad 1]$. Computing net input, we get

$$y_{inj} = x \cdot W$$

$$= [-1 \quad -1 \quad 1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad 2]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

- Test input $x = [1 \quad 1 \quad 1 \quad 1]$. Computing net input, we get

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad 2]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

Testing the network with two missing entry

- Test input $x = [0 \quad 0 \quad 1 \quad 1]$. Computing net input, we get

$$y_{inj} = x \cdot W = [0 \quad 0 \quad 1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad 2]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

- Test input $x = [-1 \quad 0 \quad 0 \quad 1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W$$

$$= [-1 \quad 0 \quad 0 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad 2]$$

Applying the activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$ which is the correct response.

Testing the network with two mistaken entry

Test input $x = [-1 \quad -1 \quad -1 \quad 1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W$$

$$= [-1 \quad -1 \quad -1 \quad 1] \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

$$= [0 \quad 0 \quad 0 \quad 0]$$

Applying the activations over the net input, we get $y_j = [0 \quad 0 \quad 0 \quad 0]$ which is the incorrect response. Thus, the network with two mistakes is not recognized.

9. Check the autoassociative network for input vector $[1 \quad 1 \quad -1]$. Form the weight vector with no self-connection. Test whether the net is able to recognize with one missing entry.

Solution: Input vector $x = [1 \quad 1 \quad -1]$. The weight vector is

$$W = \sum s^T(p)s(p) = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} [1 \quad 1 \quad -1]$$

$$= \begin{bmatrix} 1 & 1 & -1 \\ 1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix}$$

The weight vector with no self-connection (make the diagonal elements in the weight vector zero) is given by

$$W = \begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

Testing the network with one missing entry

- Test input $x = [1 \quad 0 \quad -1]$

$$y_{inj} = x \cdot W = [1 \quad 0 \quad -1] \begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$= [1 \quad 2 \quad -1]$$

Applying the activations, we get $y_j = [1 \quad 1 \quad -1]$, hence a correct response is obtained.

- Test input $x = [1 \quad 1 \quad 0]$

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 0] \begin{bmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ -1 & -1 & 0 \end{bmatrix}$$

$$= [1 \quad 1 \quad -2]$$

Applying the activations, we get $y_j = [1 \quad 1 \quad -1]$, hence a correct response is obtained.

10. Use outer products rule to store the vectors $[1 \quad 1 \quad 1 \quad 1]$ and $[-1 \quad 1 \quad 1 \quad -1]$ in an autoassociative network. (a) Find the weight matrix (do not set diagonal term to zero). (b) Test the vector using $[1 \quad 1 \quad 1 \quad 1]$ as input. (c) Test the vector $[-1 \quad 1 \quad 1 \quad -1]$ as input. (d) Test the net using $[1 \quad 1 \quad 1 \quad 0]$ as input. (e) Repeat (a)–(d) with the diagonal terms in the weight matrix to be zero.

Solution:

- Weight matrix for $[1 \quad 1 \quad 1 \quad 1]$ is

$$W_1 = \sum s^T(p)s(p)$$

$$= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [1 \quad 1 \quad 1 \quad 1] = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Weight matrix for $[-1 \quad 1 \quad 1 \quad -1]$ is

$$W_2 = \sum s^T(p)s(p) = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} [-1 \quad 1 \quad 1 \quad -1]$$

$$= \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

The weight matrix to store two vectors is

$$W = w_1 + w_2$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$$

- *Test the vector using [1 1 1 1] as input*

Test vector $x = [1 \quad 1 \quad 1 \quad 1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$$

$$= [4 \quad 4 \quad 4 \quad 4]$$

Applying the activations to calculate output, we get $y_j = [1 \quad 1 \quad 1 \quad 1]$, hence correct response is obtained.

- *Test the vector using [−1 1 1 −1] as input*

Test vector $x = [-1 \quad 1 \quad 1 \quad -1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W = [-1 \quad 1 \quad 1 \quad -1] \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$$

$$= [-4 \quad 4 \quad 4 \quad -4]$$

Applying the activations to calculate output, we get $y_j = [-1 \quad 1 \quad 1 \quad -1]$, hence correct response is obtained.

- *Test the net using [1 1 1 0] as input*

Test vector $x = [1 \quad 1 \quad 1 \quad 0]$. Computing net input, we obtain

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 1 \quad 0] \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$$

$$= [2 \quad 4 \quad 4 \quad 2]$$

Applying the activations, we get $y_j = [1 \quad 1 \quad 1 \quad 1]$, hence the known response is obtained.

- *Repeat parts a-d with diagonal element in weight matrix set to zero*

(i) The weight matrix is

$$W = \begin{bmatrix} 2 & 0 & 0 & 2 \\ 0 & 2 & 2 & 0 \\ 0 & 2 & 2 & 0 \\ 2 & 0 & 0 & 2 \end{bmatrix}$$

(ii) Test the vector using $x = [1 \quad 1 \quad 1 \quad 1]$ as input. Computing net input, we obtain

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 1 \quad 1] \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

$$= [2 \quad 2 \quad 2 \quad 2]$$

Applying the activations, we get $y_j = [1 \quad 1 \quad 1 \quad 1]$, hence correct response is obtained.

(iii) Test the vector using $x = [-1 \quad 1 \quad 1 \quad -1]$ as input.

$$y_{inj} = x \cdot W = [-1 \quad 1 \quad 1 \quad -1] \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad -2]$$

Applying the activations to calculate output, we get $y_j = [-1 \quad 1 \quad 1 \quad -1]$, hence an unknown response is obtained.

(iv) Test the vector using $x = [1 \quad 1 \quad 1 \quad 0]$ as input.

$$y_{inj} = x \cdot W = [1 \quad 1 \quad 1 \quad 0] \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 2 & 0 & 0 \\ 2 & 0 & 0 & 0 \end{bmatrix}$$

$$= [0 \quad 2 \quad 2 \quad 2]$$

Applying the activations to calculate output, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$, hence an unknown response is obtained.

11. Find the weight matrix required to store the vectors $[1 \quad 1 \quad -1 \quad -1]$, $[-1 \quad 1 \quad 1 \quad -1]$ and $[-1 \quad 1 \quad -1 \quad 1]$ into W_1, W_2, W_3 respectively. Calculate the toral weight matrix to store all the

vector and check whether it is capable of recognizing the same vectors presented. Let the weight matrix be with no self-connection.

Solution: For the first vector [1 1 −1 −1]

$$W_1 = \sum s^T(p)s(p) = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [1 \quad 1 \quad -1 \quad -1]$$

$$= \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

With no self-connection,

$$W_{10} = \begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix}$$

For the second vector [−1 1 1 −1]

$$W_2 = \sum s^T(p)s(p) = \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} [-1 \quad 1 \quad 1 \quad -1]$$

$$= \begin{bmatrix} 1 & -1 & -1 & 1 \\ -1 & 1 & 1 & -1 \\ -1 & 1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

With no self-connection,

$$W_{20} = \begin{bmatrix} 0 & -1 & -1 & 1 \\ -1 & 0 & 1 & -1 \\ -1 & 1 & 0 & -1 \\ 1 & -1 & -1 & 0 \end{bmatrix}$$

For the third vector [−1 1 −1 1]

$$W_3 = \sum s^T(p)s(p) = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix} [-1 \quad 1 \quad -1 \quad 1]$$

$$= \begin{bmatrix} 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & 1 \\ -1 & 1 & -1 & 1 \end{bmatrix}$$

With no self-connection,

$$W_{30} = \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

The total weight matrix required to store all this is

$$W = W_{10} + W_{20} + W_{30}$$

$$= \begin{bmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Testing the network

- With first vector $x = [1 \quad 1 \quad -1 \quad -1]$. Net input is given by

$$y_{inj} = x \cdot W$$

$$= [1 \quad 1 \quad -1 \quad -1] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

$$= [1 \quad 1 \quad -1 \quad -1]$$

Applying activations, we get $y_j = [1 \quad 1 \quad -1 \quad -1]$ which is the correct response.

- With second vector $x = [-1 \quad 1 \quad 1 \quad -1]$. Net input is given by

$$y_{inj} = x \cdot W$$

$$= [-1 \quad 1 \quad 1 \quad -1] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

$$= [-1 \quad 1 \quad 1 \quad -1]$$

Applying activations, we get $y_j = [-1 \quad 1 \quad 1 \quad -1]$ which is the correct response.

- With third vector $x = [-1 \quad 1 \quad -1 \quad 1]$. Computing net input, we get

$$y_{inj} = x \cdot W$$

$$= [-1 \quad 1 \quad -1 \quad 1] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & -1 & -1 \\ -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

$$= [-1 \quad 1 \quad -1 \quad 1]$$

Applying activations, we get $y_j = [-1 \quad 1 \quad -1 \quad 1]$ which is the correct response.

Thus, the network is capable of recognizing the vectors.

12. Construct an autoassociative network to store vectors [−1 1 1 1]. Use iterative autoassociative network to test the vector with three missing elements.

Solution: The input vector is $x = [-1 \quad 1 \quad 1 \quad 1]$. The weight matrix is obtained as

$$W = \sum_p s^T(p)s(p) = \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [-1 \quad 1 \quad 1 \quad 1]$$

$$= \begin{bmatrix} 1 & -1 & -1 & -1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \end{bmatrix}$$

The weight matrix with no self-connection is

$$W_0 = \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & 1 & 1 & 0 \end{bmatrix}$$

Test vector with three missing elements

- For test input vector $x = [-1 \quad 0 \quad 0 \quad 0]$, the net input is calculated as

$$y_{inj} = x \cdot W$$

$$= [-1 \quad 0 \quad 0 \quad 0] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & 1 & 1 & 0 \end{bmatrix}$$

$$= [0 \quad 1 \quad 1 \quad 1]$$

Applying activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$, i.e., known response is obtained.

- For test input vector $x = [0 \quad 0 \quad 0 \quad 1]$. Computing net input, we obtain

$$y_{inj} = x \cdot W$$

$$= [0 \quad 0 \quad 0 \quad 1] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & 1 & 1 & 0 \end{bmatrix}$$

$$= [-1 \quad 1 \quad 1 \quad 0]$$

Applying activations, we get $y_j = [-1 \quad 1 \quad 1 \quad -1]$, i.e., unknown response is obtained. Iterate the network again using the net input calculated as input vector:

$$y_{inj} = [-1 \quad 1 \quad 1 \quad 0] \begin{bmatrix} 0 & -1 & -1 & -1 \\ -1 & 0 & 1 & 1 \\ -1 & 1 & 0 & 1 \\ -1 & 1 & 1 & 0 \end{bmatrix}$$

$$= [-2 \quad 2 \quad 2 \quad 3]$$

Applying activations, we get $y_j = [-1 \quad 1 \quad 1 \quad 1]$, i.e., known response is obtained after iteration. Thus, iterative autoassociative network recognizes the test pattern. Similarly, the network can be tested for the test input vectors [0 1 0 0] and [0 0 1 0].

13. Construct an autoassociative discrete Hopfield network with input vector [1 1 1 −1]. Test the discrete Hopfield network with missing entries in first and second components of the stored vector.

Solution: The input vector is $x = [1 \quad 1 \quad 1 \quad -1]$. The weight matrix is given by

$$W = \sum s^T(p)s(p) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix} [1 \quad 1 \quad 1 \quad -1]$$

$$= \begin{bmatrix} 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ 1 & 1 & 1 & -1 \\ -1 & -1 & -1 & 1 \end{bmatrix}$$

The weight matrix with no self-connection is

$$W = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

The binary representation for the given input vector is [1 1 1 0]. We carry out asynchronous updation of weights here. Let it be Y_1, Y_4, Y_3, Y_2.

For the test input vector with two missing entries in first and second components of the stored vector.

Iteration I

Step 0: Weights are initialized to store patterns:

$$W = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Step 1: The input vector is $x = [0\ 0\ 1\ 0]$.

Step 2: For this vector $y = [0\ 0\ 1\ 0]$.

Step 3: Choose unit Y_1 for updating its activations:

$$y_{in1} = x_1 + \sum_{j=1}^{4} y_j w_{j1}$$

$$= 0 + [0\ 0\ 1\ 0] \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} = 0 + 1 = 1$$

Applying activations we get $y_{in1} > 0 \Rightarrow y_1 = 1$. Broadcasting y_1 to all other units, we get

$y = [1\ 0\ 1\ 0] \rightarrow$ No convergence

Step 4: Choosing unit Y_4 for updating its activations:

$$y_{in4} = x_4 + \sum_{j=1}^{4} y_j w_{j4}$$

$$= 0 + [1\ 0\ 1\ 0] \begin{bmatrix} -1 \\ -1 \\ -1 \\ 0 \end{bmatrix}$$

$$= 0 - 1 - 1 = -2$$

Applying activations we get $y_{in4} < 0 \Rightarrow y_4 = 0$. Therefore, $y = [1\ 0\ 1\ 0] \rightarrow$ No convergence.

Step 5: Choosing unit Y_3 for updating its activations:

$$y_{in3} = x_3 + \sum_{j=1}^{4} y_j w_{j3}$$

$$= 1 + [1\ 0\ 1\ 0] \begin{bmatrix} 1 \\ 1 \\ 0 \\ -1 \end{bmatrix}$$

$$= 1 + 1 = 2$$

Applying activations we get $y_{in3} > 0 \Rightarrow y_3 = 1$. Therefore, $y = [1\ 0\ 1\ 0] \rightarrow$ No convergence.

Step 6: Choosing unit Y_2 for updating its activations:

$$y_{in2} = x_2 + \sum_{j=1}^{4} y_j w_{j2}$$

$$= 0 + [1\ 0\ 1\ 0] \begin{bmatrix} 1 \\ 0 \\ 1 \\ -1 \end{bmatrix}$$

$$= 0 + 2 = 2$$

Applying activations we get $y_{in2} > 0 \Rightarrow y_2 = 1$. Therefore, $y = [1\ 1\ 1\ 0] \rightarrow$ Converges with vector x.

Thus, the output y has converged with vector x in this iteration itself. But, one more iteration can be done to check whether further activations are there or not.

Iteration 2

Step 0: Weights are initialized to store patterns.

$$W = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}$$

Step 1: The input vector is $x = [1\ 1\ 1\ 0]$.

Step 2: For this vector $y = [1\ 1\ 1\ 0]$.

Step 3: Choosing unit Y_1 for updating its activations:

$$y_{in1} = x_1 + \sum_{j=1}^{4} y_j w_{j1}$$

$$= 1 + [1\ 1\ 1\ 0] \begin{bmatrix} 0 \\ 1 \\ 1 \\ -1 \end{bmatrix} = 3$$

Apply activations we get $y_{in1} > 0 \Rightarrow y_1 = 1$. Now $y = [1\ 1\ 1\ 0]$.

Step 4: Choose unit Y_4 for updation.

$$y_{in4} = x_4 + \sum_{j=1}^{4} y_j w_{j4}$$

$$= 0 + [1\ 1\ 1\ 0] \begin{bmatrix} -1 \\ -1 \\ -1 \\ 0 \end{bmatrix} = -3$$

Applying activations we get $y_{in4} < 0 \Rightarrow y_4 = 0$. Therefore, $y = [1\ 1\ 1\ 0]$.

Step 5: Choose unit Y_3 for updation.

$$y_{in3} = x_3 + \sum_{j=1}^{4} y_j w_{j3}$$

$$= 0 + [1\ 1\ 1\ 0] \begin{bmatrix} 1 \\ 1 \\ 0 \\ -1 \end{bmatrix} = 3$$

Applying activations we get $y_{in3} > 0 \Rightarrow y_3 = 1$. Therefore, $y = [1\ 1\ 1\ 0]$.

Step 6: Choose unit Y_2 for updation.

$$y_{in2} = x_2 + \sum_{j=1}^{4} y_j w_{j2}$$

$$= 0 + [1\ 1\ 1\ 0] \begin{bmatrix} 1 \\ 0 \\ 1 \\ -1 \end{bmatrix} = 3$$

Applying activations we get $y_{in2} > 3 \Rightarrow y_2 = 1$. Therefore, $y = [1\ 1\ 1\ 0]$.

Thus, further iterations do not change the activation of any unit.

14. Construct an autoassociative network to store the vectors $x_1 = [1\ 1\ 1\ 1\ 1]$, $x_2 = [1\ -1\ -1\ 1\ -1]$, $x_3 = [-1\ 1\ -1\ -1\ -1]$. Find weight matrix with no self-connection. Calculate the energy of the stored patterns. Using discrete Hopfield network test patterns if the test pattern are given as $x_1 = [1\ 1\ 1\ -1\ 1]$, $x_2 = [1\ -1\ -1\ -1\ -1]$ and $x_3 = [1\ 1\ -1\ -1\ -1]$. Compare the test patters energy with the stored patterns energy.

Solution: The weights matrix for the three given vectors is

$$W = \sum s_i^T(p)\, s_j(p)$$

$$= \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [1\ 1\ 1\ 1\ 1] + \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix} [1\ -1\ -1\ 1\ -1]$$

$$+ \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} [-1\ 1\ -1\ -1\ -1]$$

$$= \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 \\ -1 & 1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 & -1 \\ -1 & 1 & 1 & -1 & 1 \end{bmatrix}$$

$$+\begin{bmatrix} 1 & -1 & 1 & 1 & 1 \\ -1 & 1 & -1 & -1 & -1 \\ 1 & -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 3 & -1 & 1 & 3 & 1 \\ -1 & 3 & 1 & -1 & 1 \\ 1 & 1 & 3 & 1 & 3 \\ 3 & -1 & 1 & 3 & 1 \\ 1 & 1 & 3 & 1 & 3 \end{bmatrix}$$

The weight matrix with no self-connection is

$$W_0 = \begin{bmatrix} 0 & -1 & 1 & 3 & 1 \\ -1 & 0 & 1 & -1 & 1 \\ 1 & 1 & 0 & 1 & 3 \\ 3 & -1 & 1 & 0 & 1 \\ 1 & 1 & 3 & 1 & 0 \end{bmatrix}$$

The energy function is defined as

$$E = -0.5[x\, W^T x^T]$$

Therefore the energy for the ith pattern is given by

$$E_i = -0.5[x_i\, W^T x_i^T]$$

Energy for first pattern

$$E_1 = -0.5[x_1 W^T x_1^T]$$

$$= -0.5\,[1 \;\; 1 \;\; 1 \;\; 1 \;\; 1]_{1\times 5} \begin{bmatrix} 0 & -1 & 1 & 3 & 1 \\ -1 & 0 & 1 & -1 & 1 \\ 1 & 1 & 0 & 1 & 3 \\ 3 & -1 & 1 & 0 & 1 \\ 1 & 1 & 3 & 1 & 0 \end{bmatrix}_{5\times 5} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}_{5\times 1}$$

$$= -0.5\,[1 \;\; 1 \;\; 1 \;\; 1 \;\; 1]_{1\times 5} \begin{bmatrix} 4 \\ 0 \\ 6 \\ 4 \\ 6 \end{bmatrix}_{5\times 1}$$

$$= -0.5\,[4+0+6+4+6]_{1\times 1}$$

$$= -0.5\,[20] = -10$$

Energy for second pattern

$$E_2 = -0.5[x_2 W^T x_2^T]$$

$$= -0.5\,[1 \;\; -1 \;\; -1 \;\; 1 \;\; -1] \begin{bmatrix} 0 & -1 & 1 & 3 & 1 \\ -1 & 0 & 1 & -1 & 1 \\ 1 & 1 & 0 & 1 & 3 \\ 3 & -1 & 1 & 0 & 1 \\ 1 & 1 & 3 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

$$= -0.5\,[1 \;\; -1 \;\; -1 \;\; 1 \;\; -1] \begin{bmatrix} 2 \\ -4 \\ -2 \\ 2 \\ -2 \end{bmatrix}$$

$$= -0.5\,[2+4+2+2+2] = -0.5\,[12] = -6$$

Energy for third pattern

$$E_3 = -0.5[x_3 W^T x_3^T]$$

$$= -0.5\,[-1 \;\; 1 \;\; -1 \;\; -1 \;\; -1] \begin{bmatrix} 0 & -1 & 1 & 3 & 1 \\ -1 & 0 & 1 & -1 & 1 \\ 1 & 1 & 0 & 1 & 3 \\ 3 & -1 & 1 & 0 & 1 \\ 1 & 1 & 3 & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

$$= -0.5\,[-1 \;\; 1 \;\; -1 \;\; -1 \;\; -1] \begin{bmatrix} -6 \\ 0 \\ -4 \\ -6 \\ -4 \end{bmatrix}$$

$$= -0.5\,[6+0+4+6+4] = -0.5\,[20] = -10$$

Applying test patterns

For first test pattern $x_1' = [1 \;\; 1 \;\; 1 \;\; -1 \;\; 1]$ and $y = [1 \;\; 1 \;\; 1 \;\; -1 \;\; 1]$. Choosing unit 4 for updation, we get

$$y_{in4} = x_4 + \sum_{j=1}^{4} y_j w_{j4}$$

$$= -1 + [1 \;\; 1 \;\; 1 \;\; -1 \;\; 1] \begin{bmatrix} 3 \\ -1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$= -1 + 3 - 1 + 1 - 0 + 1 = 3 > 0$$

Applying activations, we get $y_4 = 1$. Therefore, $x_1' = [1 \;\; 1 \;\; 1 \;\; 1 \;\; 1] \rightarrow$ convergence. The energy function is given by

$$E_1' = -0.5[x_1' W^T x_1'^T]$$

On substituting the corresponding values, we get

$$E_1' = -10$$

- For second test pattern $x_2' = [1 \; -1 \; -1 \; -1 \; -1]$ and $y = [1 \; -1 \; -1 \; -1 \; -1]$. Choosing unit 4 for updation, we get

$$y_{in4} = x_4 + \sum_{j=1}^{4} y_j w_{j4}$$

$$= -1 + [1 \;\; -1 \;\; -1 \;\; -1 \;\; -1] \begin{bmatrix} 3 \\ -1 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

$$= -1 + 3 + 1 - 1 - 0 - 1 = 1 > 0$$

Applying activations, we get $y_4 = 1$. Therefore, $x_2' = [1 \;\; -1 \;\; 1 \;\; -1] \rightarrow$ convergence. The energy function is given by

$$E_2' = -0.5[x_2' W^T x_2'^T]$$

$$= -0.5\,[1 \;\; -1 \;\; -1 \;\; 1 \;\; -1]\,[W^T] \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \\ -1 \end{bmatrix}$$

$$= -0.5\,[12] = -6$$

- For third test pattern $x_3' = [1 \;\; 1 \;\; -1 \;\; -1 \;\; -1]$ and $y = [1 \;\; 1 \;\; -1 \;\; -1 \;\; -1]$. Choosing unit 1 for updation, we get

$$y_{in1} = x_1 + \sum_{j=1}^{4} y_j w_{j1}$$

$$= 1 + [1 \;\; 1 \;\; -1 \;\; -1 \;\; -1] \begin{bmatrix} 0 \\ -1 \\ 1 \\ 3 \\ 1 \end{bmatrix}$$

$$= 1 + 1 - 1 - 3 - 1 = -5 < 0$$

Applying activations, we get $y = -1$. Therefore, modified $x_3' = [-1 \;\; 1 \;\; -1 \;\; -1 \;\; -1] \rightarrow$ convergence. The energy function is given by

$$E_3' = -0.5[x_3' W^T x_3'^T]$$

$$= -0.5[-1 \;\; 1 \;\; -1 \;\; -1 \;\; -1] \;\; [W^T] \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \\ -1 \end{bmatrix}$$

$$= -0.5[20] = -10$$

Thus, the energy of the stored pattern is same as that of the test pattern.

15. Construct and test a BAM network to associate letters E and F with simple bipolar input–output vectors. The target output for E is (−1, 1) and for F is (1, 1). The display matrix size is 5 × 3. The input patters are

"E"	"F"
* * *	* * *
* • •	* * *
* * *	* • •
* • •	* • •
* * *	* • •
Target output (−1, 1)	(1, 1)

Solution: The inputs are

Input pattern	Inputs	Targets	Weights
E	[1 1 1 1−1−1 1 1 1 1−1−1 1 1 1]	[−1, 1]	W_1
F	[1 1 1 1 1 1 1−1−1 1−1−1 1−1−1]	[1 1]	W_2

(i) X vectors as input: The weight matrix is obtained by

$$W = \sum s^T(p)\, t(p)$$

$$W_1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ 1 \\ 1 \end{bmatrix} [-1 \quad 1] = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ -1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [1 \quad 1] = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \end{bmatrix}$$

The total weight matrix is

$$W = W_1 + W_2 = \begin{bmatrix} -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \\ 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \\ 1 & 1 \\ -1 & -1 \\ -1 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 2 & 0 \\ 2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ 0 & -2 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \end{bmatrix}$$

Testing the network with test vectors "E" and "F."

- For test pattern E, computing net input we get

$$y_{in} = [1\ 1\ 1\ 1\ -1\ -1\ 1\ 1\ 1\ 1\ -1\ -1\ 1\ 1\ 1]_{1\times 15} \begin{bmatrix} 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 2 & 0 \\ -2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ 0 & -2 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \end{bmatrix}_{15\times 2}$$

$$= [-12 \quad 18]_{1\times 2}$$

Applying activations, we get $y = [-1\ 1]$, hence correct response is obtained.

- For test pattern F. Computing net input, we get

$$y_{in} = [1\ 1\ 1\ 1\ 1\ 1\ 1\ -1\ -1\ 1\ -1\ -1\ 1\ -1\ -1] \begin{bmatrix} 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 0 & 2 \\ 2 & 0 \\ 2 & 0 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \\ 0 & 2 \\ 0 & -2 \\ 0 & -2 \\ 0 & 2 \\ -2 & 0 \\ -2 & 0 \end{bmatrix}$$

$$= [12 \quad 18]$$

Applying activations over the net input, to calculate output, we get $y = [1 \quad 1]$, hence correct response is obtained.

(ii) Y vectors as input: The weight matrix when Y vectors are used as input is obtained as the transpose of the weight matrix when X vectors were presented as input, i.e.,

$$W^T = \begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & -2 & -2 \\ 2 & 2 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & -2 & -2 & 2 & 0 & 0 \end{bmatrix}$$

Testing the network

(a) For test pattern E, now the input is $[-1\ 1]$. Computing net input, we have

$$y_{in} = x \cdot W^T = [-1\ 1] \cdot \begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & -2 & -2 \\ 2 & 2 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & -2 & -2 & 2 & 0 & 0 \end{bmatrix}$$

$$= [2\ 2\ 2\ 2\ -2\ -2\ 2\ 2\ 2\ 2\ -2\ -2\ 2\ 2\ 2]$$

Applying the activation functions, we get

$$y = [1\ 1\ 1\ 1\ -1\ -1\ 1\ 1\ 1\ 1\ -1\ -1\ 1\ 1\ 1]$$

which is the correct response.

(b) For test pattern F, now the input is [1, 1]. Computing net input, we have

$$y_{in} = x \cdot W^T = [1\ 1] \begin{bmatrix} 0 & 0 & 0 & 0 & 2 & 2 & 0 & -2 & -2 & 0 & 0 & 0 & 0 & -2 & -2 \\ 2 & 2 & 2 & 2 & 0 & 0 & 2 & 0 & 0 & 2 & -2 & -2 & 2 & 0 & 0 \end{bmatrix}$$

$$= [2\ 2\ 2\ 2\ 2\ 2\ 2\ -2\ -2\ 2\ -2\ -2\ 2\ -2\ -2]$$

Applying the activation functions, we get

$$y = [1\ 1\ 1\ 1\ 1\ 1\ 1\ -1\ -1\ 1\ -1\ -1\ 1\ -1\ -1]$$

which is the correct response. Thus, a BAM network has been constructed and tested in both the directions from X to Y and Y to X.

16. (a) Find the weight matrix in bipolar form for the bidirectional associative memory using outer products rule for the following binary input–output vector pairs:

$s(1) = (1\ 0\ 0\ 0), \quad t(1) = (1\ 0)$
$s(2) = (1\ 0\ 0\ 1), \quad t(2) = (1\ 0)$
$s(3) = (0\ 1\ 0\ 0), \quad t(3) = (0\ 1)$
$s(4) = (0\ 1\ 1\ 0), \quad t(4) = (0\ 1)$

(b) Using the unit step function (with threshold 0) as the output units activation function, test the response of the network on each of the input patterns.
(c) Test the response of the network on various combinations of input patterns with "mistakes" or "missing" data.
(i) $[1\ 0\ -1\ -1]$; (ii) $[-1\ 0\ 0\ -1]$; (iii) $[-1\ 1\ 0\ -1]$; (iv) $[1\ 1\ -1\ -1]$; (v) $[1\ 1]$

Solution:

(a) The weight matrix for storing the four input vectors in bipolar form is

$$W = \sum_{p=1}^{4} s_i^T(p)\, t_j(p)$$

$$= \begin{bmatrix} 1 \\ -1 \\ -1 \\ -1 \end{bmatrix} [1\ -1] + \begin{bmatrix} 1 \\ -1 \\ -1 \\ 1 \end{bmatrix} [1\ -1] + \begin{bmatrix} -1 \\ 1 \\ -1 \\ -1 \end{bmatrix} [-1\ 1] + \begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix} [-1\ 1]$$

$$= \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ -1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ -1 & +1 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ -1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$W = \begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$

(b) The unit step function for binary with threshold 0 is used.

For Y layer $\Rightarrow$ $y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ y_j & \text{if } y_{inj} = 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$

For X layer $\Rightarrow$ $x_i = \begin{cases} 1 & \text{if } x_{ini} > 0 \\ x_i & \text{if } x_{ini} = 0 \\ 0 & \text{if } x_{ini} < 0 \end{cases}$

Presenting s-input pattern

- $s(1) = [1\ 0\ 0\ 0]$. Computing net input, we have

$$t_{inj} = [1\ 0\ 0\ 0] \begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$

$$= [4 \quad -4]$$

Applying activations we get $t_j = [1 \quad 0]$ which is the correct response.

- $s(2) = [1\ 0\ 0\ 1]$. Computing net input,

$$t_{inj} = [1\ 0\ 0\ 1] \begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$

$$= [6 \quad -6]$$

Applying activations we get $t_j = [1 \quad 0]$ which is the correct response.

- $s(3) = [0\ 1\ 0\ 0]$. Computing the net input, we have

$$t_{inj} = [0\ 1\ 0\ 0]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [-4\ \ 4]$$

Applying activations we get $t_j = [0\ \ 1]$ which is the correct response.

- $s(4) = [0\ 1\ 1\ 0]$. Computing the net input, we get

$$t_{inj} = [0\ 1\ 1\ 0]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [-6\ \ 6]$$

Applying activations we get $t_j = [0\ \ 1]$ which is the correct response.

Presenting t-input pattern

- $t(1) = [1\ \ 0]$. Computing the net input, we obtain

$$s_{ini} = [1\ 0]\begin{bmatrix} 4 & -4 & -2 & 2 \\ -4 & 4 & 2 & -2 \end{bmatrix}$$
$$= [4\ -4\ -2\ 2]$$

Applying activations we get $s_i = [1\ \ 0\ \ 0\ \ 1]$ which is the correct response.

- $t(3) = [0\ \ 1]$. Computing the net input, we obtain

$$s_{ini} = [0\ 1]\begin{bmatrix} 4 & -4 & -2 & 2 \\ -4 & 4 & 2 & -2 \end{bmatrix}$$
$$= [-4\ \ 4\ \ 2\ \ -2]$$

Applying activations we get $s_i = [0\ \ 1\ \ 1\ \ 0]$ which is the correct response.

On presenting the pattern $[1\ \ 0]$ we obtain only $[1\ \ 0\ \ 0\ \ 1]$ and not $[1\ \ 0\ \ 0\ \ 0]$. Similarly, on presenting the pattern $[0\ \ 1]$ we obtain only $[0\ \ 1\ \ 1\ \ 0]$ and not $[0\ \ 1\ \ 0\ \ 0]$. This depends upon the missing data entries.

(c) Test response of network

(i) Here $x = [1\ \ 0\ \ -1\ \ -1]$. Calculating net input, we get

$$y_{inj} = x \cdot W$$
$$= [1\ \ 0\ \ -1\ \ -1]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [4\ \ -4]$$

Applying activations we get $y_j = [1\ \ 0]$ which is the correct response.

(ii) Here $x = [-1\ \ 0\ \ 0\ \ -1]$. Calculating the net input, we get

$$y_{inj} = [-1\ \ 0\ \ 0\ \ -1]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [-6\ \ 6]$$

Applying activations we get $y_j = [0\ \ 1]$ which is the correct response.

(iii) Here $x = [-1\ \ 1\ \ 0\ \ -1]$. Calculating the net input, we get

$$y_{inj} = [-1\ \ 1\ \ 0\ \ -1]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [-1\ \ 0\ \ 1\ \ 0]$$

Applying activations we get $y_j = [0\ \ 1]$ which is the correct response.

(iv) Here $x = [1\ \ 1\ \ -1\ \ -1]$. Calculating the net input, we get

$$y_{inj} = [1\ \ 1\ \ -1\ \ -1]\begin{bmatrix} 4 & -4 \\ -4 & 4 \\ -2 & 2 \\ 2 & -2 \end{bmatrix}$$
$$= [0\ \ 0]$$

Applying the previous activation and taking closely related pattern activation we get $y_j = [0\ \ 1]$.

(v) $Y = [1\ \ 1]$. Computing the net input, we get

$$x_{ini} = [1\ \ 1]\begin{bmatrix} 4 & -4 & -2 & 2 \\ -4 & 4 & 2 & -2 \end{bmatrix}$$
$$= [0\ \ 0\ \ 0\ \ 0]$$

Thus, in this case since all the x_{inj} values are zero, to apply the activation function it may take the previous x_i values for $x_{ini} = 0$. Hence the closely related pattern can be taken to obtain the correct response.

17. Find the hamming distance and average hamming distance for the two given input vectors below.

$X_1 = [1\ 1\ -1\ -1\ -1\ 1\ -1\ -1\ -1\ 1\ -1\ -1]$

$X_2 = [-1\ 1\ 1\ -1\ 1\ -1\ 1\ 1\ -1\ 1\ -1\ 1]$

Solution: The hamming distance is number of different bits in two binary or bipolar vectors. Here

$$H[X_1, X_2] = 8$$

Average hamming distance equals

$$\frac{1}{n} H[X_1, X_2] = \frac{1}{12} \times 8 = \frac{8}{12}$$

("n" is the no. of components in given vector.)

4.11 Review Questions

1. What is content addressable memory?
2. Specify the functional difference between a RAM and a CAM.
3. Indicate the two main types of associative memory.
4. State the advantages of associative memory.
5. Discuss the limitations of associative memory network.
6. Explain the Hebb rule training algorithm used in pattern association.
7. State the outer products rule used for training pattern association networks.
8. Draw the architecture of an autoassociative network.
9. Explain the testing algorithm adopted to test an autoassociative network.
10. What is a heteroassociative memory network?
11. With a neat architecture, explain the training algorithm of a heteroassociative network.
12. What is a bidirectional associative memory network?
13. Is it true that input patterns may be applied at the outputs of a BAM?
14. List the activation functions used in BAM net.
15. What are the two types of BAM?
16. How are the weights determined in a discrete BAM?
17. State the testing algorithm of a discrete BAM.
18. What is the activation function used in continuous BAM?
19. Define hamming distance and storage capacity.
20. What is an energy function of a discrete BAM?
21. What is a Hopfield net?
22. Compare and contrast BAM and Hopfield networks.
23. Mention the applications of Hopfield network.
24. What is the necessity of weights with no self-connection?
25. Why are symmetrical weights and weights with no self-connection important in discrete Hopfield net?

26. What is a recurrent neural network?
27. What are the two types of Hopfield net?
28. Draw the architecture of discrete Hopfield net.
29. State the testing algorithm used in discrete Hopfield network.
30. What is the energy function of a discrete Hopfield network?
31. Mention the formula used for determining the storage capacity of a discrete Hopfield net.
32. Discuss in detail on continuous Hopfield network.
33. Make an analysis of energy function of a continuous Hopfield network.
34. What are iterative autoassociative memory nets?
35. Explain in detail on linear autoassociative memory. State the conditions of linearity.
36. Write short note on brain-in-the box model.
37. What is the functional equivalent of a temporal associative memory network?

4.12 Exercise Problems

1. Train a heteroassociative memory network using Hebb rule to store input row vector $s = (s_1\ s_2\ s_3\ s_4)$ to the output row vector $t = (t_1\ t_2)$. The vector pairs are given as below:

$$s(1) = (1\ 0\ 0\ 1), \quad t(1) = (1\ 0)$$
$$s(2) = (1\ 1\ 1\ 1), \quad t(2) = (1\ 0)$$
$$s(3) = (1\ 1\ 0\ 0), \quad t(3) = (0\ 1)$$
$$s(4) = (0\ 0\ 1\ 1), \quad t(4) = (0\ 1)$$

2. Construct and test a heteroassociative memory network using outer products rule to store the given input–target vector pairs:

$$s(1) = (1\ 0\ 1), \quad t(1) = (1\ 0)$$
$$s(2) = (0\ 1\ 1), \quad t(2) = (0\ 1)$$

3. Construct and test a heteroassociative memory net to store the given vector pairs:

$$s(1) = (0\ 0\ 0\ 1), \quad t(1) = (0\ 1)$$
$$s(2) = (0\ 0\ 1\ 1), \quad t(2) = (0\ 1)$$
$$s(3) = (0\ 1\ 0\ 0), \quad t(3) = (1\ 0)$$
$$s(4) = (1\ 1\ 0\ 0), \quad t(4) = (1\ 0)$$

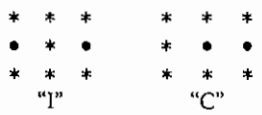
Also test the network with "noisy" input patterns included.

4. Construct and train a heteroassociative network to store the following input–output vector pair. The training input–target output vector pairs are in binary form. Obtain the weight vector in bipolar form. The binary vector pairs are:

$$s(1) = (1\ 0), \quad t(1) = (0\ 1)$$
$$s(2) = (1\ 1), \quad t(2) = (1\ 0).$$

Also test the performance of the network with missing and mistaken data.

5. Construct a heteroassociative network for the pattern given below:



The target of "I" and "C" are $(1, -1)$ and $(-1, 1)$ respectively. Store the pattern and as well recognize the pattern.

6. Train an autoassociative network for input vector $[-1\ 1\ 1\ -1]$ and also test the network with same input vector. Test the autoassociative network with one missing, one mistake, two missing and two mistake entries in test vector.
7. Check the autoassociative network for input vector $[-1\ -1\ 1]$. Form the weight vector with no self-connection. Test whether the net is able to recognize with one missing and two missing data. Comment on network performance.
8. Use outer products rule to store vectors $[-1\ -1\ -1\ 1]$ and $[1\ 1\ 1\ -1]$ in an autoassociative network.

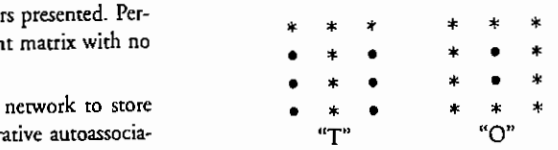
- Find the weight without setting diagonal terms to zero.
- Test vector using $[-1\ -1\ -1\ -1]$ as input.
- Test network using $[1\ 1\ 1\ 1]$ as input.
- Test the net using $[0\ 1\ 1\ 0]$ as input.
- Repeat (a)–(d) with diagonal elements set to zero.

9. Find the weight matrix required to store the vectors $[1\ 1\ -1\ 1\ -1]$, $[1\ 1\ 1\ 1\ -1]$, $[-1\ -1\ 1\ 1\ -1]$ and $[1\ 1\ -1\ -1\ 1]$ in w_1, w_2, w_3, w_4, respectively. Calculate the total weight matrix to store all the vectors and check whether it is capable of recognizing the same vectors presented. Perform the association for weight matrix with no self-connection.
10. Construct an autoassociative network to store vector $[1\ 1\ -1\ +1]$. Use iterative autoassociative network to test the vector with three missing elements.
11. Construct and test an associative discrete Hopfield network with input vector $[1\ -1\ 1\ 1]$. Test the network with missing entries in first and fourth components of the stored vector.
12. Construct an autoassociative network to store the vectors $x_1 = [1\ 1\ 1\ 1\ 1\ -1]$, $x_2 = [-1\ -1\ -1\ 1\ 1\ 1]$, $x_3 = [1\ 1\ 1\ -1\ -1\ -1]$. Find weight matrix with no self-connection. Calculate the energy of the stored patterns.
13. Consider a two node continuous Hopfield network. Assume the conductance is $gr_1 = gr_2 = 3$ mho. The gain parameter is $\lambda = 1.2$ and the external inputs are zero. Calculate the accurate energy value of the state $y = [0.1\ 0.1]^T$
14. Design a linear heteroassociate network that associates the following pairs of vectors.

$$x_1 = [1, 3, -5, 1]^T,\ y_1 = [0\ 0\ 0]^T$$
$$x_2 = [2, 2, 0, -4]^T,\ y_2 = [1\ 0\ 1]^T$$
$$x_3 = [1, 0, -3, 4]^T,\ y_3 = [0\ 1\ 1]^T$$

Verify that vectors x_1, x_2 and x_3 are linearly independent. Compute weight matrix of linear associates.

15. Consider a discrete Hopfield network with a synchronous update.
    - Show that if all given pattern vectors are orthogonal, then every original pattern is an global minimum.
    - Show that in general other global minima exist.
16. Construct and test a BAM network to associate letters T and O with simple bipolar input–output vectors. The target output for T is $(1, -1)$ and for O is $(1, 1)$. The display matrix size is 4×3. The input patterns are



17. Find the weight matrix in bipolar form for the BAM using outer products rule for the following binary input–output vector pairs.

$$s(1) = (1\ 0\ 0\ 0), \quad t(1) = (0\ 1)$$
$$s(2) = (0\ 1\ 1\ 0), \quad t(2) = (1\ 0)$$

Using the unit step function as the output unit's activation function, test the response of the network on each of the input patterns. Also test the response of the network on various combinations of input pattern with "mistakes" or "missing" data.

18. Find the hamming distance and average hamming distance for the two given input vectors below:

$$X_1 = [1\ 1\ 1\ -1\ -1\ 1\ 1\ 1\ -1\ -1\ -1\ -1\ 1\ 1\ -1]$$
$$X_2 = [1\ 1\ -1\ 1\ 1\ 1\ -1\ 1\ -1\ -1\ 1\ 1\ 1\ -1\ 1\ 1\ -1]$$

19. Prove the stability of the continuous BAM using
    (a) Kohonen Grossberg theorem and
    (b) the Lyapunov theorem.

20. Design a BAM-based temporal associative memory with a threshold activation function to recall the following sequence:

$$s = \{[1\ 1\ 1\ 1\ -1\ 1\ 1], [1\ 1\ 1\ 1\ -1\ -1\ -1], [-1\ 1\ 1\ 1\ 1\ -1\ -1\ 1]\}$$

Compute the weight matrix W and check the recall of patterns in forward and backward directions.

4.13 Projects

1. Write a computer program to implement a heteroassociative memory network using Hebb rule to set the weights. Develop the input patterns and target output of your own.
2. Write a program to construct and test an autoassociative network to store numerical values from 0–9. Also create the patterns for 0–9 using a 5×3 array matrix. Add "noise" to the input signals and test the network.
3. Write a "C" program to implement a discrete Hopfield net to store the letters A–E. Form the input patterns for the letters in a 4×3 array matrix.
4. Write a computer program to implement a bipolar BAM. Allow 15 units in X layer and 3 units in Y layer use the program to store the following patterns (the X layer vectors are the letters given in the 5×3 arrays and the associated Y layer vectors are given below in each x pattern):

"A"	"B"	"C"
• * •	* * •	• * *
* • *	* • *	* • •
* * *	* * *	* • •
* • *	* • *	* • •
* • *	* * •	• * *
(1, 1, 1)	(−1, −1, 1)	(1, −1, 1)

"D"	"E"	"F"
* * •	* * *	* * *
* • *	* • •	* • •
* • *	* * *	* * •
* • *	* • •	* • •
* * •	* * *	* • •
(−1, 1, 1)	(1, 1, −1)	(−1, −1, −1)

Is it possible to store all six patterns at once? If not, how many can be stored at the same time? Perform some experiments with noisy data.

5 Unsupervised Learning Networks

Learning Objectives

- Definition of unsupervised networks.
- Gives details on fixed weight competitive nets like Maxnet, Mexican hat and Hamming net.
- Discusses the neighborhood topology of Kohonen self-organizing feature maps.
- Provides architecture, training algorithm, flowchart depicting training process and testing algorithm of different unsupervised networks like KSOFM, counterpropagation network, adaptive resonance theory and LVQ.
- Enhance the features and star topology of CPN network.
- Details the variants of LVQ (LVQ2, LVQ3) and ART (ART 1 and ART 2).
- Variety of solved problems using unsupervised learning network.

5.1 Introduction

In this chapter, the study is made on the second major learning paradigm–unsupervised learning. In this learning, there exists no feedback from the system (environment) to indicate the desired outputs of a network. The network by itself should discover any relationships of interest, such as features, patterns, contours, correlations or categories, classifications in the input data, and thereby translate the discovered relationships into outputs. Such networks are also called self-organizing networks. An unsupervised learning can judge how similar a new input pattern is to typical patterns already seen, and the network gradually learns what similarity is; the network may construct a set of axes along which to measure similarity to previous patterns, i.e., it performs principal component analysis, clustering, adaptive vector quantization and feature mapping. For example, when net has been trained to classify the input patterns into any one of the output classes, say, P, Q, R, S or T, the net may respond to both the classes, P and Q or R and S. In the case mentioned, only one of several neurons should fire, i.e., respond. Hence the network has an added structure by means of which the net is forced to make a decision, so that only one unit will respond. The process for achieving this is called competition. Practically, considering a set of students, if we want to classify them on the basis of evaluation performance, their score may be calculated, and the one whose score is higher than the others should be the winner. The same principle adopted here is followed in the neural networks for pattern classification. In this case, there may exist a tie; a suitable solution is presented even when a tie occurs. Hence these nets may also be called competitive nets. The extreme form of these competitive nets is called winner-take-all. The name itself implies that only one neuron in the competing group will possess a nonzero output signal at the end of competition.

There exist several neural networks that come under this category. To list out a few: Maxnet, Mexican hat, Hamming net, Kohonen self-organizing feature map, counterpropagation net, learning vector quantization (LVQ) and adaptive resonance theory (ART). These networks are dealt in detail in forthcoming sections. In the case of unsupervised learning, the net seeks to find patterns or regularity in the input data by forming clusters. ART networks are called clustering nets. In these types of clustering nets, there are as many input units as an input vector possessing components. Since each output unit represents a cluster, the number of output units will limit the number of clusters that can be formed.

The learning algorithm used in most of these nets is known as Kohonen learning. In this learning, the units update their weights by forming a new weight vector, which is a linear combination of the old weight vector and the new input vector. Also, the learning continues for the unit whose weight vector is closest to the input vector. The weight updation formula used in Kohonen learning for output cluster unit j is given as

$$w_j(\text{new}) = w_{\Theta j}(\text{old}) + \alpha \left[x - w_{\Theta j}(\text{old})\right]$$

where x is the input vector; $w_{\Theta j}$ the weight vector for unit j; α the learning rate whose value decreases monotonically as training continues. There exist two methods to determine the winner of the network during competition. One of the methods for determining the winner uses the square of the Euclidean distance between the input vector and weight vector, and the unit whose weight vector is at the smallest Euclidean distance from the input vector is chosen as the winner. The next method uses the dot product of the input vector and weight vector. The dot product between the input vector and weight vector is nothing but the net inputs calculated for the corresponding cluster units. The unit with the largest dot product is chosen as the winner and the weight updation is performed over it because the one with largest dot product corresponds to the smallest angle between the input and weight vectors, if both are of unit length. Both the methods can be applied for vectors of unit length. But generally, to avoid normalization of the input and weight vectors, the square of the Euclidean distance may be used.

5.2 Fixed Weight Competitive Nets

These competitive nets are those where the weights remain fixed, even during training process. The idea of competition is used among neurons for enhancement of contrast in their activation functions. In this section, three nets – Maxnet, Mexican hat and Hamming net – are discussed in detail.

5.2.1 Maxnet

In 1987, Lippmann developed the Maxnet which is an example for a neural net based on competition. The Maxnet serves as a subnet for picking the node whose input is larger. All the nodes present in this subnet are fully interconnected and there exist symmetrical weights in all these weighted interconnections. As such, there is no specific algorithm to train Maxnet; the weights are fixed in this case.

5.2.1.1 Architecture of Maxnet

The architecture of Maxnet is shown in Figure 5-1, where fixed symmetrical weights are present over the weighted interconnections. The weights between the neurons are inhibitory and fixed. The Maxnet with this structure can be used as a subnet to select a particular node whose net input is the largest.

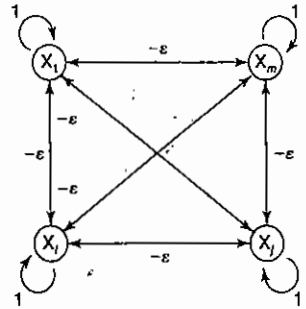


Figure 5-1 Maxnet structure.

5.2.1.2 Testing/Application Algorithm of Maxnet

The Maxnet uses the following activation function:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leqslant 0 \end{cases}$$

The testing algorithm is as follows:

Step 0: Initial weights and initial activations are set. The weight is set as $[0 < \varepsilon < 1/m]$, where "m" is the total number of nodes. Let

$$x_j(0) = \text{input to the node } X_j$$

and

$$w_{ij} = \begin{cases} 1 & \text{if } i = j \\ -\varepsilon & \text{if } i \neq j \end{cases}$$

Step 1: Perform Steps 2–4, when stopping condition is false.

Step 2: Update the activations of each node. For $j = 1$ to m,

$$x_j(\text{new}) = f\left[x_j(\text{old}) - \varepsilon \sum_{i \neq j} x_k(\text{old})\right]$$

Step 3: Save the activations obtained for use in the next iteration. For $j = 1$ to m,

$$x_j(\text{old}) = x_j(\text{new})$$

Step 4: Finally, test the stopping condition for convergence of the network. The following is the stopping condition: If more than one node has a nonzero activation, continue; else stop.

In this algorithm, the input given to the function $f(\cdot)$ is simply the total input to node X_j from all others, including its own input.

5.2.2 Mexican Hat Net

In 1989, Kohonen developed the Mexican hat network which is a more generalized contrast enhancement network compared to the earlier Maxnet. There exist several "cooperative neighbors" (neurons in close proximity) to which every other neuron is connected by excitatory links. Also each neuron is connected over inhibitory weights to a number of "competitive neighbors" (neurons present farther away). There are several other farther neurons to which the connections between the neurons are not established. Here, in addition to the connections within a particular layer of neural net, the neurons also receive some other external signals. This interconnection pattern is repeated for several other neurons in the layer.

5.2.2.1 Architecture

The architecture of Mexican hat is shown in Figure 5-2, with the interconnection pattern for node X_i. The neurons here are arranged in linear order; having positive connections between X_i and near neighboring units, and negative connections between X_i and farther away neighboring units. The positive connection region is called region of cooperation and the negative connection region is called region of competition. The size of these regions depends on the relative magnitudes existing between the positive and negative weights and also on the topology of regions such as linear, rectangular, hexagonal grids, etc. In Mexican Hat, there exist two symmetric regions around each individual neuron.

The individual neuron in Figure 5-2 is denoted by X_i. This neuron is surrounded by other neurons X_{i+1}, X_{i-1}, X_{i+2}, X_{i-2}, ... The nearest neighbors to the individual neuron X_i are X_{i+1}, X_{i-1}, X_{i+2}, and X_{i-2}. Hence, the weights associated with these are considered to be positive and are denoted by w_1 and w_2. The farthest neighbors to the individual neuron X_i are taken as X_{i+3} and X_{i-3}, the weights associated with these are negative and are denoted by w_3. It can be seen that X_{i+4} and X_{i-4} are not connected to the individual neuron X_i, and therefore no weighted interconnections exist between these connections. To make it easier, the units present within a radius of 2 [query for unit] to the unit X_i are connected with positive weights, the units within radius 3 are connected with negative weights and the units present farther away from radius 3 are not connected in any manner to the neuron X_i.

5.2.2.2 Flowchart

The flowchart for Mexican hat is shown in Figure 5-3. This clearly depicts the flow of the process performed in Mexican hat network.

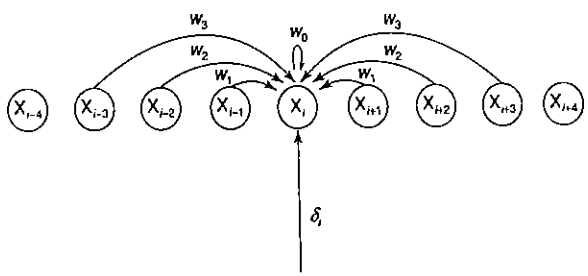


Figure 5-2 Structure of Mexican hat.

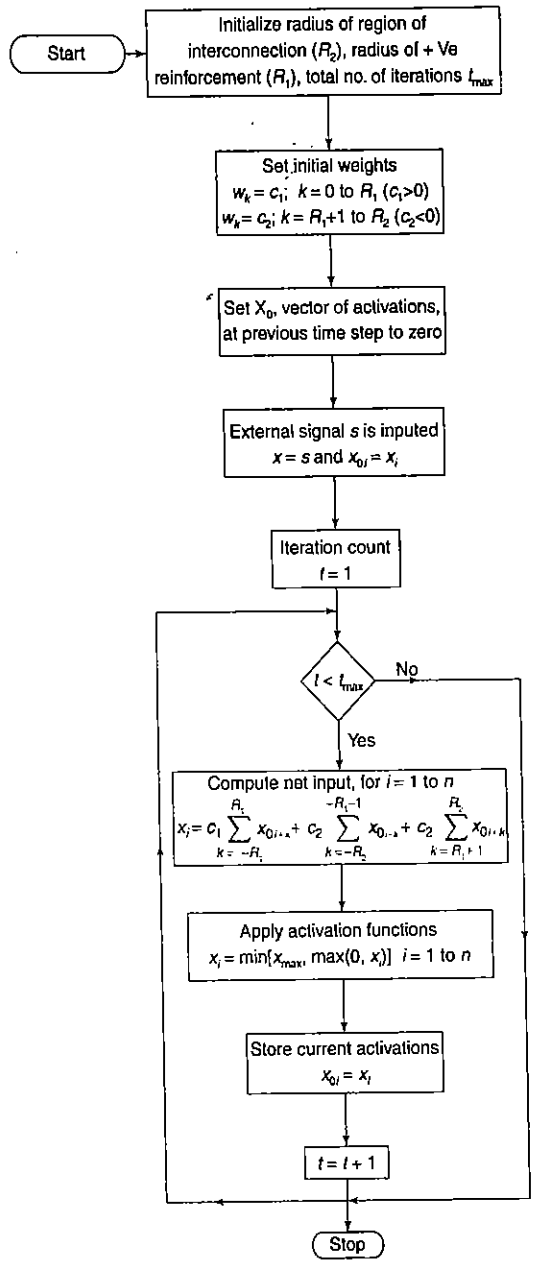


Figure 5-3 Flowchart of Mexican hat.

5.2.2.3 *Algorithm*

The various parameters used in the training algorithm are as shown below.

R_2 = radius of regions of interconnections

X_{i+k} and X_{i-k} are connected to the individual units X_i for $k = 1$ to R_2.

R_1 = radius of region with positive reinforcement ($R_1 < R_2$)

W_k = weight between X_i and the units X_{i+k} and X_{i-k}

$$0 \leqslant k \leqslant R_1, \quad w_k = \text{positive}$$
$$R_1 \leqslant k \leqslant R_2, \quad w_k = \text{negative}$$

s = external input signal

x = vector of activation

x_0 = vector of activations at previous time step

t_{max} = total number of iterations of contrast enhancement.

Here the iteration is started only with the incoming of the external signal presented to the network.

Step 0: The parameters R_1, R_2, t_{max} are initialized accordingly. Initialize weights as

$$w_k = c_1 \quad \text{for } k = 0, \ldots, R_1 \quad (\text{where } c_1 > 0)$$
$$w_k = c_2 \quad \text{for } k = R_1 + 1, \ldots, R_2 \quad (\text{where } c_2 < 0)$$

Initialize $x_0 = 0$.

Step 1: Input the external signal s:

$$x = s$$

The activations occurring are saved in array x_0. For $i = 1$ to n,

$$x_{0i} = x_i$$

Once activations are stored, set iteration counter $t = 1$.

Step 2: When t is less than t_{max}, perform Steps 3–7.

Step 3: Calculate net input. For $i = 1$ to n,

$$x_i = c_1 \sum_{k=-R_1}^{R_1} x_{0_{i+k}} + c_2 \sum_{k=-R_2}^{-R_1-1} x_{0_{i+k}} + c_2 \sum_{k=R_1+1}^{R_2} x_{0_{i+k}}$$

Step 4: Apply the activation function. For $i = 1$ to n,

$$x_i = \min[x_{max}, \max(0, x_i)]$$

Step 5: Save the current activations in x_0, i.e., for $i = 1$ to n,

$$x_{0i} = x_i$$

Step 6: Increment the iteration counter:

$$t = t + 1$$

Step 7: Test for stopping condition. The following is the stopping condition:

If $t < t_{max}$, then continue

Else stop

The positive reinforcement here has the capacity to increase the activation of units with larger initial activations and the negative reinforcement has the capacity to reduce the activation of units with smaller initial activations. The activation function used here for unit X_i at a particular time instant "t" is given by

$$x_i(t) = f\left[s_i(t) + \sum_k w_k x_{i+k} + k(t-1)\right]$$

The terms present within the summation symbol are the weighted signals that arrived from other units at the previous time step.

5.2.3 Hamming Network

The Hamming network selects stored classes, which are at a maximum Hamming distance (H) from the noisy vector presented at the input (Lippmann, 1987). The vectors involved in this case are all binary and bipolar. Hamming network is a maximum likelihood classifier that determines which of several exemplar vectors (the weight vector for an output unit in a clustering net is exemplar vector or code book vector for the pattern of inputs, which the net has placed on that cluster unit) is most similar to an input vector (represented as an n-tuple). The weights of the net are determined by the exemplar vectors. The difference between the total number of components and the Hamming distance between the vectors gives the measure of similarity between the input vector and stored exemplar vectors. It is already discussed in Chapter 4 that the Hamming distance between the two vectors is the number of components in which the vectors differ.

Consider two bipolar vectors x and y, we use a relation

$$x \cdot y = a - d$$

where a is the number of components in which the vectors agree, d the number of components in which the vectors disagree. The value "$a - d$" is the Hamming distance existing between two vectors. Since, the total number of components is n, we have,

$$n = a + d$$
$$\text{i.e.,} \quad d = n - a$$

On simplification, we get

$$x \cdot y = a - d$$
$$x \cdot y = a - (n - a)$$
$$x \cdot y = 2a - n$$
$$2a = x \cdot y + n$$
$$a = \frac{1}{2}(x \cdot y) + \frac{1}{2}(n)$$

From the above equation, it is clearly understood that the weights can be set to one-half the exemplar vector and bias can be set initially to $n/2$. By calculating the unit with the largest net input, the net is able to locate a particular unit that is closest to the exemplar. The unit with the largest net input is obtained by the Hamming net using Maxnet as its subnet.

5.2.3.1 *Architecture*

The architecture of Hamming network is shown in Figure 5-4. The Hamming network consists of two layers. The first layer computes the difference between the total number of components and Hamming distance between the input vector x and the stored pattern of vectors in the feed-forward path. The efficient response in this layer of a neuron is the indication of the minimum Hamming distance value between the input and the category, which this neuron represents. The second layer of the Hamming network is composed of Maxnet (used as a subnet) or a winner-take-all network which is a recurrent network. The Maxnet is found to suppress the values at Maxnet output nodes except the initially maximum output node of the first layer.

The function of Maxnet is to enhance the initial dominant response of the node and suppress others. Since Maxnet possesses recurrent processing, the jth node is found to respond positively while the response of all the remaining nodes decays to zero. This result needs a positive self-feedback connection with itself and a negative lateral inhibition connection.

5.2.3.2 *Testing Algorithm*

The given bipolar input vector is x and for a given set of "m" bipolar exemplar vectors say $e(1), \ldots, e(j), \ldots, e(m)$, the Hamming network is used to determine the exemplar vector that is closest to the input

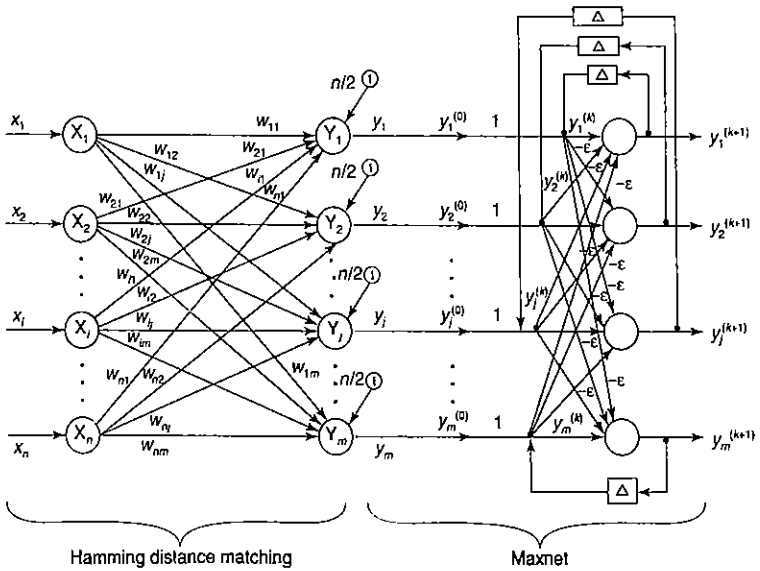


Figure 5-4 Structure of Hamming network.

vector x. The net input entering unit Y_j gives the measure of the similarity between the input vector and exemplar vector. The parameters used here are the following:

n = number of input units (number of components of *input–output* vector)

m = number of output units (number of components of exemplar vector)

$e(j)$ = jth exemplar vector, i.e.,

$$e(j) = [e_1(j), \ldots, e_i(j), \ldots, e_n(j)]$$

The testing algorithm for the Hamming Net is as follows:

Step 0: Initialize the weights. For $i = 1$ to n and $j = 1$ to m,

$$w_{ij} = \frac{e_i(j)}{2}$$

Initialize the bias for storing the "m" exemplar vectors. For $j = 1$ to m,

$$b_j = \frac{n}{2}$$

Step 1: Perform Steps 2–4 for each input vector x.

Step 2: Calculate the net input to each unit Y_j, i.e.,

$$y_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

Step 3: Initialize the activations for Maxnet, i.e.,

$$y_j(0) = y_{inj}, \quad j = 1 \text{ to } m$$

Step 4: Maxnet is found to iterate for finding the exemplar that best matches the input patterns.

The Hamming network is found to retrieve only the closest class index and not the entire vector. Hence, the Hamming network is a classifier, rather than being an associative memory. The Hamming network can be modified to be an associative memory by just adding an extra layer over the Maxnet, such that the winner unit, $y_i(k+1)$, present in the Maxnet may trigger a corresponding stored weight vector. Such an associative memory network can be called a Hamming memory network.

5.3 Kohonen Self-Organizing Feature Maps

5.3.1 Theory

Feature mapping is a process which converts the patterns of arbitrary dimensionality into a response of one- or two-dimensional arrays of neurons, i.e., it converts a wide pattern space into a typical feature space. The network performing such a mapping is called feature map. Apart from its capability to reduce the higher dimensionality, it has to preserve the neighborhood relations of the input patterns, i.e., it has to obtain a

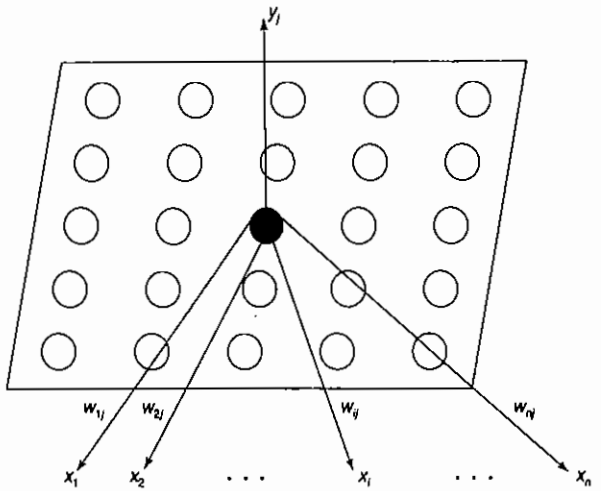


Figure 5-5 One-dimensional feature mapping network.

topology preserving map. For obtaining such feature maps, it is required to find a self-organizing neural array which consists of neurons arranged in a one-dimensional array or a two-dimensional array. To depict this, a typical network structure where each component of the input vector x is connected to each of the nodes is shown in Figure 5-5.

On the other hand, if the input vector is two-dimensional, the inputs, say $x(a, b)$, can arrange themselves in a two-dimensional array defining the input space (a, b) as in Figure 5-6. Here, the two layers are fully connected.

The topological preserving property is observed in the brain, but not found in any other artificial neural network. Here, there are m output cluster units arranged in a one- or two-dimensional array and the input signals are n-tuples. The cluster (output) units' weight vector serves as an exemplar of the input pattern that is associated with that cluster. At the time of self-organization, the weight vector of the cluster unit which matches the input pattern very closely is chosen as the winner unit. The closeness of weight vector of cluster unit to the input pattern may be based on the square of the minimum Euclidean distance. The weights are updated for the winning unit and its neighboring units. It should be noted that the weight vectors of the neighboring units are not close to the input pattern and the connective weights do not multiply the signal sent from the input units to the cluster units until dot product measure of similarity is being used.

5.3.2 Architecture

Consider a linear array of cluster units as in Figure 5-7. The neighborhoods of the units designated by "o" of radii $N_i(k_1)$, $N_i(k_2)$ and $N_i(k_3)$, $k_1 > k_2 > k_3$, where $k_1 = 2, k_2 = 1, k_3 = 0$.

For a rectangular grid, a neighborhood (N_i) of radii k_1, k_2 and k_3 is shown in Figure 5-8 and for a hexagonal grid the neighborhood is shown in Figure 5-9. In all the three cases (Figures 5-7–5-9), the unit with

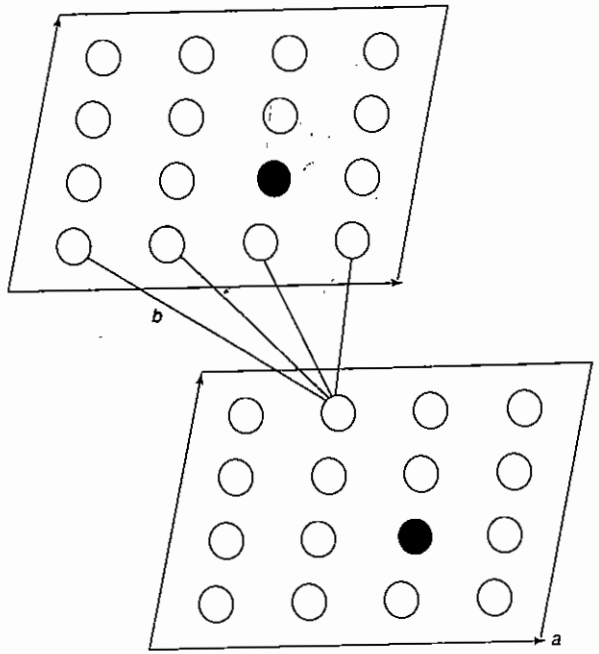


Figure 5-6 Two-dimensional feature mapping network.

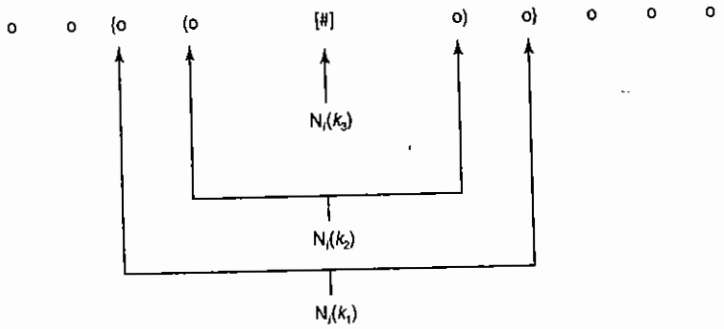


Figure 5-7 Linear array of cluster units.

"#" symbol is the winning unit and the other units are indicated by "o." In both rectangular and hexagonal grids, $k_1 > k_2 > k_3$, where $k_1 = 2, k_2 = 1, k_3 = 0$.

For rectangular grid, each unit has eight nearest neighbors but there are only six neighbors for each unit in the case of a hexagonal grid. Missing neighborhoods may just be ignored. A typical architecture of Kohonen self-organizing feature map (KSOFM) is shown in Figure 5-10.

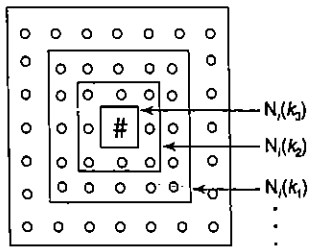


Figure 5-8 Rectangular grid.

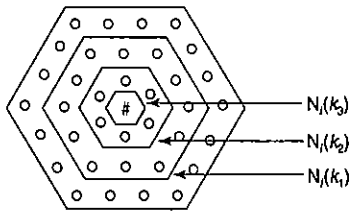


Figure 5-9 Hexagonal grid.

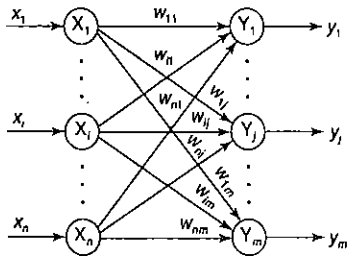


Figure 5-10 Kohonen self-organizing feature map architecture.

5.3.3 Flowchart

The flowchart for KSOFM is shown in Figure 5-11, which indicates the flow of training process. The process is continued for particular number of epochs or till the learning rate reduces to a very small rate.

The architecture consists of two layers: input layer and output layer (cluster). There are "n" units in the input layer and "m" units in the output layer. Basically, here the winner unit is identified by using either dot product or Euclidean distance method and the weight updation using Kohonen learning rules is performed over the winning cluster unit.

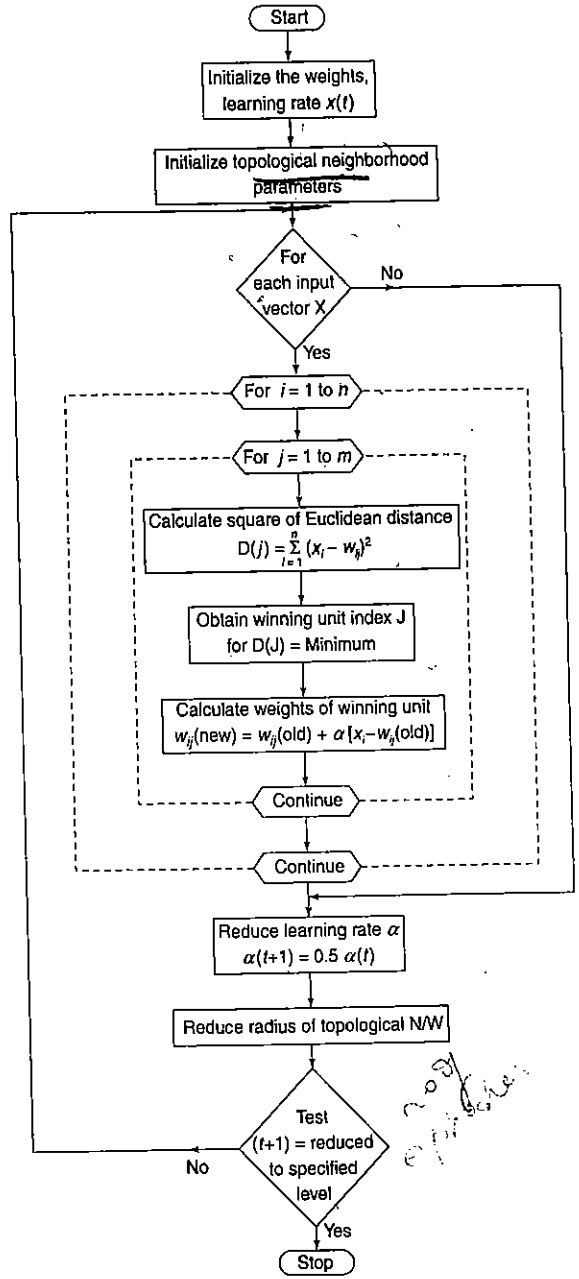


Figure 5-11 Flowchart for training process of KSOFM.

5.3.4 Training Algorithm

The steps involved in the training algorithm are as shown below.

Step 0:
- Initialize the weights w_{ij}: Random values may be assumed. They can be chosen as the same range of values as the components of the input vector. If information related to distribution of clusters is known, the initial weights can be taken to reflect that prior knowledge.
- Set topological neighborhood parameters: As clustering progresses, the radius of the neighborhood decreases.
- Initialize the learning rate α: It should be a slowly decreasing function of time.

Step 1: Perform Steps 2–8 when stopping condition is false.

Step 2: Perform Steps 3–5 for each input vector x.

Step 3: Compute the square of the Euclidean distance, i.e., for each $j = 1$ to m,

$$D(j) = \sum_{i=1}^{n} \sum_{j=1}^{m} (x_i - w_{ij})^2$$

Step 4: Find the winning unit index J, so that D(J) is minimum. (In Steps 3 and 4, dot product method can also be used to find the winner, which is basically the calculation of net input, and the winner will be the one with the largest dot product.)

Step 5: For all units j within a specific neighborhood of J and for all i, calculate the new weights:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha [x_i - w_{ij}(\text{old})]$$

or

$$w_{ij}(\text{new}) = (1 - \alpha) w_{ij}(\text{old}) + \alpha x_i$$

Step 6: Update the learning rate α using the formula $\alpha(t + 1) = 0.5\alpha(t)$.

Step 7: Reduce radius of topological neighborhood at specified time intervals.

Step 8: Test for stopping condition of the network.

Thus using this training algorithm, an efficient training can be performed for an unsupervised learning network.

5.3.5 Kohonen Self-Organizing Motor Map

The extension of Kohonen feature map for a multilayer network involves the addition of an association layer to the output of the self-organizing feature map layer. The output node is found to associate the desired output values with certain input vectors. This type of architecture is called as Kohonen self-organizing motor map (KSOMM; Ritter, 1992) and layer that is added is called a motor map in which the movement commands are being mapped into two-dimensional locations of excitation. The architecture of KSOMM is shown in Figure 5-12. Here, the feature map is a hidden layer and this acts as a competitive network which classifies the input vectors. The feature map is trained as discussed in Section 5.3.3. The motor map formation is based on the learning of a control task. The motor map learning may be either supervised or unsupervised learning and can be performed by delta learning rule or outstar learning rule (to be discussed later). The motor map learning is an extension of Kohonen's original learning algorithm.

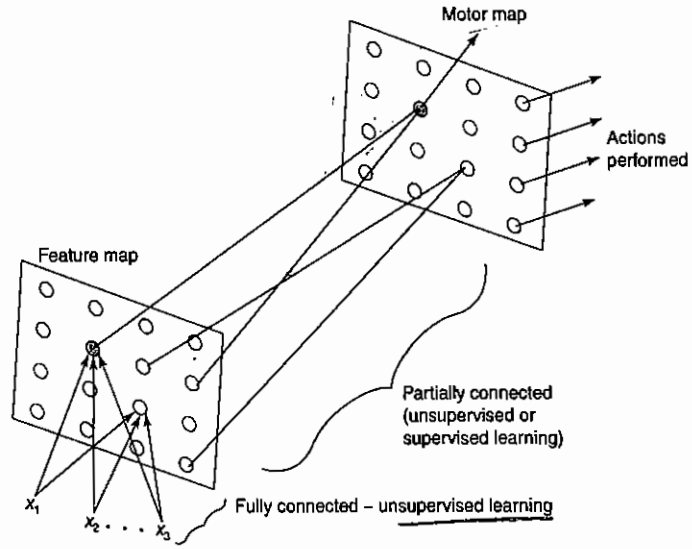


Figure 5-12 Architecture of Kohonen self-organizing motor map.

5.4 Learning Vector Quantization

5.4.1 Theory

Learning vector quantization (LVQ) is a process of classifying the patterns, wherein each output unit represents a particular class. Here, for each class several units should be used. The output unit weight vector is called the reference vector or code book vector for the class which the unit represents. This is a special case of competitive net, which uses supervised learning methodology. During training, the output units are found to be positioned to approximate the decision surfaces of the existing Bayesian classifier. Here, the set of training patterns with known classifications is given to the network, along with an initial distribution of the reference vectors. When the training process is complete, an LVQ net is found to classify an input vector by assigning it to the same class as that of the output unit, which has its weight vector very close to the input vector. Thus, LVQ is a classifier paradigm that adjusts the boundaries between categories to minimize existing misclassification. LVQ is used for optical character recognition, converting speech into phonemes and other applications as well. LVQ net may resemble KSOFM net. Unlike LVQ, KSOFM output nodes do not correspond to the known classes but rather correspond to unknown clusters that the KSOFM finds in the data autonomously.

5.4.2 Architecture

Figure 5-13 shows the architecture of LVQ, which is almost the same as that of KSOFM, with the difference being that in the case of LVQ, the topological structure at the output unit is not being considered. Here, each output unit has knowledge about what a known class represents.

From Figure 5-13 it can be noticed that there exists input layer with "n" units and output layer with "m" units. The layers are found to be fully interconnected with weighted linkage acting over the links.

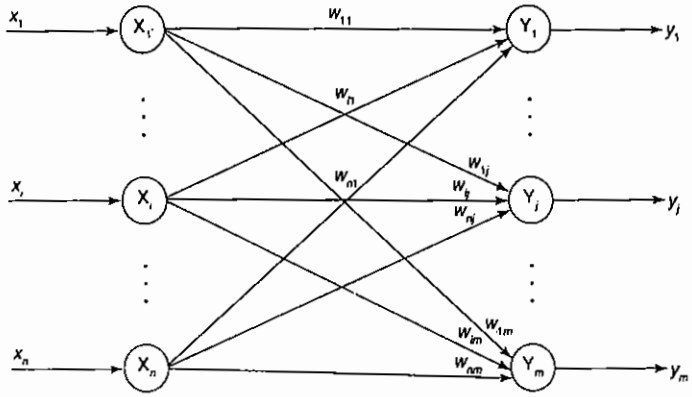


Figure 5-13 Architecture of LVQ.

5.4.3 Flowchart

The parameters used for the training process of a LVQ include the following:

x = training vector $(x_1, \ldots, x_i, \ldots, x_n)$

T = category or class for the training vector x

w_j = weight vector for jth output unit $(w_{1j}, \ldots, w_{ij}, \ldots, w_{nj})$

c_j = cluster or class or category associated with jth output unit.

The Euclidean distance of jth output unit is $D(j) = \sum (x_i - w_{ij})^2$. The flowchart indicating the flow of training process is shown in Figure 5-14.

5.4.4 Training Algorithm

In case of training, a set of training input vectors with a known classification is provided with some initial distribution of reference vector. Here, each output unit will have a known class. The objective of the algorithm is to find the output unit that is closest to the input vector.

Step 0: Initialize the reference vectors. This can be done using the following steps.

- From the given set of training vectors, take the first "m" (number of clusters) training vectors and use them as weight vectors, the remaining vectors can be used for training.
- Assign the initial weights and classifications randomly.
- K-means clustering method.

Set initial learning rate α.

Step 1: Perform Steps 2–6 if the stopping condition is false.

Step 2: Perform Steps 3–4 for each training input vector x.

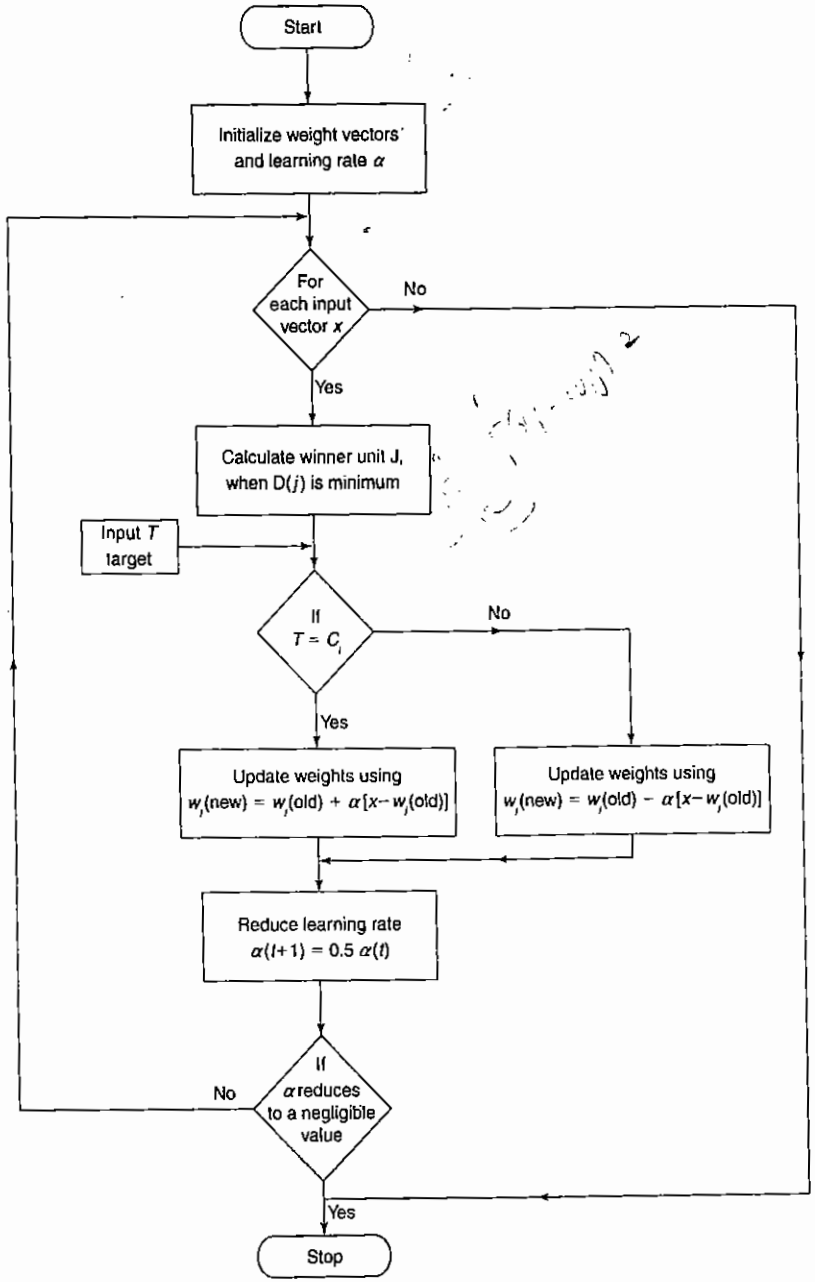


Figure 5-14 Flowchart for LVQ.

Step 3: Calculate the Euclidean distance; for $i = 1$ to n, $j = 1$ to m,

$$D(j) = \sum_{i=1}^{n}\sum_{j=1}^{m}(x_i - w_{ij})^2$$

Find the winning unit index J, when D(J) is minimum.

Step 4: Update the weights on the winning unit, w_J using the following conditions.

If $T = q$, then $w_J(\text{new}) = w_J(\text{old}) + \alpha\,[x - w_J(\text{old})]$

If $T \neq q$, then $w_J(\text{new}) = w_J(\text{old}) - \alpha\,[x - w_J(\text{old})]$

Step 5: Reduce the learning rate α.

Step 6: Test for the stopping condition of the training process. (The stopping conditions may be fixed number of epochs or if learning rate has reduced to a negligible value.)

5.4.5 Variants

There exists several variants of LVQ net proposed by Kohonen. These include LVQ2, LVQ2.1 and LVQ3. In the LVQ algorithm, only the reference vector that is closest to the input vector is updated. The movement it moves is based on whether or not the winning vector belongs to the same class as the input vector. In the developed versions of LVQ, two vectors called winner vector and runner-up vector learns if several conditions are satisfied. Here two distances have to be calculated. Learning takes place only if the input is approximately the same distance from winner and runner. One distance is from winner to input layer and the other is from runner to input layer.

5.4.5.1 LVQ 2

The conditions over which both vectors are modified in the case of LVQ 2 are the following.

1. The winner and the runner-up unit belong to different classes.
2. The runner-up vector is of the same class as the input vector.
3. The distances between the input vector and winner and between the input vector and runner-up are almost equal to each other.

If x is the current input vector, y_1 the reference vector closer to x (winner), y_2 the reference vector next closer to x (runner-up), d_1 the distance from x to y_1, d_2 the distance from x to y_2, then the conditions for the updation of the reference vector can be defined as follows:

$$\frac{d_1}{d_2} > (1-\varepsilon)$$

and

$$\frac{d_2}{d_1} < (1+\varepsilon)$$

where the value of ε is based on the number of training samples. The weight updation formulas in this case are given by

$$y_1(t+1) = y_1(t) - \alpha\,(t)[x(t) - y_1(t)] \text{ (belongs to different class)}$$

$$y_2(t+1) = y_2(t) + \alpha\,(t)[x(t) - y_2(t)] \text{ (belongs to same class)}$$

5.4.5.2 LVQ 2.1

In LVQ 2.1, the two closest reference vectors y_{1c} and y_{2c} are taken. Here updating is done on the basis of the requirements that (a) y_{1c} belongs to the correct class for the given input vector x and (b) y_{2c} does not belong to the same class as x. LVQ 2.1 does not distinguish whether the closest vector is that representing the correct class or incorrect class for the given input. The conditions for this case are given by

$$\min\left[\frac{d_{1c}}{d_{2c}}, \frac{d_{2c}}{d_{1c}}\right] > (1-\varepsilon)$$

and

$$\max\left[\frac{d_{1c}}{d_{2c}}, \frac{d_{2c}}{d_{1c}}\right] < (1+\varepsilon)$$

Here, it is not sure whether x is closer to y_{1c} or to y_{2c}. When the above conditions are met, the following weight updation formulas are used. If the reference vector belongs to the same class as input vector, then

$$y_{1c}(t+1) = y_{1c}(t) + \alpha\,(t)[x(t) - y_{1c}(t)]$$

else

$$y_{2c}(t+1) = y_{2c}(t) - \alpha\,(t)[x(t) - y_{2c}(t)]$$

5.4.5.3 LVQ 3

In LVQ 3, the two closest vectors are allowed to learn as long as the input vector satisfies the condition (take $\varepsilon = 0.2$)

$$\min\left[\frac{d_{1c}}{d_{2c}}, \frac{d_{2c}}{d_{1c}}\right] > (1-\varepsilon)(1+\varepsilon)$$

The weight updations are done in a similar manner as in LVQ 2.1 if one of the two closest vectors, y_{1c}, belongs to the same class as the input vector x and the other vector, y_{2c}, belongs to a different class. LVQ 3 extends this training algorithm to provide training if x, y_{1c} and y_{2c} belong to the same class. The weight updates, here, are given by the equation

$$y_c(t+1) = y_c(t) + \beta(t)[x(t) - y_c(t)]$$

Replace y_c with y_{1c} or y_{2c}, as the case may be. The learning rate $\beta(t)$ is a multiple of the learning rate $\alpha(t)$ that is used if y_{1c} and y_{2c} belong to different classes, i.e.,

$$\beta(t) = q\,\alpha(t)$$

where q is between 0.1 and 0.5

5.5 Counterpropagation Networks

5.5.1 Theory

Counterpropagation networks were proposed by Hecht Nielsen in 1987. They are multilayer networks based on the combinations of the input, output and clustering layers. The applications of counterpropagation nets are data compression, function approximation and pattern association. The counterpropagation network is basically constructed from an instar–outstar model. This model is a three-layer neural network that performs input–output data mapping, producing an output vector y in response to an input vector x, on the basis of competitive learning. The three layers in an instar–outstar model are the input layer, the hidden (competitive)

layer and the output layer. The connections between the input layer and the competitive layer are the instar structure, and the connections existing between the competitive layer and the output layer are the outstar structure. The competitive layer is going to be a winner-take-all network or a Maxnet with lateral feedback connections. There exists no lateral connection within the input layer and the output layer. The connections between the layers are fully connected.

A counterpropagation net is an approximation of its training input vector pairs by adaptively constructing a look-up-table. By this method, several data points can be compressed to a more manageable number of look-up-table entries. The accuracy of the function approximation and data compression is based on the number of entries in the look-up-table, which equals the number of units in the cluster layer of the net.

There are two stages involved in the training process of a counterpropagation net. The input vectors are clustered in the first stage. Originally, it is assumed that there is no topology included in the counterpropagation network. However, on the inclusion of a linear topology, the performance of the net can be improved. The clusters are formed using Euclidean distance method or dot product method. In the second stage of training, the weights from the cluster layer units to the output units are tuned to obtain the desired response. There are two types of counterpropagation nets: (i) Full counterpropagation net and (ii) forward-only counterpropagation net.

5.5.2 Full Counterpropagation Net

Full counterpropagation net (full CPN) efficiently represents a large number of vector pairs $x{:}y$ by adaptively constructing a look-up-table. The approximation here is $x^*{:}y^*$, which is based on the vector pairs $x{:}y$, possibly with some distorted or missing elements in either vector or both vectors. The network is defined to approximate a continuous function f, defined on a compact set A. The full CPN works best if the inverse function f^{-1} exists and is continuous. The vectors x and y propagate through the network in a counterflow manner to yield output vectors x^* and y^*, which are the approximations of x and y, respectively. During competition, the winner can be determined either by Euclidean distance or by dot product method. In case of dot product method, the one with the largest net input is the winner. Whenever vectors are to be compared using the dot product metric, they should be normalized. Even though the normalization can be performed without loss of information by adding an extra component, yet to avoid the complexity Euclidean distance method can be used. On the basis of this, direct comparison can be made between the full CPN and forward-only CPN.

For continuous function, the CPN is as efficient as the back-propagation net; it is a universal continuous function approximator. In case of CPN, the number of hidden nodes required to achieve a particular level of accuracy is greater than the number required by the back-propagation network. The greatest appeal of CPN is its speed of learning. Compared to various mapping networks, it requires only fewer steps of training to achieve best performance. This is common for any hybrid learning method that combines unsupervised learning (e.g., instar learning) and supervised learning (e.g., outstar learning).

As already discussed, the training of CPN occurs in two phases. In the input phase, the units in the cluster layer and input layer are found to be active. In CPN, no topology is assumed for the cluster layer units; only the winning units are allowed to learn. The weight updation learning rule on the winning cluster units is

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha\,[x_i - v_{ij}(\text{old})], \quad i = 1 \text{ to } n$$
$$w_{kj}(\text{new}) = w_{kj}(\text{old}) + \beta\,[y_k - w_{kj}(\text{old})], \quad k = 1 \text{ to } m$$

The above is standard Kohonen learning which consists of competition among the units and selection of winner unit. The weight updation is performed for the winning unit.

In the second phase of training, only the winner unit J remains active in the cluster layer. The weights between the winning cluster unit J and the output units are adjusted so that the vector of activations of the units in the Y-output layer is Y* which is an approximation to the input vector y and X* which is an approximation to the input vector x. The weight updations for the units in the Y-output and X-output layers are

$$u_{jk}(\text{new}) = u_{jk}(\text{old}) + a[y_k - u_{jk}(\text{old})], \quad k = 1 \text{ to } m$$
$$t_{ji}(\text{new}) = t_{ji}(\text{old}) + b[x_i - t_{ji}(\text{old})], \quad i = 1 \text{ to } n$$

This is Grossberg learning, a more general case of outstar learning. Outstar learning is found to occur for all units in a particular layer; there exists no competition among those units. The form of weight updation is similar for Kohonen learning and Grossberg learning. The learning rule for the output layers can also be viewed as delta learning rule. The weight change in all these cases is the product of the learning rate and the error. When tie occurs in the selection of winning unit, the unit with smallest index is chosen as the winner.

5.5.2.1 *Architecture*

The general structure of full CPN is shown in Figure 5-15. The complete architecture of full CPN is shown in Figure 5-16.

The four major components of the instar–outstar model are the input layer, the instar, the competitive layer and the outstar. For each node i in the input layer, there is an input value x_i. An instar responds maximally to the input vectors from a particular cluster. All the instars are grouped into a layer called the competitive layer. Each of the instar responds maximally to a group of input vectors in a different region of space. This layer of instars classifies any input vector because, for a given input, the winning instar with the strongest response identifies the region of space in which the input vector lies. Hence, it is necessary that the competitive layer single outs the winning instar by setting its output to a nonzero value and also suppressing the other outputs to zero. That is, it is a winner-take-all or a Maxnet-type network. An outstar model is found to have all the nodes in the output layer and a single node in the competitive layer. The outstar looks like the fan-out of a node. Figures 5-17 and 5-18 indicate the units that are active during each of the two phases of training a full CPN.

In the instar–outstar network model, the competitive layer participates in both the instar and outstar structures of the network. The function of these competitive instars is to recognize an input pattern through a winner-take-all competition. The winner activates a corresponding outstar which associates some desired output pattern with input pattern.

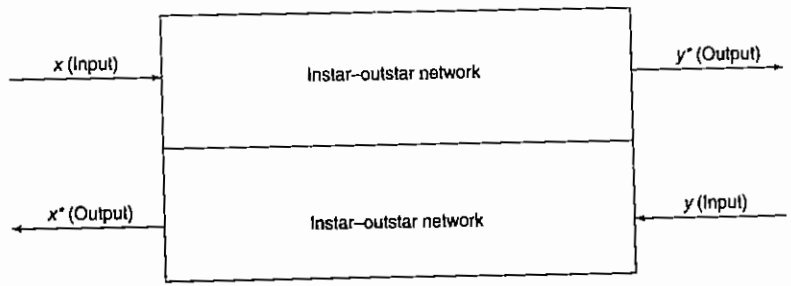


Figure 5-15 General structure of full CPN.

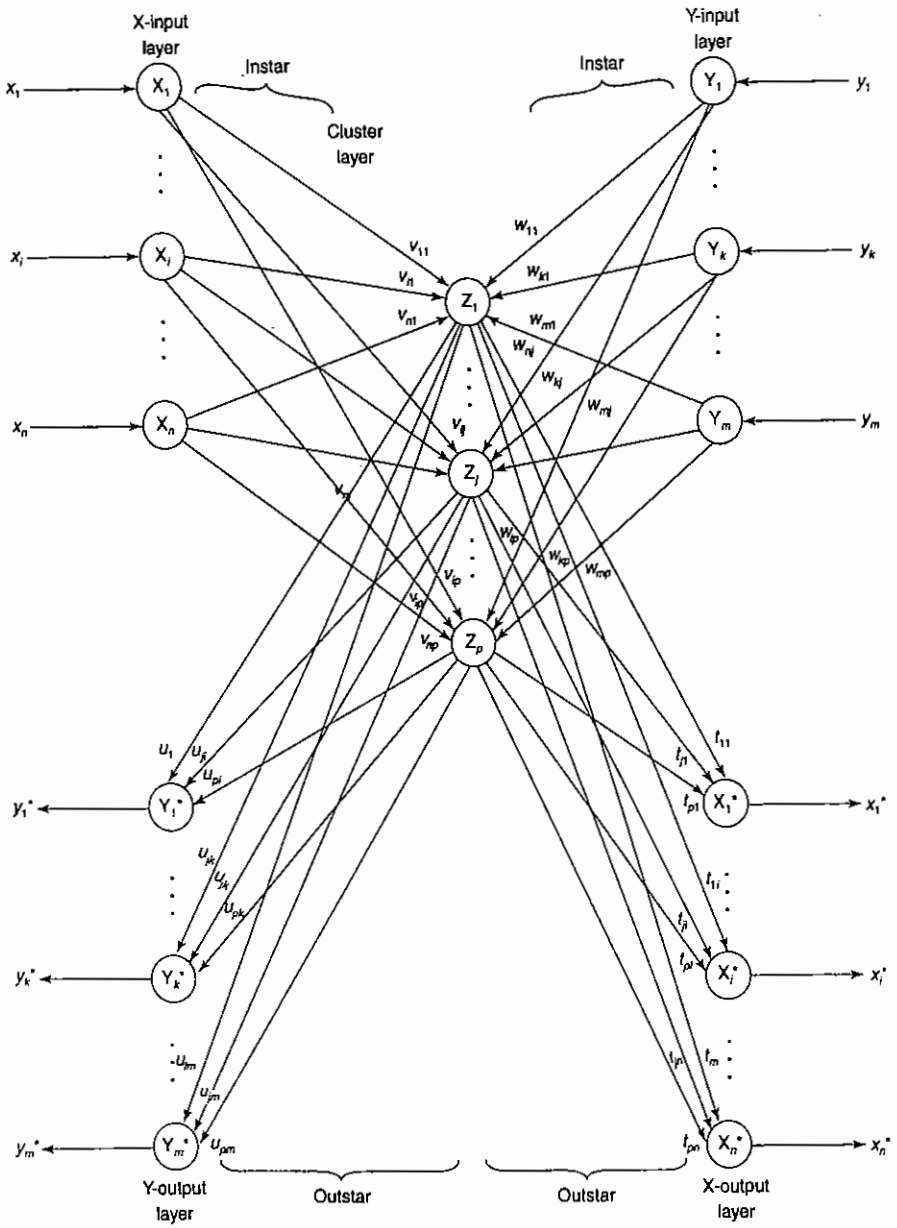


Figure 5-16 Architecture of full CPN.

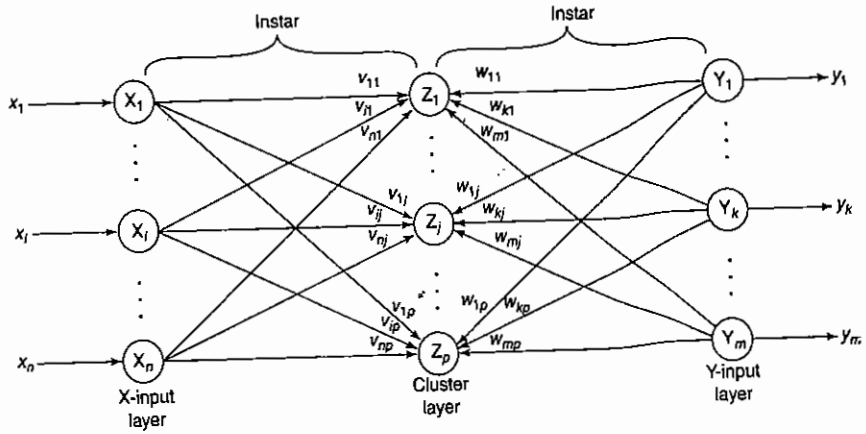


Figure 5-17 First phase of training of full CPN.

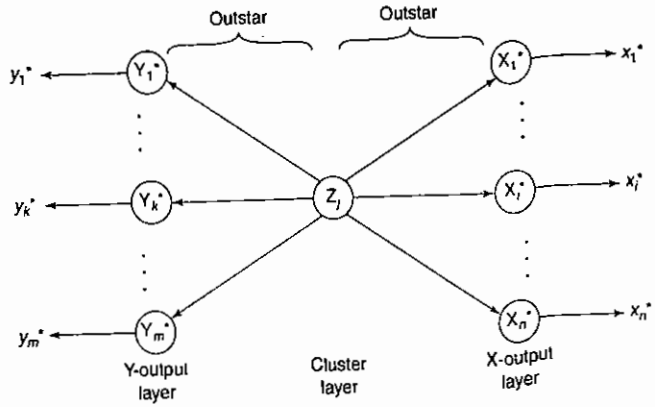


Figure 5-18 Second phase of training of full CPN.

5.5.2.2 *Flowchart*

The flowchart for the training process of full CPN is shown in Figure 5-19. The parameters used in the CPN are as follows:

x = input training vector $x = (x_1, \ldots, x_i, \ldots, x_n)$

y = target output corresponding to input x, $y = (y_1, \ldots, y_k, \ldots, y_m)$

z_j = the output of cluster layer unit z_j

v_{ij} = weight from X-input layer unit X_i to cluster layer unit z_j

w_{kj} = weight from Y-input layer unit Y_k to cluster layer unit z_j

u_{jk} = weight from cluster layer unit z_j to Y-output layer unit Y_k^*

t_{ji} = weight from cluster layer unit z_j to X-output layer unit X_i^*

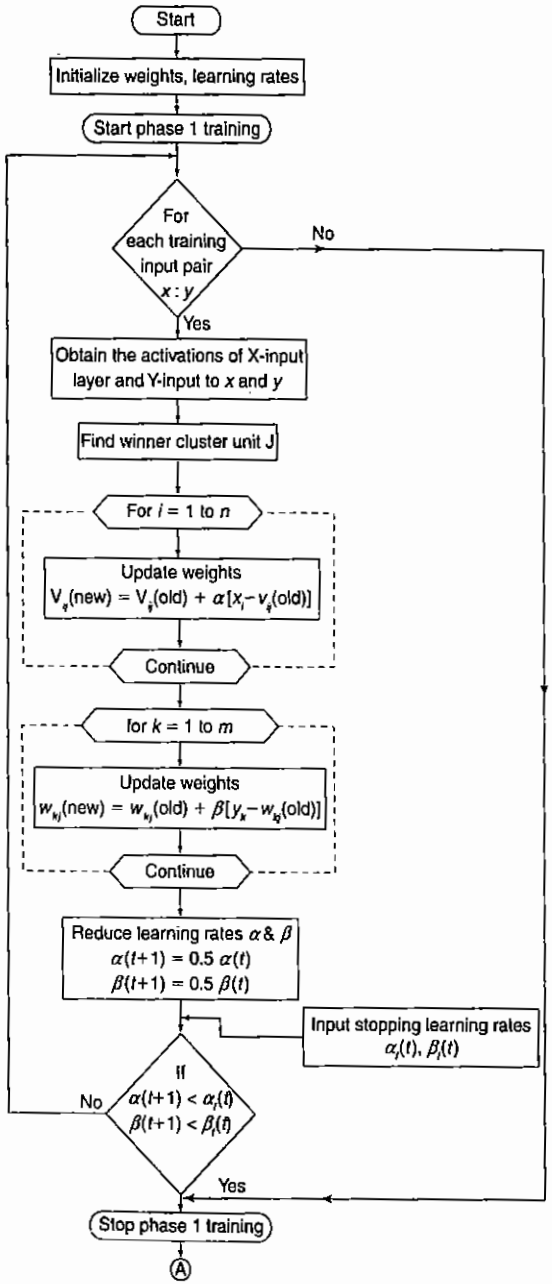


Figure 5-19 Flowchart for training of full CPN.

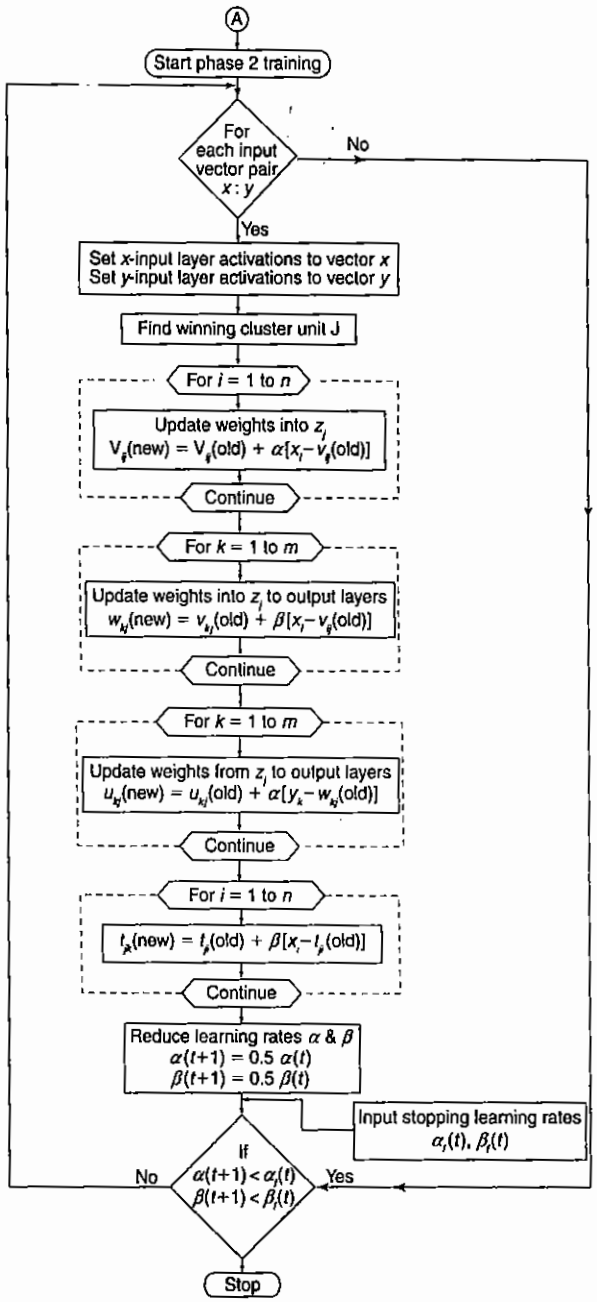


Figure 5-19 (*continued*).

X^* = calculated approximation to vector x

Y^* = calculated approximation to vector y

a, b = learning rates for weights out from cluster layer

α, β = learning rates for weights into cluster layer

The training phase is performed here in two stages. The stopping conditions here may be number of epochs to be reached. So the training process is performed until the number of epochs specified is completed. The reduction in learning rate can also be a stopping condition. The formula for reduction of learning rate is $\alpha(t+1) = 0.5\,\alpha(t)$, where $\alpha(t)$ is learning rate at time instant "t" and $\alpha(t+1)$ is learning rate of next epoch for a time instant "$t+1$".

5.5.2.3 *Training Algorithm*

The steps involved in the training process of a full CPN are given below.

Step 0: Set the initial weights and the initial learning rate.

Step 1: Perform Steps 2–7 if stopping condition is false for phase I training.

Step 2: For each of the training input vector pair $x : y$ presented, perform Steps 3–5.

Step 3: Make the X-input layer activations to vector X.
Make the Y-input layer activations to vector Y.

Step 4: Find the winning cluster unit.
If dot product method is used, find the cluster unit z_j with target net input: for $j = 1$ to p,

$$z_{inj} = \sum_{i=1}^{n} x_i v_{ij} + \sum_{k=1}^{m} y_k w_{kj}$$

If Euclidean distance method is used, find the cluster unit z_j whose squared distance from input vectors is the smallest:

$$D_j = \sum_{i=1}^{n} (x_i - v_{ij})^2 + \sum_{k=1}^{m} (y_k - w_{kj})^2$$

If there occurs a tie in case of selection of winner unit, the unit with the smallest index is the winner. Take the winner unit index as J.

Step 5: Update the weights over the calculated winner unit z_j.

$$\text{For } i = 1 \text{ to } n, \quad v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad w_{kj}(\text{new}) = w_{kj}(\text{old}) + \beta[y_k - w_{kj}(\text{old})]$$

Step 6: Reduce the learning rates.

$$\alpha(t+1) = 0.5\,\alpha(t); \quad \beta(t+1) = 0.5\,\beta(t)$$

Step 7: Test stopping condition for phase I training.

Step 8: Perform Steps 9–15 when stopping condition is false for phase II training.

Step 9: Perform Steps 10–13 for each training input pair $x : y$. Here α and β are small constant values.

Step 10: Make the X-input layer activations to vector x. Make the Y-input layer activations to vector y.

Step 11: Find the winning cluster unit (use formulas from Step 4). Take the winner unit index as J.

Step 12: Update the weights entering into unit z_j.

$$\text{For } i = 1 \text{ to } n, \quad v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad w_{kj}(\text{new}) = w_{kj}(\text{old}) + \beta[y_k - w_{kj}(\text{old})]$$

Step 13: Update the weights from unit z_j to the output layers.

$$\text{For } i = 1 \text{ to } n, \quad t_{ji}(\text{new}) = t_{ji}(\text{old}) + b[x_i - t_{ji}(\text{old})]$$
$$\text{For } k = 1 \text{ to } m, \quad u_{jk}(\text{new}) = u_{jk}(\text{old}) + a[y_k - u_{jk}(\text{old})]$$

Step 14: Reduce the learning rates a and b.

$$a(t+1) = 0.5\,a(t); \quad b(t+1) = 0.5\,b(t)$$

Step 15: Test stopping condition for phase II training.

If during training process initial weights are chosen appropriately, then after the completion of phase I of training, the cluster units will be uniformly distributed. When phase II of training is completed, the weights to the output units will be approximately the same as the weights into the cluster unit.

5.5.2.4 *Testing (Application) Algorithm*

A CPN once trained can be used for finding approximations X* and Y* to the input–output vector pair X and Y. The application algorithm for full CPN is as follows:

Step 0: Initialize the weights (from training algorithm).

Step 1: Perform Steps 2–4 for each input pair X : Y.

Step 2: Set X-input layer activations to vector X.
Set Y-input layer activations to vector Y.

Step 3: Find the cluster unit z_j that is closest to the input pair.

Step 4: Calculate approximations to x and y:

$$x_i^* = t_{ji}; \quad y_k^* = u_{jk}$$

One important variation of the CPN is operating it in an interpolation mode after the training has been completed. Here, more than one hidden mode is allowed to win the competition, i.e., we have first winner, second winner, third winner, fourth winner and so on, with nonzero output values. On making the total strength of these multiple winners normalized to 1, the total output will interpolate linearly among the individual vectors. To select which nodes to fire, we can choose all those with weight vectors within a certain radius of the input x. The interpolated approximations to x and y are then

$$x_i^* = \sum_j z_j t_{ji}; \quad y_k^* = \sum_j z_j u_{jk}$$

By using interpolation, the approximation accuracy is highly increased.

5.5.3 Forward-Only Counterpropagation Net

A simplified version of full CPN is the forward-only CPN. The approximation of the function $y = f(x)$ but not of $x = f(y)$ can be performed using forward-only CPN, i.e., it may be used if the mapping from x to y is well defined but mapping from y to x is not defined. In forward-only CPN only the x-vectors are used to form the clusters on the Kohonen units. Forward-only CPN uses only the x vectors to form the clusters on the Kohonen units during first phase of training.

In case of forward-only CPN, first input vectors are presented to the input units. The cluster layer units compete with each other using winner-take-all policy to learn the input vector. Once entire set of training vectors has been presented, there exist reduction in learning rate and the vectors are presented again, performing several iterations. First, the weights between the input layer and cluster layer are trained. Then the weights between the cluster layer and output layer are trained. This is a specific competitive network, with target known. Hence, when each input vector is presented to the input vector, its associated target vectors are presented to the output layer. The winning cluster unit sends its signal to the output layer. Thus each of the output unit has a computed signal (w_{jk}) and the target value (y_k). The difference between these values is calculated; based on this, the weights between the winning layer and output layer are updated.

The weight updation from input units to cluster units is done using the learning rule given below: For $i = 1$ to n,

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})] = (1-\alpha)v_{ij}(\text{old}) + \alpha x_i$$

The weight updation from cluster units to output units is done using following the learning rule: For $k = 1$ to m,

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + a[y_k - w_{jk}(\text{old})] = (1 - a)w_{jk}(\text{old}) + ay_k$$

The learning rule for weight updation from the cluster units to output units can be written in the form of delta rule when the activations of the cluster units (z_j) are included, and is given as

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + az_j[y_k - w_{jk}(\text{old})]$$

where

$$z_j = \begin{cases} 1 & \text{if } j = J \\ 0 & \text{if } j \neq J \end{cases}$$

This occurs when w_{jk} is interpreted as the computed output (i.e., $y_k = w_{jk}$). In the formulation of forward-only CPN also, no topological structure was assumed.

5.5.3.1 Architecture

Figure 5-20 shows the architecture of forward-only CPN. It consists of three layers: input layer, cluster (competitive) layer and output layer. The architecture of forward-only CPN resembles the back-propagation network, but in CPN there exists interconnections between the units in the cluster layer (which are not connected in Figure 5-20). Once competition is completed in a forward-only CPN, only one unit will be active in that layer and it sends signal to the output layer. As inputs are presented to the network, the desired outputs will also be presented simultaneously.

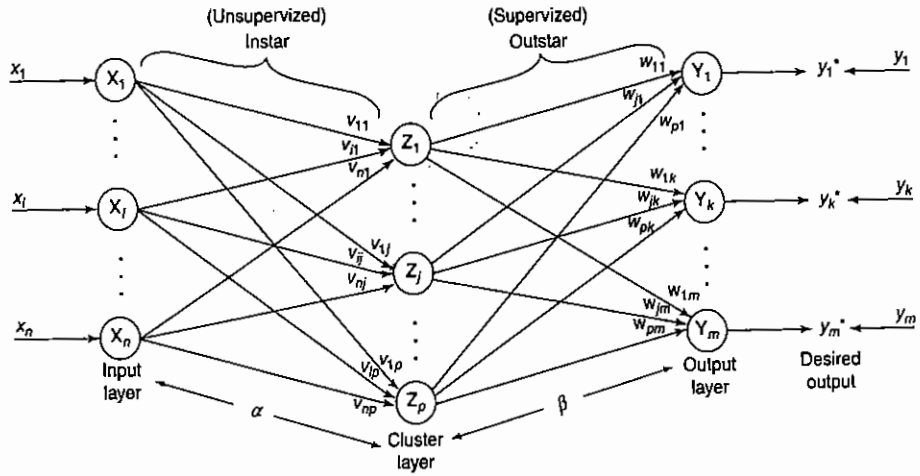


Figure 5-20 Architecture of forward-only CPN.

5.5.3.2 Flowchart

The flowchart helps in depicting the training process of forward-only CPN and the manner in which the weights are updated. The training is performed in two phases. The parameters used in flowchart and training algorithm are as follows:

α, β = learning rate parameters where α = 0.5 to 0.8 and β = 0 to 1. The typical values of learning rates may be $\alpha = 0.6$ and $\beta = 1$

X = activation vector for input layer units, i.e.,

$X = (x_1, \ldots, x_i, \ldots, x_n)$

$\|x - v\|$ = Euclidean distance between vectors X and V

Figure 5-21 shows the flowchart for training process of forward-only CPN.

5.5.3.3 Training Algorithm

The steps involved in the training algorithm of forward-only CPN are as follows:

Step 0: Initialize the weights and learning rates.

Step 1: Perform Steps 2–7 when stopping condition for phase I training is false.

Step 2: Perform Steps 3–5 for each of training input X.

Step 3: Set the X-input layer activations to vector X.

Step 4: Compute the winning cluster unit (J). If dot product method is used, find the cluster unit z_J with the largest net input:

$$z_{inj} = \sum_{i=1}^{n} x_i v_{ij}$$

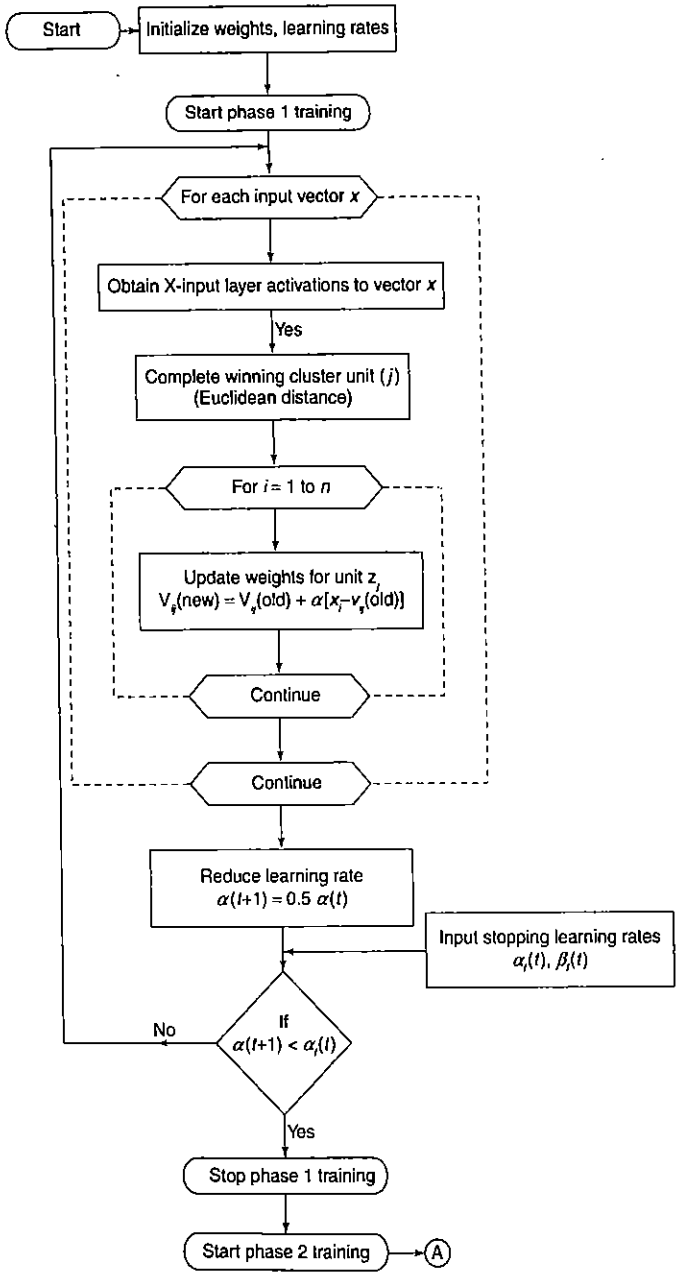


Figure 5-21 Flowchart for training of forward-only CPN.

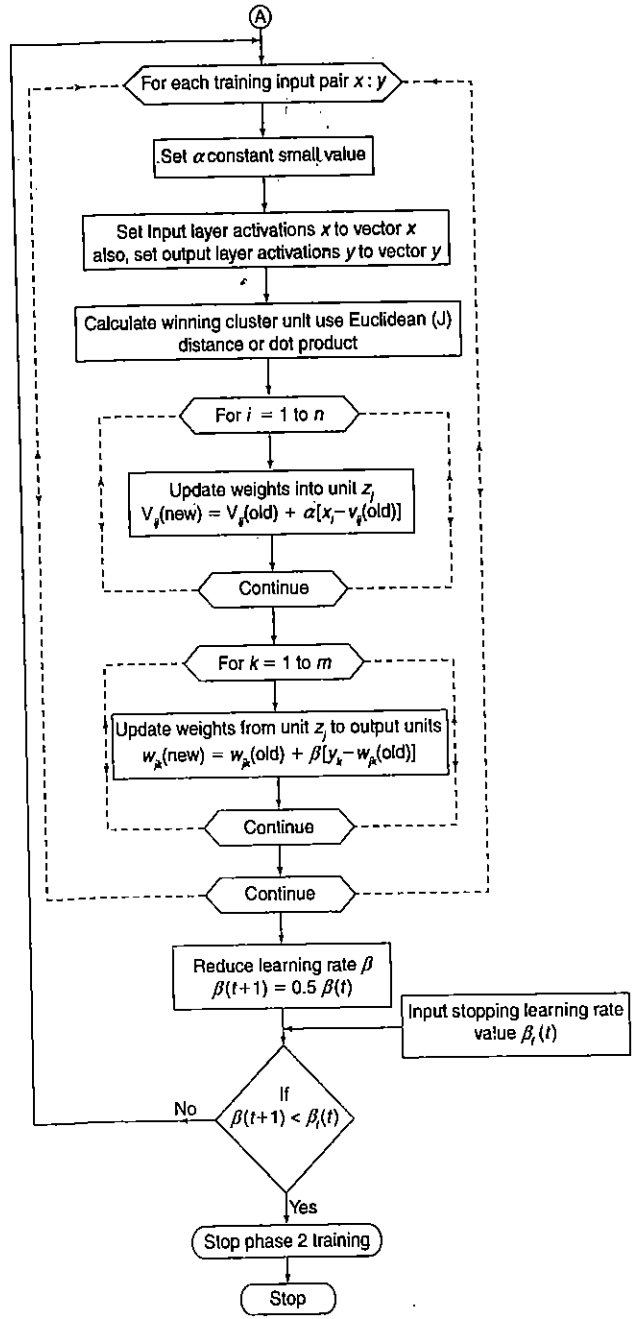


Figure 5-21 (*continued*).

If Euclidean distance is used, find the cluster unit z_j square of whose distance from the input pattern is smallest:

$$D_j = \sum_{i=1}^{n} (x_i - v_{ij})^2$$

If there exists a tie in the selection of winner unit, the unit with the smallest index is chosen as the winner.

Step 5: Perform weight updation for unit z_j. For $i = 1$ to n,

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

Step 6: Reduce learning rate α

$$\alpha(t+1) = 0.5\,\alpha(t)$$

Step 7: Test the stopping condition for phase I training.

Step 8: Perform Steps 9–15 when stopping condition for phase II training is false. (Set α a small constant value for phase II training.)

Step 9: Perform Steps 10–13 for each training input pair x:y.

Step 10: Set X-input layer activations to vector X. Set Y-output layer activations to vector Y.

Step 11: Find the winning cluster unit (J) [use formulas as in Step 4].

Step 12: Update the weights into unit z_j. For $i = 1$ to n,

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

Step 13: Update the weights from unit z_j to the output units. For $k = 1$ to m,

$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \beta\,[y_k - w_{jk}(\text{old})]$$

Step 14: Reduce learning rate β, i.e.,

$$\beta\,(t+1) = 0.5\beta\,(t)$$

Step 15: Test the stopping condition for phase II training.

The stopping condition for both phase I and phase II training may be the reduction in learning rate or number of iterations to be performed.

5.5.3.4 *Testing Algorithm*

The testing algorithm used for forward-only CPN is given as follows:

Step 0: Set initial weights. (The initial weights here are the weights obtained during training.)

Step 1: Present input vector X.

Step 2: Find unit J that is closest to vector X.

Step 3: Set activations of output units:

$$y_k = w_{jk}$$

As in the case of full CPN, the forward-only CPN can also be used in the interpolation mode. Here, if more than one unit is the winner, with nonzero activation value, then

$$\sum_{j=1}^{p} Z_j = 1$$

Hence the activation of the output unit is given by

$$y_k = \sum_j Z_j w_{jk}$$

Use of interpolation mode results in increase of accuracy.

5.6 Adaptive Resonance Theory Network

5.6.1 Theory

The adaptive resonance theory (ART) network, developed by Steven Grossberg and Gail Carpenter (1987), is consistent with behavioral models. This is an unsupervised learning, based on competition, that finds categories autonomously and learns new categories if needed. The adaptive resonance model was developed to solve the problem of instability occurring in feed-forward systems. There are two types of ART: ART 1 and ART 2. ART 1 is designed for clustering binary vectors and ART 2 is designed to accept continuous-valued vectors. In both the nets, input patterns can be presented in any order. For each pattern, presented to the network, an appropriate cluster unit is chosen and the weights of the cluster unit are adjusted to let the cluster unit learn the pattern. This network controls the degree of similarity of the patterns placed on the same cluster units. During training, each training pattern may be presented several times. It should be noted that the input patterns should not be presented on the same cluster unit, when it is presented each time. On the basis of this, the stability of the net is defined as that wherein a pattern is not presented to previous cluster units. The stability may be achieved by reducing the learning rates. The ability of the network to respond to a new pattern equally at any stage of learning is called as plasticity. ART nets are designed to possess the properties, stability and plasticity. The key concept of ART is that the stability plasticity can be resolved by a system in which the network includes bottom-up (input–output) competitive learning combined with top-down (output–input) learning. The instability of instar–outstar networks could be solved by reducing the learning rate gradually to zero by freezing the learned categories. But, at this point, the net may lose its plasticity or the ability to react to new data. Thus it is difficult to possess both stability and plasticity. ART networks are designed particularly to resolve the stability–plasticity dilemma, that is, they are stable to preserve significant past learning but nevertheless remain adaptable to incorporate new information whenever it appears.

5.6.1.1 *Fundamental Architecture*

Three groups of neurons are used to build an ART network. These include:

1. Input processing neurons (F_1 layer).

2. Clustering units (F_2 layer).
3. Control mechanism (controls degree of similarity of patterns placed on the same cluster).

The input processing neuron (F_1) layer consists of two portions: Input portion and interface portion. The input portion may perform some processing based on the inputs it receives. This is especially performed in the case of ART 2 compared to ART 1. The interface portion of the F_1 layer combines the input from input portion of F_1 and F_2 layers for comparing the similarity of the input signal with the weight vector for the cluster unit that has been selected as a unit for learning. F_1 layer input portion may be denoted as F_1(a) and interface portion as F_1(b).

There exist two sets of weighted interconnections for controlling the degree of similarity between the units in the interface portion and the cluster layer. The bottom-up weights are used for the connection from F_1(b) layer to F_2 layer and are represented by b_{ij} (*i*th F_1 unit to *j*th F_2 unit). The top-down weights are used for the connection from F_2 layer to F_1(b) layer and are represented by t_{ji} (*j*th F_2 unit to *i*th F_1 unit). The competitive layer in this case is the cluster layer and the cluster unit with largest net input is the victim to learn the input pattern, and the activations of all other F_2 units are made zero. The interface units combine the data from input and cluster layer units. On the basis of the similarity between the top-down weight vector and input vector, the cluster unit may be allowed to learn the input pattern. This decision is done by reset mechanism unit on the basis of the signals it receives from interface portion and input portion of the F_1 layer. When cluster unit is not allowed to learn, it is inhibited and a new cluster unit is selected as the victim.

5.6.1.2 *Fundamental Operating Principle*

In ART network, presentation of one input pattern forms a learning trial. The activations of all the units in the net are set to zero before an input pattern is presented. All units in the F_2 layer are inactive. On presentation of a pattern, the input signals are sent continuously, until the learning trial is completed. There exists a user-defined parameter, called vigilance parameter, which controls the degree of similarity of the patterns assigned to the same cluster unit. The function of the reset mechanism is to control the state of each node in F_2 layer. Each unit in F_2 layer, at any time instant, can be in any one of the three states mentioned below.

1. Active: Unit is ON. The activation in this case is equal to 1. For ART 1, $d = 1$ and for ART 2, $0 < d < 1$.
2. Inactive: Unit is OFF. The activation here is zero and the unit may be available to participate in competition.
3. Inhibited: Unit is OFF. The activation here is also zero but the unit here is prevented from participating in any further competition during the presentation of current input vector.

The ART nets can perform their learning in two ways: Fast learning and slow learning. The weight updation takes place rapidly in fast learning, relative to the length of time a pattern is being presented on any particular learning trial. In fast learning, the weights reach equilibrium in each trial. On the contrary, in slow learning the weight change occurs slowly relative to the time taken for a learning trial and the weights do not reach equilibrium in each trial. More patterns have to be presented for slow learning compared to that for fast learning. For each learning trial, there occurs only minimum number of calculations in slow learning. In case of fast learning, the net is considered to be stabilized when each pattern closes its correct cluster unit.

The patterns are binary in ART 1 network, hence the weights associated with each cluster unit stabilize in the fast learning mode. The weight vectors obtained are appropriate for the type of input patterns used in ART 1. In case of ART 2 network, the weights produced by fast learning continue to change each time a pattern is presented. The net is found to stabilize only after few presentations of each training pattern. It is not easy to find equilibrium weights immediately for ART 2 as it is for ART 1. In slow learning process, the weight changes do not reach equilibrium during any particular learning trial and more trials are required before the net stabilizes. Slow learning is generally not adopted for ART 1. For ART 2, the weights produced by slow learning are far better than those produced by fast learning for particular types of data.

5.6.1.3 *Fundamental Algorithm*

This algorithm discovers clusters of a set of pattern vectors. The steps involved in various stages of training algorithm are as follows:

Step 0: Initialize the necessary parameters.
Step 1: Perform Steps 2–9 when stopping condition is false.
Step 2: Perform Steps 3–8 for each input vector.
Step 3: F_1 layer processing is done.
Step 4: Perform Steps 5–7 when reset condition is true.
Step 5: Find the victim unit to learn the current input pattern. The victim unit is going to be the F_2 unit (that is not inhibited) with the largest input.
Step 6: F_1(b) units combine their inputs from F_1(a) and F_2.
Step 7: Test for reset condition.
If reset is true, then the current victim unit is rejected (inhibited); go to Step 4. If reset is false, then the current victim unit is accepted for learning; go to next step (Step 8).
Step 8: Weight updation is performed.
Step 9: Test for stopping condition.

The ART network does not require all training patterns to be presented in the same order, it also accepts if all patterns are presented in the same order; we refer to this as an epoch. The flowchart showing the flow of training process is depicted separately for ART 1 and ART 2.

5.6.2 Adaptive Resonance Theory 1

Adaptive resonance theory 1 (ART 1) network is designed for binary input vectors. As discussed generally, the ART 1 net consists of two fields of units–input unit (F_1 unit) and output unit (F_2 unit)–along with the reset control unit for controlling the degree of similarity of patterns placed on the same cluster unit. There exist two sets of weighted interconnection path between F_1 and F_2 layers. The supplemental unit present in the net provides the efficient neural control of the learning process. Carpenter and Grossberg have designed ART 1 network as a real-time system. In ART 1 network, it is not necessary to present an input pattern in a particular order; it can be presented in any order. ART 1 network can be practically implemented by analog circuits governing the differential equations, i.e., the bottom-up and top-down weights are controlled by differential equations. ART 1 network runs throughout autonomously. It does not require any external control signals and can run stably with infinite patterns of input data.

ART 1 network is trained using fast learning method, in which the weights reach equilibrium during each learning trial. During this resonance phase, the activations of F_1 units do not change; hence the equilibrium weights can be determined exactly. The ART 1 network performs well with perfect binary input patterns, but it is sensitive to noise in the input data. Hence care should be taken to handle the noise.

5.6.2.1 Architecture

The ART 1 network is made up of two units:

1. Computational units.
2. Supplemental units.

In this section we will discuss in detail about these two units.

Computational units

The computational unit for ART 1 consists of the following:

1. Input units (F_1 unit – both input portion and interface portion).
2. Cluster units (F_2 unit – output unit).
3. Reset control unit (controls degree of similarity of patterns placed on same cluster).

The basic architecture of ART 1 (computational unit) is shown in Figure 5-22. Here each unit present in the input portion of F_1 layer (i.e., $F_1(a)$ layer unit) is connected to the respective unit in the interface portion of F_1 layer (i.e., $F_1(b)$ layer unit). Reset control unit has connections from each of $F_1(a)$ and $F_1(b)$ units. Also, each unit in $F_1(b)$ layer is connected through two weighted interconnection paths to each unit in F_2 layer and the reset control unit is connected to every F_2 unit. The X_i unit of $F_1(b)$ layer is connected to Y_j unit of F_2 layer through bottom-up weights (b_{ij}) and the Y_j unit of F_2 is connected to X_i unit of F_1 through top-down weights (t_{ji}). Thus ART 1 includes a bottom-up competitive learning system combined with a top-down outstar learning system. In Figure 5-22 for simplicity only the weighted interconnections b_{ij} and t_{ji} are shown, the other units' weighted interconnections are in a similar way. The cluster layer (F_2 layer) unit is a competitive layer, where only the uninhibited node with the largest net input has nonzero activation.

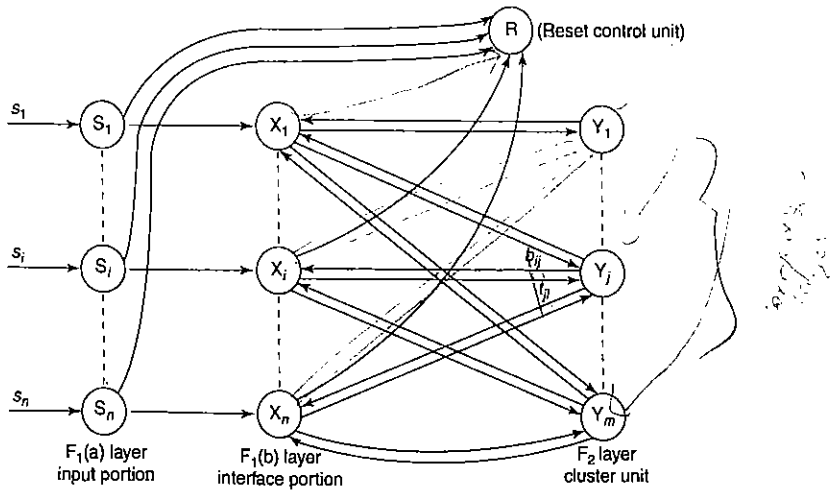


Figure 5-22 Basic architecture of ART 1.

Supplemental units

Figure 5-23 shows the supplemental unit interconnection involving two gain control units along with one reset unit. The discussion on supplemental units is important based on theoretical point of view.

Difficulty faced by computational units: It is necessary for these units to respond differently at different stages of the process, and these are not supported by any of the biological neuron to decide what to do when. The other difficulty is that the operation of the reset mechanism is not well defined for its implementation in neural systems.

The above difficulties are rectified by the introduction of two supplemental units (called as gain control units) G_1 and G_2, along with the reset control unit R. These three units receive signals from and send signals to all of the units in input layer and cluster layer. In Figure 5-23, the excitatory weighted signals are denoted by "+" and inhibitory signals are indicated by "−." Whenever any unit in designated layer is "on," a signal is sent. $F_1(b)$ unit and F_2 unit receive signal from three sources. $F_1(b)$ unit can receive signal from either $F_1(a)$ unit or F_2 units or G_1 unit. In the similar way, F_2 unit receives signal from either $F_1(b)$ unit or reset control unit R or gain control unit G_2. An $F_1(b)$ unit or F_2 unit should receive two excitatory signals for them to be on. Both $F_1(b)$ unit and F_2 unit can receive signals through three possible ways; this is called as two-thirds rule. The $F_1(b)$ unit should send a signal whenever it receives input from $F_1(a)$ and no F_2 node is active. After an F_2 node has been chosen in competition, it is necessary that only $F_1(b)$ units whose input signal and top-down signal match remain constant. This is performed by the two gain control units G_1 and G_2, in addition with two-thirds rule. Whenever F_2 unit is on, G_1 unit is inhibited. When no F_2 unit is on, each F_1 interface unit receives a signal from G_1 unit; here, all of the units that receive a positive input signal from the input vector presented fire. In the same way, G_2 unit controls the firing of F_2 units, obeying the *two thirds rule*. The choice of parameters and initial weights may also be based on two-thirds rule. On the other hand, the vigilance matching is controlled by the reset control unit R. An excitatory signal is always sent to R when any unit in $F_1(a)$ layer is on. The strength of the signal depends on how many F_1 (input) units are on. It should be noted that the reset control unit R also receives inhibitory signals from the F_1 interface units that are on. If sufficient number of interface units is on, then unit "F" may be prevented from firing. When unit "R" fires, it will inhibit any F_2 unit that is on. This may force the F_2 layer to choose a new winning node.

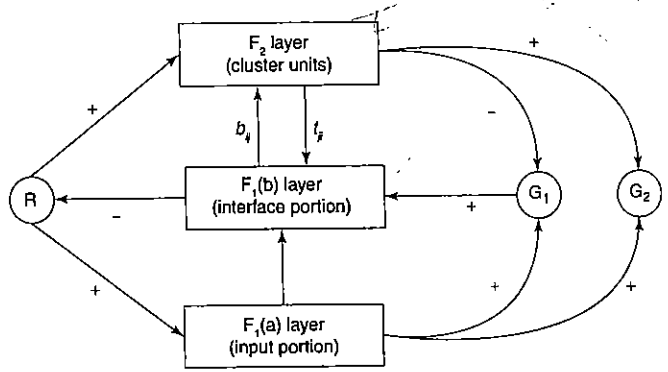


Figure 5-23 Supplemental unit of ART 1.

5.6.2.2 Flowchart of Training Process

The flowchart for the training process of ART 1 network is shown in Figure ??. The parameters used in flowchart and training algorithm are as follows:

n = number of components in training input vector

m = maximum number of cluster units that can be formed

p = vigilance parameter (0 to 1)

b_{ij} = bottom-up weights (weights from X_i unit of F_1(b) layer to Y_j unit of F_2 layer)

t_{ji} = top-down weights (weights from Y_j units of F_2 layer to X_i unit of F_1(b) layer)

s = binary input vector

x = activation vector for F_1(b) layer

$\|x\|$ = norm of vector x that is defined as the sum of components of $x_i (i = 1$ to $n)$

Initially, binary input vector "s" is presented in the F_1(a) layer. Then the signals are sent to the corresponding X layer, i.e., F_1(b) layer. Each F_1(b) layer sends the activation to the F_2 layer over the weighted interconnection paths. Each F_2 layer unit then calculates the net input. The unit with the largest net input is selected as the winner and will have activation "1," the other units' activation will be 0. The winning unit is specified by its index "J." Only this winner unit can learn the current input pattern. Then the signal is send from F_2 layer to F_1(b) layer over the top-down weights (i.e., signals get multiplied with top-down weights). The X units present in the interface portion F_1(b) layer remain on, only if they receive a nonzero signal from both F_1(a) and F_2 layer units.

Now we calculate the factor $\|x\|$. The norm of vector x gives the number of components in which the top-down weight vector for the winning F_2 unit t_J and input vector s are both 1. This is called *Match*. The ratio of norm of x, $\|x\|$, to norm of s, $\|s\|$, is called *Match Ratio*, which if greater than or equal to vigilance parameter, then both the top-down and bottom-up weights have to be adjusted. This is called reset condition. That is

- If $\|x\|/\|s\| \geq \rho$, then weight updation is done. This testing condition is called reset condition.
- If $\|x\|/\|s\| < \rho$, then current unit is rejected and another unit should be chosen. The current winning cluster unit becomes inhibited, so this unit again cannot be chosen as a unit, on this particular learning trial, and the activations of the F_1 units are reset to zero.

This process is repeated until a satisfactory match is found (units get accepted) or until all the units are inhibited.

5.6.2.3 Training Algorithm

The training algorithm for ART 1 network is shown below.

Step 0: Initialize the parameters:

$$\alpha > 1 \text{ and } 0 < \rho \leq 1$$

Initialize the weights:

$$0 < b_{ij}(0) < \frac{\alpha}{\alpha - 1 + n} \text{ and } t_{ji}(0) = 1$$

Step 1: Perform Steps 2–13 when stopping condition is false.

Step 2: Perform Steps 3–12 for each of the training input.

Step 3: Set activations of all F_2 units to zero. Set the activations of F_1(a) units to input vectors.

Step 4: Calculate the norm of s:

$$\|s\| = \sum_i s_i$$

Step 5: Send input signal from F_1(a) layer to F_1(b) layer:

$$x_i = s_i$$

Step 6: For each F_2 node that is not inhibited, the following rule should hold: If $y_j \neq -1$, then

$$y_j = \sum_i b_{ij} x_i$$

Step 7: Perform Steps 8–11 when reset is true.

Step 8: Find J for $y_J \geq y_j$ for all nodes j. If $y_J = -1$, then all the nodes are inhibited and note that this pattern cannot be clustered.

Step 9: Recalculate activation X of F_1(b):

$$x_i = s_i t_{Ji}$$

Step 10: Calculate the norm of vector x:

$$\|x\| = \sum_i x_i$$

Step 11: Test for reset condition.
If $\|x\|/\|s\| < \rho$, then inhibit node J, $y_J = -1$. Go back to step 7 again.
Else if $\|x\|/\|s\| \geq \rho$, then proceed to the next step (Step 12).

Step 12: Perform weight updation for node J (fast learning):

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|}$$

$$t_{Ji}(\text{new}) = x_i$$

Step 13: Test for stopping condition. The following may be the stopping conditions:

a. No change in weights.

b. No reset of units.

c. Maximum number of epochs reached.

When calculating the winner unit, if there occurs a tie, the unit with smallest index is chosen as winner. Note that in Step 3 all the inhibitions obtained from the previous learning trial are removed. When $y_J = -1$, the node is inhibited and it will be prevented from becoming the winner. The unit x_i in Step 9 will be ON only if it receives both an external signal s_i and the other signal from F_2 unit to F_1(b) unit, t_{ji}. Note that t_{ji} is either 0 or 1, and once it is set to 0, during learning, it can never be set back to 1 (provides stable learning method).

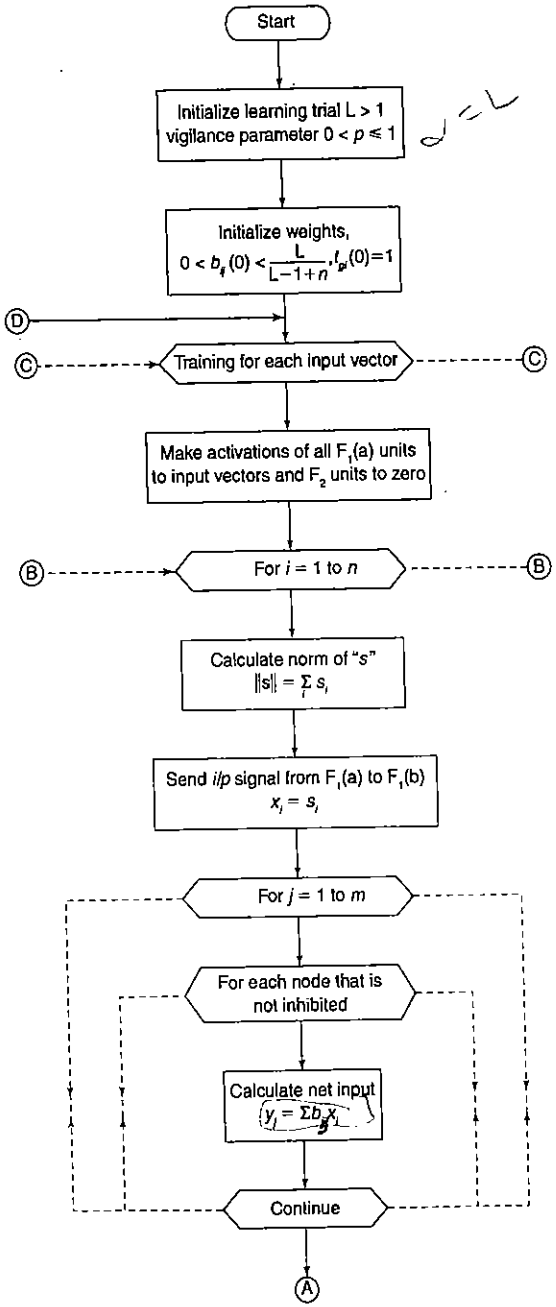


Figure 5-24 Flowchart for training of ART 1 network.

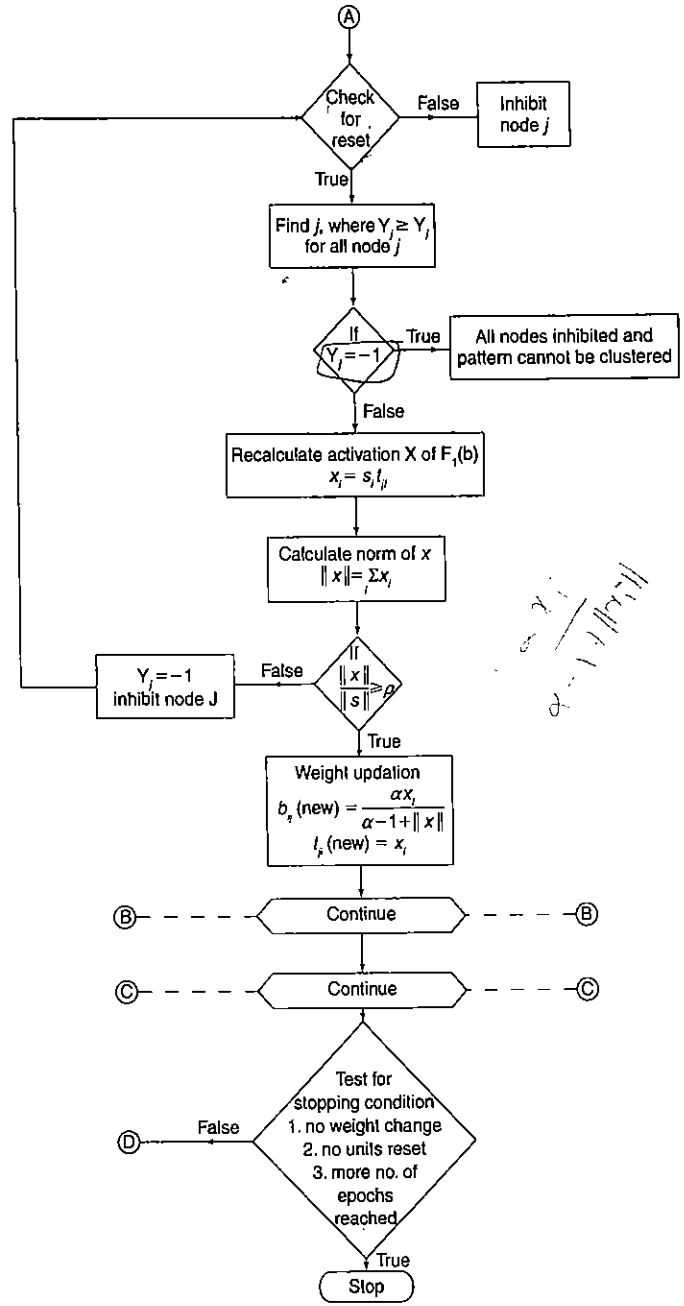


Figure 5-24 (*continued*).

The optimal values of the initial parameters are $\alpha = 2, \rho = 0.9, b_{ij} = 1/1 + n$ and $t_{ji} = 1$. The algorithm uses fast learning, which uses the fact that the input pattern is presented for a longer period of time for weights to reach equilibrium.

5.6.3 Adaptive Resonance Theory 2

Adaptive resonance theory 2 (ART 2) is for continuous-valued input vectors. In ART 2 network complexity is higher than ART 1 network beeause much processing is needed in F_1 layer. ART 2 network was developed by Carpenter and Grossberg in 1987. ART 2 network was designed to self-organize recognition categories for analog as well as binary input sequences. The major difference between ART 1 and ART 2 networks is the input layer. On the basis of the stability criterion for analog inputs, a three-layer feedback system in the input layer of ART 2 network is required: A bottom layer where the input patterns are read in, a top layer where inputs coming from the output layer are read in and a middle layer where the top and bottom patterns are combined together to form a matched pattern which is then fed back to the top and bottom input layers. The complexity in the F_1 layer is essential because continuous-valued input vectors may be arbitrarily close together. The F_1 layer consists of normalization and noise suppression parameter, in addition to comparison of the bottom-up and top-down signals, needed for the reset mechanism.

The continuous-valued inputs presented to the ART 2 network may be of two forms. The first form is a "noisy binary" signal form, where the information about patterns is delivered primarily based on the components which are "on" or "off," rather than the differences existing in the magnitude of the components that are positive. In this case, fast learning mode is best adopted. The second form of patterns are those, in which the range of values of the components carries significant information and the weight vector for a cluster is found to be interpreted as exemplar for the patterns placed on that unit. In this type of pattern, slow learning mode is best adopted. The second form of data is "truly continuous."

5.6.3.1 *Architecture*

A typical architecture of ART 2 network is shown in Figure 5-25. From the figure, we can notice that F_1 layer consists of six types of units – W, X, U, V, P, Q – and there are "n" units of each type. In Figure 5-25, only one of these units is shown. The supplemental part of the connection is shown in Figure 5-26.

The supplemental unit "N" between units W and X receives signals from all "W" units, computes the norm of vector w and sends this signal to each of the X units. This signal is inhibitory signal. Each of this $(X_1, \ldots, X_i, \ldots, X_n)$ also receives excitatory signal from the corresponding W unit. In a similar way, there exists supplemental units between U and V, and P and Q, performing the same operation as done between W and X. Each X unit and Q unit is connected to V unit. The connections between P_i of the F_1 layer and Y_j of the F_2 layer show the weighted interconnections, which multiplies the signals transmitted over those paths. The winning F_2 units' activation is d $(0 < d < 1)$. There exists normalization between W and X, V and U, and P and Q. The normalization is performed approximately to unit length.

The operations performed in F_2 layer are same for both ART 1 and ART 2. The units in F_2 layer compete with each other in a winner-take-all policy to learn each input pattern. The testing of reset condition differs for ART 1 and ART 2 networks. Thus in ART 2 network, some processing of the input vector is necessary because the magnitudes of the real valued input vectors may vary more than for the binary input vectors.

5.6.3.2 *Algorithm*

A detailed description of algorithm used in ART 2 network is discussed below. First, let us analyze the supplemental connection between W_i and X_i units.

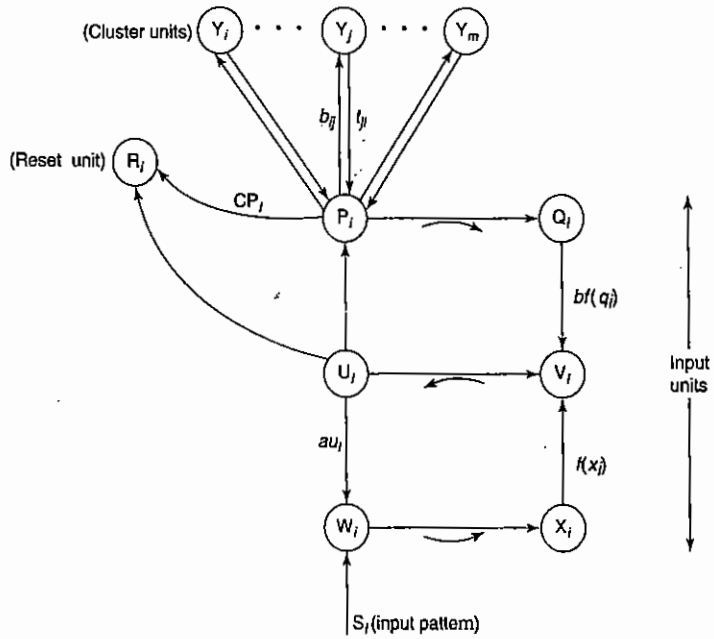


Figure 5-25 Architecture of ART 2 network.

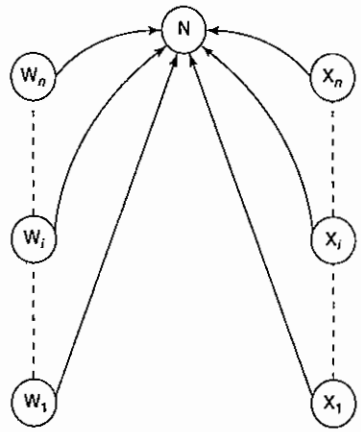


Figure 5-26 Supplemental part of connection between W and X.

Supplemental connection between W_i and X_i units

As discussed in Section 5.6.3.1, there exist supplemental connections between W and X, U and V, and P and Q. Each of the x_i receives signal from w_i units. After receiving, it will calculate the norm of w, $\|w\|$ and then sends that signal to each of the X units. Normalization is done in the F_1 units from W to X, V to U and P to Q. Each of the X_i units are connected to V_i and Q_i units are also connected to V_i. The weights a, b, c shown in Figure 5-25 are fixed. The weights on the connection path indicate the transformation taking place from one unit to other (no multiplication takes place here), i.e., u_i is transformed to au_i but not multiplied. When signals are transferred from F_1 units to F_2 units, i.e., from P_i to Y_j, the multiplication of weights is done. The activation of the F_2 unit is "d" which ranges between 0 and 1 ($0 < d < 1$). It should be noted that these activations are continuously changing.

Processing of F_1 layer and F_2 layer

For understanding the training algorithm of ART 2 network, it is important to know the processing of F_1 and F_2 layers. In F_1 layer, the output activation from P_i is p and output activation from Q_i is q. The activation vector q, which is the activation of Q_i units, should be equal to vector p, activation of P_i units that is normalized approximately for unit length. U_i unit performs similar process of F_1(a) layer of ART 1 and P_i unit performs similar process of F_1(b) layer of ART 1 network. The activation function used here is the functional representation of noise suppression parameter "Q," and is given by

$$f(x) = \begin{cases} x & x \geq \theta \\ 0 & x < \theta \end{cases}$$

The noise suppression parameter Q is defined by the user and is used to achieve stability. Stability occurs where there is no reset, i.e., the same winner unit is chosen in the next trial also. Units x_i and Q_i apply activation to V_i, which suppresses the components to achieve stability. Hence Q is used here.

In ART 2 network continuous processing of the input units is done. The continuous-valued input signals $s = (s_1, \ldots, s_i, \ldots, s_n)$ are sent continuously. For each learning trial, one input pattern is presented. At the beginning of training, the activations are set to zero, i.e., inactive not inhibit. The computation cycle for a particular learning trial within F_1 layers starts with u_i which is equal to activation of V_i approximated to unit length. Unit u_i is given by

$$u_i = \frac{v_i}{e + \|v\|}$$

where "e" is a small parameter for preventing the division by zero when $\|v\|$ becomes zero. Also q_i and x_i are given by

$$q_i = \frac{p_i}{e + \|p\|}; \quad x_i = \frac{w_i}{e + \|w\|}$$

The noise suppression parameter is applied only to x_i and q_i.

The signal will be sent from each unit of u_i to w_i and p_i. The activations of units w_i and p_i have to be done. The activation of w_i is the sum of input signal received (s_i) and au_i:

$$w_i = s_i + au_i$$

P_i is found to receive signals from u_i and top-down weights, i.e., sums u_i (activation of u_i) and top-down weight (t_{ji}), and is given by

$$P_i = u_i + dt_{ji}$$

where d is the activation of winning F_2 unit. Before entering into V_i, activation function is applied to each of x_i and Q_i units. Unit V_i sums the signals from x_i and Q_i which receive signal concurrently:

$$v_i = f(x_i) + bf(q_i)$$

Activation function is designed to select the noise suppression parameter (user specified "Q"). According to Carpenter and Grossberg, the activations of P_i and Q_i (i.e., the outputs) will reach equilibrium (stable set of weights) only after two updates of weights. This completes the phase I or one-cycle process of F_1 layer. Only after F_1 units reach equilibrium, the processing of F_2 layer starts (i.e., after three updates).

F_2 layer being a competitive layer uses winner-take-all policy to determine its winner. Dot product method may be used for the selection of the winner. When the top-down and weight vector remain similar, then that unit is the winner (active). If for a unit, the top-down and input vectors are not similar, then that unit becomes inhibit. This layer receives signals from P_i units via bottom-up weights and P_i units in turn send signals to F_i unit. Here only the winner unit is allowed to learn the input pattern S_i.

The reset mechanism controls the degree of similarity of the input patterns. The checking for reset condition in ART 2 differs from ART 1 network. The reset is checked every time it receives signal from u_i and P_i.

In fast learning mode, the updation of weights is continued until the weights reach equilibrium on each trial. It requires only less number of epochs, but a large number of iterations through the weight update–F_1 portion must be performed on each learning trial. Here, the placement of patterns on clusters stabilizes, but the weight will change for each pattern presented.

In slow learning mode, only one iteration of weight updates will be performed on each learning trial. Large number of learning trials is required for each pattern, but only little computation is done on each trial. There is no requirement that the patterns should be presented in the same order or that exactly the same set of patterns is presented on each cycle through them. Thus it is preferable to have slow learning than fast learning.

Computations for algorithm

The following computations have to be performed in several steps of the algorithm and are referred as "updation of F_1 activations." Unit J is the winning F_2 unit after competition is completed. If no winning unit is chosen, then "d" is zero for all units. The calculations for P_i and w_i, and x_i and q_i can be done in parallel.

F_1 layer consists of six units; the update F_1 activations are given by

$$u_i = \frac{v_i}{e + \|v\|}; \quad P_i = u_i + dt_{ji}$$

$$w_i = s_i + au_i; \quad x_i = \frac{w_i}{e + \|w\|}$$

$$q_i = \frac{p_i}{e + \|p\|}; \quad v_i = f(x_i) + bf(q_i)$$

The activation function is given by

$$f(x) = \begin{cases} x & f(x) \geq \theta \\ 0 & f(x) < \theta \end{cases}$$

5.6.3.3 *Flowchart*

The flowchart for the training process of ART 2 network is shown in Figure 5-27. The flowchart clearly depicts the flow of the training process of the network. The check for reset in the flowchart differs for ART 1 and ART 2 networks.

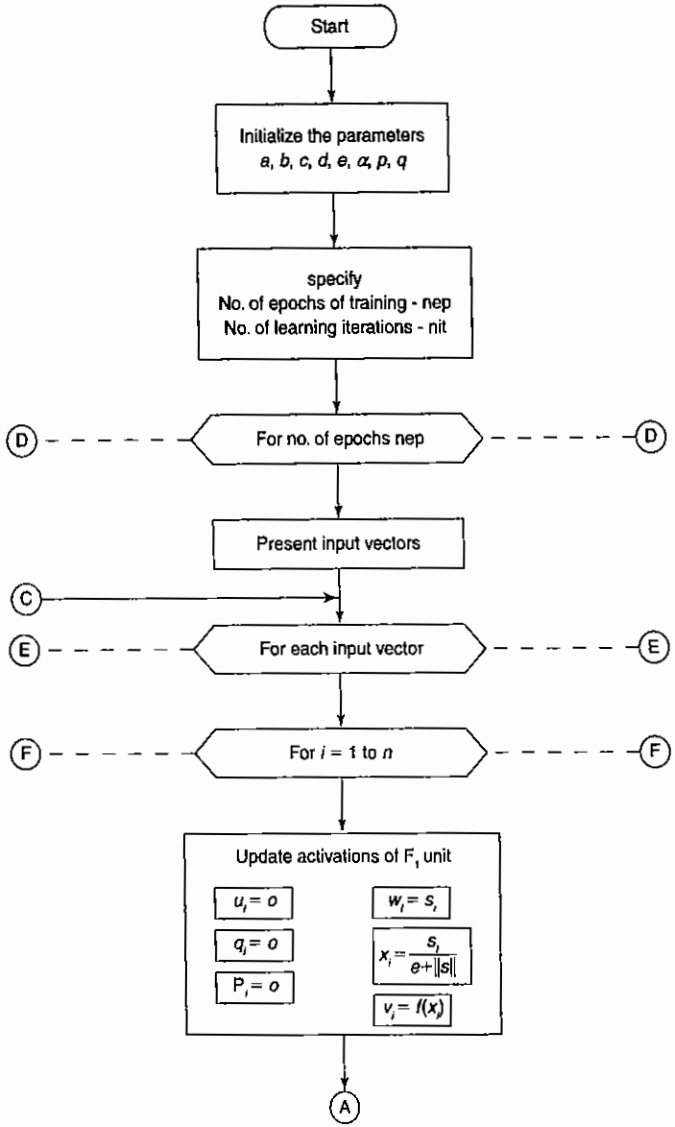


Figure 5-27 Flowchart for training of ART 2 network.

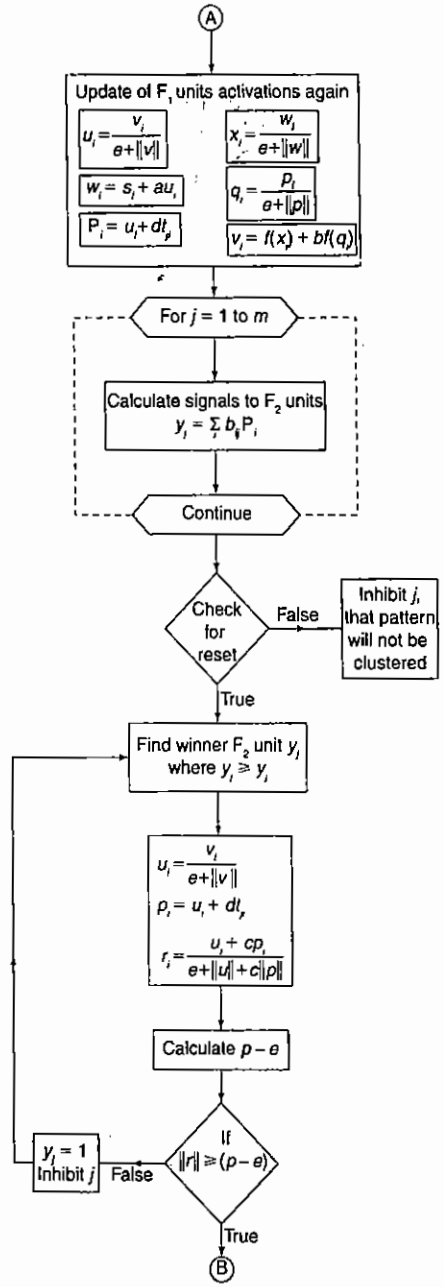


Figure 5-27 (*continued*).

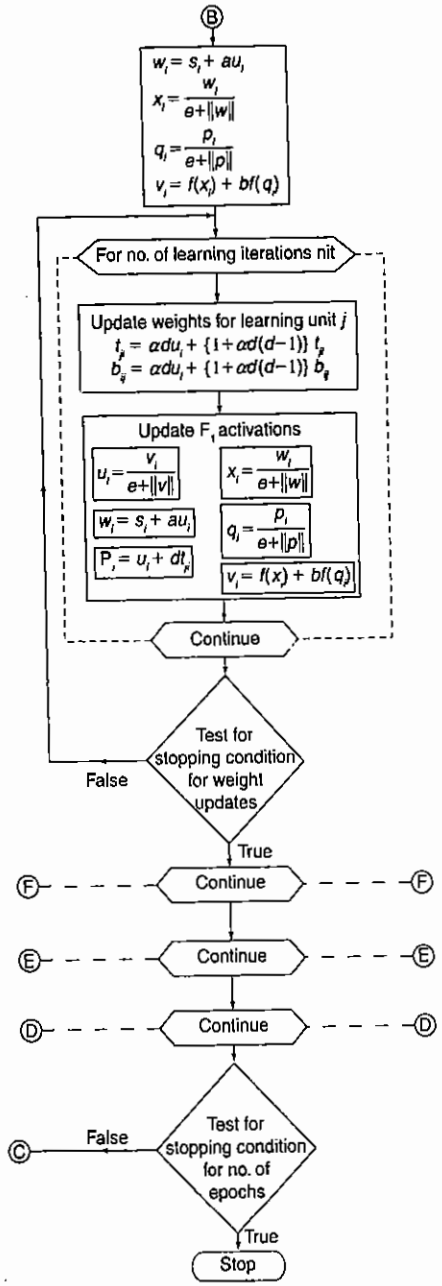


Figure 5-27 (*continued*).

5.6.3.4 Training Algorithm

The training algorithm of ART 2 network is shown below.

Step 0: Initialize the following parameters: a, b, c, d, e, α, ρ, θ. Also, specify the number of epochs of training (nep) and number of learning iterations (nit).

Step 1: Perform Steps 2–12 (nep) times.

Step 2: Perform Steps 3–11 for each input vector s.

Step 3: Update F_1 unit activations:

$$u_i = 0; \quad w_i = s_i; \quad P_i = 0; \quad q_i = 0; \quad v_i = f(x_i);$$

$$x_i = \frac{s_i}{e + \|s\|}$$

Update F_1 unit activations again:

$$u_i = \frac{v_i}{e + \|v\|}; \quad w_i = s_i + au_i;$$

$$P_i = u_i; \quad x_i = \frac{w_i}{e + \|w\|};$$

$$q_i = \frac{p_i}{e + \|p\|}; \quad v_i = f(x_i) + bf(q_i)$$

In ART 2 networks, norms are calculated as the square root of the sum of the squares of the respective values.

Step 4: Calculate signals to F_2 units:

$$y_j = \sum_{i=1}^{n} b_{ij} p_i$$

Step 5: Perform Steps 6 and 7 when reset is true.

Step 6: Find F_2 unit Y_J with largest signal (J is defined such that $y_j \geq y_j, j = 1$ to m).

Step 7: Check for reset:

$$u_i = \frac{v_i}{e + \|v\|}; \quad P_i = u_i + dt_{ji}; \quad r_i = \frac{u_i + cP_i}{e + \|u\| + c\|p\|}$$

If $\|r\| < (\rho - e)$, then $y_J = -1$ (inhibit J). Reset is true; perform Step 5.
If $\|r\| \geq (\rho - e)$, then

$$w_i = s_i + au_i; \quad x_i = \frac{w_i}{e + \|w\|};$$

$$q_i = \frac{p_i}{e + \|p\|}; \quad v_i = f(x_i) + bf(q_i)$$

Reset is false. Proceed to Step 8.

Step 8: Perform Steps 9–11 for specified number of learning iterations.

Step 9: Update the weights for winning unit J:

$$t_{Ji} = \alpha d u_i + \{[1+\alpha d(d-1)]\} t_{Ji}$$
$$b_{iJ} = \alpha d u_i + \{[1+\alpha d(d-1)]\} b_{iJ}$$

Step 10: Update F_1 activations:

$$u_i = \frac{v_i}{e + \|v\|}; \quad w_i = s_i + a u_i;$$
$$P_i = u_i + d t_{Ji}; \quad x_i = \frac{w_i}{e + \|w\|};$$
$$q_i = \frac{P_i}{e + \|p\|}; \quad v_i = f(x_i) + b f(q_i)$$

Step 11: Check for the stopping condition of weight updation.

Step 12: Check for the stopping condition for number of epochs.

In the above algorithm, at resonance period, reset will not occur and new winning unit cannot be chosen. Since in slow learning number of learning iterations is 1, Step 10 in training algorithm need not be processed. Perform Step 8 until the weight changes are below some specified tolerance. If slow learning is performed, then repeat Step 1 until the weight changes are below some specified tolerance. If fast learning is adopted, then repeat Step 1 until the patterns placement on the cluster units do not change from one epoch to the next.

5.6.3.5 *Sample Values of Parameter*

The sample values of the parameters used in ART 2 network and their role in effective training process are mentioned below.

- $n =$ number of F_1 layer input units
- $m =$ number of F_2 layer cluster units
- $a, b =$ fixed weights present in the F_1 layer. The sample values are $a = 10$ and $b = 10$; when $a = 0$ and $b = 0$, the net becomes instable
- $c =$ fixed weight for testing of reset. The sample value is $c = 0.1$. For small c, larger effective range of the vigilance parameter is achieved
- $d =$ activation of winning F_2 unit. Sample value is $d = 0.9$. The values of c and d should be selected satisfying the inequality $cd/1 - d \leq 1$. The value of $cd/1 - d$ should be closer to 1, so that effective vigilance could be achieved
- $e =$ a small parameter included to prevent division by zero error when the norm of vector is zero.
- $\theta =$ noise suppression parameter. A sample value of noise suppression parameter is $\theta = 1/\sqrt{n}$. The components of the normalized input vector, which are less than this value, are set to zero.
- $\alpha =$ learning rate parameter. In both slow and fast learning methods, a small value of α slows down the learning process.
- $\rho =$ vigilance parameter used in reset condition. Vigilance parameter can range from 0 to 1. For effectively controlling the number of clusters, a sample value of 0.7–1 may be allowed. The range of ρ may also be affected by the values of c and d.
- $t_{ji}(0) =$ initial top-down weights. The initial weights of this weight vectors are given by $t_{ji}(0) = 0$.
- $b_{ij}(0) =$ initial bottom-up weights. These should be chosen to satisfy the inequality $b_{ij}(0) \leq 1/(1-d)\sqrt{n}$. High values of b_{ij} allow the net to form more clusters.

On the basis of all these roles of parameters and their sample values, care should be taken in selecting their values for effective training of the network.

5.7 Summary

Unsupervised learning networks are widely used when clustering of units is performed. In case of unsupervised learning networks, the information about the output is not known; when the weights of a net remain fixed for the entire operation then it results in fixed weight competitive nets. Fixed weight competitive nets include Maxnet, Mexican hat and Hamming net. In case of Hamming net, Maxnet is used as a subnet. The most important unsupervised learning network is the Kohonen self-organizing feature map, where clustering is performed over the training vectors and the network training is achieved. An extension of KSOFM, Kohonen self-organizing motor map is also included. An unsupervised learning network with targets known is the learning vector quantization (LVQ) network. A study is made on LVQ net with its architecture, flowchart for training process and training algorithm. The variants of LVQ net are also included. The compression network discussed in this chapter is the counterpropagation network (CPN). The two types of counterpropagation networks – full CPN and forward-only CPN – are discussed. The testing algorithms for these networks are also given. Another important unsupervised learning network is the adaptive resonance theory (ART) network. In this chapter, ART 1 and ART 2 networks with all relevant information are discussed in detail.

5.8 Solved Problems

1. Construct a Maxnet with four neurons and inhibitory weight $\varepsilon = 0.2$, given the initial activations (input signals) as follows:

$$a_1(0) = 0.3; \quad a_2(0) = 0.5; \quad a_3(0) = 0.7;$$
$$a_4(0) = 0.9$$

Solution: Update the activations for each node, i.e.,

$$a_j(\text{new}) = f\left[a_j(\text{old}) - \varepsilon \sum_{k \neq j} a_k(\text{old})\right]$$

The activation function is given by

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

First iteration:

$$a_1(1) = f\left[a_1(0) - \varepsilon \sum_{k \neq j} a_k(0)\right]$$
$$= f[0.3 - 0.2(0.5 + 0.7 + 0.9)]$$
$$= f(0.3 - 0.42) = f(-0.12) = 0$$

$$a_2(1) = f\left[a_2(0) - \varepsilon \sum_{k \neq j} a_k(0)\right]$$
$$= f[0.5 - 0.2(0.3 + 0.7 + 0.9)]$$
$$= f(0.12) = 0.12$$

$$a_3(1) = f\left[a_3(0) - \varepsilon \sum_{k \neq j} a_k(0)\right]$$
$$= f[0.7 - 0.2(0.3 + 0.5 + 0.9)]$$
$$= f(0.36) = 0.36$$

$$a_4(1) = f\left[a_4(0) - \varepsilon \sum_{k \neq j} a_k(0)\right]$$
$$= f[0.9 - 0.2(0.3 + 0.5 + 0.7)]$$
$$= f(0.9 - 0.3) = f(0.6) = 0.6$$

Second iteration:

$$a_1(2) = f\left[a_1(1) - \varepsilon \sum_{k \neq j} a_k(1)\right]$$
$$= f[0 - 0.2(0.12 + 0.36 + 0.6)]$$
$$= f(0 - 0.216) = 0$$

$$a_2(2) = f\left[a_2(1) - \varepsilon \sum_{k \neq j} a_k(1)\right]$$
$$= f[0.12 - 0.2(0 + 0.36 + 0.6)]$$
$$= f(-0.072) = 0$$

$$a_3(2) = f\left[a_3(1) - \varepsilon \sum_{k \neq j} a_k(1)\right]$$
$$= f[0.36 - 0.2(0 + 0.12 + 0.6)]$$
$$= f(0.216) = 0.216$$

$$a_4(2) = f\left[a_4(1) - \varepsilon \sum_{k \neq j} a_k(1)\right]$$
$$= f[0.6 - 0.2(0 + 0.12 + 0.36)]$$
$$= f(0.504) = 0.504$$

Third iteration:

$$a_1(3) = f\left[a_1(2) - \varepsilon \sum_{k \neq j} a_k(2)\right]$$
$$= f[0 - 0.2(0 + 0.216 + 0.504)]$$
$$= f(0 - 0.144) = 0$$

$$a_2(3) = f\left[a_2(2) - \varepsilon \sum_{k \neq j} a_k(2)\right]$$
$$= f[0 - 0.2(0 + 0.216 + 0.504)]$$
$$= f(0 - 0.144) = 0$$

$$a_3(3) = f\left[a_3(2) - \varepsilon \sum_{k \neq j} a_k(2)\right]$$
$$= f[0.216 - 0.2(0 + 0 + 0.504)]$$
$$= f(0.1152) = 0.1152$$

$$a_4(3) = f\left[a_4(2) - \varepsilon \sum_{k \neq j} a_k(2)\right]$$
$$= f[0.504 - 0.2(0 + 0 + 0.216)]$$
$$= f(0.4608) = 0.4608$$

Fourth iteration:

$$a_1(4) = f\left[a_1(3) - \varepsilon \sum_{k \neq j} a_k(3)\right]$$
$$= f[0 - 0.2(0 + 0.1152 + 0.4608)] = 0$$

$$a_2(4) = f\left[a_2(3) - \varepsilon \sum_{k \neq j} a_k(3)\right]$$
$$= f[0 - 0.2(0 + 0.1152 + 0.4608)] = 0$$

$$a_3(4) = f\left[a_3(3) - \varepsilon \sum_{k \neq j} a_k(3)\right]$$
$$= f[0.1152 - 0.2(0 + 0 + 0.4608)]$$
$$= f(0.02304) = 0.02304$$

$$a_4(4) = f\left[a_4(3) - \varepsilon \sum_{k \neq j} a_k(3)\right]$$
$$= f[0.4608 - 0.2(0 + 0 + 0.1152)]$$
$$= f(0.43776) = 0.43776$$

Fifth iteration:

$$a_1(5) = f\left[a_1(4) - \varepsilon \sum_{k \neq j} a_k(4)\right]$$
$$= f[0 - 0.2(0 + 0.02304 + 0.43776)] = 0$$

$$a_2(5) = f\left[a_2(4) - \varepsilon \sum_{k \neq j} a_k(4)\right]$$
$$= f[0 - 0.2(0 + 0.02304 + 0.43776)] = 0$$

$$a_3(5) = f\left[a_3(4) - \varepsilon \sum_{k \neq j} a_k(4)\right]$$
$$= f[0.0234 - 0.2(0 + 0 + 0.43776)]$$
$$= f(-0.0645) = 0$$

$$a_4(5) = f\left[a_4(4) - \varepsilon \sum_{k \neq j} a_k(4)\right]$$
$$= f[0.43776 - 0.2(0 + 0 + 0.02304)]$$
$$= f(0.433152) = 0.433152$$

Even though if further iterations are made, the value of $a_4(5)$ remains the same, since all the other values $a_1(5)$, $a_2(5)$ and $a_3(5)$ are equal to zero. Thus the convergence has occurred.

2. Construct and test the Hamming network to cluster four vectors. Given the exemplar vectors,

$$e(1) = [1\ -1\ -1\ -1]; \quad e(2) = [-1\ -1\ -1\ 1]$$

the bipolar input vectors are $x_1 = [-1\ -1\ 1\ -1]$; $x_2 = [-1\ -1\ 1\ 1]$; $x_3 = [-1\ -1\ -1\ 1]$; $x_4 = [1\ 1\ -1\ -1]$.

Solution: Number of components in input vector $n = 4$.

Number of exemplar vectors $m = 2$.

Setting the initial weights to 1/2 of the exemplar vectors, we get

$$w_{ij} = \frac{e_{ij}}{2} = \left(\frac{e(1)}{2} \quad \frac{e(2)}{2}\right)$$

where

$$e(1) = [1\ -1\ -1\ -1]; \quad e(2) = [-1\ -1\ -1\ 1]$$

Step 0: The weights are given by

$$w_{ij} = \begin{bmatrix} 0.5 & -0.5 \\ -0.5 & -0.5 \\ -0.5 & -0.5 \\ -0.5 & 0.5 \end{bmatrix}$$

Setting the bias to $n/2$, we obtain

$$b_1 = b_2 = \frac{n}{2} = \frac{4}{2} = 2$$

First input vector:

Step 1: For $x_1 = [-1\ -1\ 1\ -1]$, perform Steps 2–4.

Step 2:

$$y_{in1} = b_1 + \sum_i x_i w_{i1}$$
$$= 2 + [-1\ -1\ 1\ -1] \begin{bmatrix} 0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}$$
$$= 2 - 0.5 + 0.5 - 0.5 + 0.5 = 2$$

The value $y_{in1} = 2$ is the number of components at which the input vector x_1 and $e(1)$ agree. Now

$$y_{in2} = b_2 + \sum_i x_i w_{i2}$$
$$= 2 + [-1\ -1\ 1\ -1] \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \\ +0.5 \end{bmatrix}$$
$$= 2 + 0.5 + 0.5 - 0.5 - 0.5 = 2$$

The value $y_{in2} = 2$ is the number of components at which the input vector x_1 and $e(2)$ agree.

Step 3: Initialize the activations of Maxnet as

$$y_1(0) = 2: \quad y_2(0) = 2$$

Step 4: Since $y_1(0) = y_2(0)$, Maxnet will find the unit with the smallest index as the best match exemplar for input $x_1 = [-1\ -1\ 1\ -1]$ or in some cases both may be chosen as best match exemplars.

Second input vector:

Step 1: For $x_2 = [-1\ -1\ 1\ 1]$, perform Steps 2–4.

Step 2:

$$y_{in1} = b_1 + \sum_i x_i w_{i1}$$
$$= 2 + [-1\ -1\ 1\ 1] \begin{bmatrix} +0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}$$
$$= 2 - 0.5 + 0.5 - 0.5 - 0.5 = 1$$

$$y_{in2} = b_2 + \sum_i x_i w_{i2}$$

$$= 2 + [-1 \ -1 \ 1 \ 1] \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \\ 0.5 \end{bmatrix}$$

$$= (2 + 0.5 - 0.5 + 0.5 + 0.5) = 3$$

Step 3: Initialize the activations of Maxnet as

$$y_1(0) = 1; \quad y_2(0) = 3$$

Step 4: Since $y_2(0) > y_1(0)$, Maxnet will find that the unit y_2 has the best match exemplar for input vector $x_2 = [-1 \ -1 \ 1 \ 1]$.

Third input vector:

Step 1: For $x_3 = [-1 \ -1 \ -1 \ 1]$, perform Steps 2–4.

Step 2:

$$y_{in1} = b_1 + \sum_i x_i w_{i1}$$

$$= 2 + [-1 \ -1 \ -1 \ 1] \begin{bmatrix} 0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}$$

$$= 2 - 0.5 + 0.5 + 0.5 - 0.5 = 2$$

$$y_{in2} = b_2 + \sum_i x_i w_{i2}$$

$$= 2 + [-1 \ -1 \ -1 \ 1] \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \\ 0.5 \end{bmatrix}$$

$$= 2 + 0.5 + 0.5 + 0.5 + 0.5 = 4$$

Step 3: Initialize the activations of Maxnet as

$$y_1(0) = 2; \quad y_2(0) = 4$$

Step 4: Since $y_2(0) > y_1(0)$, Maxnet will find the unit y_2 as the best match exemplar for input vector $x_3 = [-1 \ -1 \ -1 \ 1]$.

Fourth input vector:

Step 1: For $x_4 = [1 \ 1 \ -1 \ -1]$, perform Steps 2–4.

Step 2:

$$y_{in1} = b_1 + \sum_i x_i w_{i1}$$

$$= 2 + [1 \ 1 \ -1 \ -1] \begin{bmatrix} 0.5 \\ -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}$$

$$= 2 + 0.5 - 0.5 + 0.5 + 0.5 = 3$$

$$y_{in2} = b_2 + \sum_i x_i w_{i2}$$

$$= 2 + [1 \ 1 \ -1 \ -1] \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \\ 0.5 \end{bmatrix}$$

$$= 2 - 0.5 - 0.5 + 0.5 - 0.5 = 1$$

Step 3: Initialize the activations of Maxnet as

$$y_1(0) = 3; \quad y_2(0) = 1$$

Step 4: Since $y_1(0) > y_2(0)$, Maxnet will find that the unit y_1 is the best match exemplar for the input vector $x_4 = [1 \ 1 \ -1 \ -1]$.

The architecture for the Hamming net for this problem is given by Figure 1.

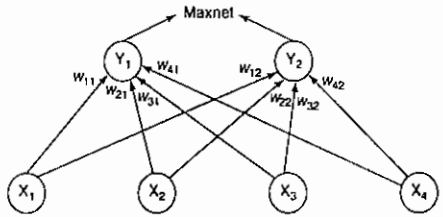


Figure 1 Hamming net architecture.

3. Construct a Kohonen self-organizing map to cluster the four given vectors, [0 0 1 1], [1 0 0 0], [0 1 1 0] and [0 0 0 1]. The number of clusters to be formed is two. Assume an initial learning rate of 0.5.

Solution: The number of input vectors is four and number of clusters to be formed is two. Thus, $n = 4$ and $m = 2$. The architecture of the Kohonen self-organizing feature map is given by Figure 2.

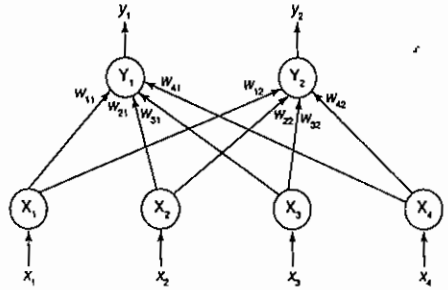


Figure 2 Architecture of KSOFM.

Step 0: Initialize the weights randomly between 0 and 1.

$$w_{ij} = \begin{bmatrix} 0.2 & 0.9 \\ 0.4 & 0.7 \\ 0.6 & 0.5 \\ 0.8 & 0.3 \end{bmatrix}_{4\times 2}; \quad R = 0; \alpha(0) = 0.5$$

First input vector:

Step 1: For $x = [0 \ 0 \ 1 \ 1]$, perform Steps 2–4.

Step 2: Calculate the Euclidean distance:

$$D(j) = \sum_i (w_{ij} - x_i)^2$$

$$D(1) = \sum_{i=1}^{4} (w_{i1} - x_i)^2$$

$$= (0.2 - 0)^2 + (0.4 - 0)^2 + (0.6 - 1)^2 + (0.8 - 1)^2$$

$$= 0.04 + 0.16 + 0.16 + 0.04$$

$$= 0.4$$

$$D(2) = \sum_{i=1}^{4} (w_{i2} - x_i)^2$$

$$= (0.9 - 0)^2 + (0.7 - 0)^2 + (0.5 - 1)^2 + (0.3 - 1)^2$$

$$= 0.81 + 0.49 + 0.25 + 0.49$$

$$= 2.04$$

Step 3: Since $D(1) < D(2)$, therefore $D(1)$ is minimum. Hence the winning cluster unit is Y_1, i.e., $J = 1$.

Step 4: Update the weights on the winning cluster unit $J = 1$.

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$

$$w_{i1}(\text{new}) = w_{i1}(\text{old}) + 0.5[x_i - w_{i1}(\text{old})]$$

$$w_{11}(n) = w_{11}(0) + 0.5[x_1 - w_{11}(0)]$$

$$= 0.2 + 0.5(0 - 0.2) = 0.1$$

$$w_{21}(n) = w_{21}(0) + 0.5[x_2 - w_{21}(0)]$$

$$= 0.4 + 0.5(0 - 0.4) = 0.2$$

$$w_{31}(n) = w_{31}(0) + 0.5[x_3 - w_{31}(0)]$$

$$= 0.6 + 0.5(1 - 0.6) = 0.8$$

$$w_{41}(n) = w_{41}(0) + 0.5[x_4 - w_{41}(0)]$$

$$= 0.8 + 0.5(1 - 0.8) = 0.9$$

The updated weight matrix after presentation of first input pattern is

$$w_{ij} = \begin{bmatrix} 0.1 & 0.9 \\ 0.2 & 0.7 \\ 0.8 & 0.5 \\ 0.9 & 0.3 \end{bmatrix}$$

Second input vector:

Step 1: For $x = [1 \ 0 \ 0 \ 0]$, perform Steps 2–4.

Step 2: Calculate the Euclidean distance:

$$D(j) = \sum_i (w_{ij} - x_i)^2$$

$$D(1) = \sum_{i=1}^{4} (w_{i1} - x_i)^2$$

$$= (0.1 - 1)^2 + (0.2 - 0)^2 + (0.8 - 0)^2 + (0.9 - 0)^2$$
$$= 0.81 + 0.04 + 0.64 + 0.81$$
$$= 2.3$$

$$D(2) = \sum_{i=1}^{4} (w_{i2} - x_i)^2$$
$$= (0.9 - 1)^2 + (0.7 - 0)^2 + (0.5 - 0)^2 + (0.3 - 0)^2$$
$$= 0.01 + 0.49 + 0.25 + 0.09$$
$$= 0.84$$

Step 3: Since $D(2) < D(1)$, therefore $D(2)$ is minimum. Hence the winning cluster unit is Y_2, i.e., $J = 2$.

Step 4: Update the weights on the winning cluster unit $J = 2$:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$
$$w_{i2}(\text{new}) = w_{i2}(\text{old}) + 0.5\,[x_i - w_{i2}(\text{old})]$$
$$w_{12}(n) = w_{12}(0) + 0.5\,[x_1 - w_{12}(0)] = 0.9 + 0.5(1 - 0.9) = 0.95$$
$$w_{22}(n) = w_{22}(0) + 0.5\,[x_2 - w_{22}(0)] = 0.7 + 0.5(0 - 0.7) = 0.35$$
$$w_{32}(n) = w_{32}(0) + 0.5\,[x_3 - w_{32}(0)] = 0.5 + 0.5(0 - 0.5) = 0.25$$
$$w_{42}(n) = w_{42}(0) + 0.5\,[x_4 - w_{42}(0)] = 0.3 + 0.5(0 - 0.3) = 0.15$$

The updated weight matrix after presentation of second input pattern is

$$w_{ij} = \begin{bmatrix} 0.1 & 0.95 \\ 0.2 & 0.35 \\ 0.8 & 0.25 \\ 0.9 & 0.15 \end{bmatrix}$$

Third input vector:

Step 1: For $x = [0\ 1\ 1\ 0]$, perform Steps 2–4.

Step 2: Calculate the Euclidean distance:

$$D(j) = \sum_i (w_{ij} - x_i)^2$$
$$D(1) = \sum_{i=1}^{4} (w_{i1} - x_i)^2 = (0.1 - 0)^2 + (0.2 - 1)^2 + (0.8 - 1)^2 + (0.9 - 0)^2 = 0.01 + 0.64 + 0.04 + 0.81 = 1.5$$
$$D(2) = \sum_{i=1}^{4} (w_{i2} - x_i)^2 = (0.95 - 0)^2 + (0.35 - 1)^2 + (0.25 - 1)^2 + (0.15 - 0)^2 = (0.9025) + (0.4225) + (0.5625) + (0.0225) = 1.91$$

Step 3: Since $D(1) < D(2)$, therefore $D(1)$ is minimum. Hence the winning cluster unit is Y_1, i.e., $J = 1$.

Step 4: Update the weights on the winning cluster unit $J = 1$:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$
$$w_{i1}(\text{new}) = w_{i1}(\text{old}) + 0.5\,[x_i - w_{i1}(\text{old})]$$
$$w_{11}(n) = w_{11}(0) + 0.5\,[x_1 - w_{11}(0)] = 0.1 + 0.5(0 - 0.1) = 0.05$$
$$w_{21}(n) = w_{21}(0) + 0.5\,[x_2 - w_{21}(0)] = 0.2 + 0.5(1 - 0.2) = 0.6$$
$$w_{31}(n) = w_{31}(0) + 0.5\,[x_3 - w_{31}(0)] = 0.8 + 0.5(1 - 0.8) = 0.9$$
$$w_{41}(n) = w_{41}(0) + 0.5\,[x_4 - w_{41}(0)] = 0.9 + 0.5(0 - 0.9) = 0.45$$

The weight update after presentation of third input pattern is

$$w_{ij} = \begin{bmatrix} 0.05 & 0.95 \\ 0.6 & 0.35 \\ 0.9 & 0.25 \\ 0.45 & 0.15 \end{bmatrix}$$

Fourth input vector:

Step 1: For $x = [0\ 0\ 0\ 1]$, perform Steps 2–4.

Step 2: Compute the Euclidean distance:

$$D(j) = \sum_{i=1}^{4} (w_{ij} - x_i)^2$$
$$D(1) = \sum_{i=1}^{4} (w_{i1} - x_i)^2 = (0.05 - 0)^2 + (0.6 - 0)^2 + (0.9 - 0)^2 + (0.45 - 1)^2 = 0.0025 + 0.36 + 0.81 + 0.3025 = 1.475$$
$$D(2) = \sum_{i=1}^{4} (w_{i2} - x_i)^2 = (0.95 - 0)^2 + (0.35 - 0)^2 + (0.25 - 0)^2 + (0.15 - 1)^2 = (0.9025) + (0.1225) + (0.0625) + (0.7225) = 1.81$$

Step 3: Since $D(1) < D(2)$, therefore $D(1)$ is minimum. Hence the winning cluster unit is Y_1, i.e., $J = 1$.

Step 4: Update the weights on the winning cluster unit $J = 1$:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$
$$w_{i1}(\text{new}) = w_{i1}(\text{old}) + 0.5\,[x_i - w_{i1}(\text{old})]$$
$$w_{11}(n) = w_{11}(0) + 0.5\,[x_1 - w_{11}(0)] = 0.05 + 0.5(0 - 0.05) = 0.025$$
$$w_{21}(n) = w_{21}(0) + 0.5\,[x_2 - w_{21}(0)] = 0.6 + 0.5(0 - 0.6) = 0.3$$
$$w_{31}(n) = w_{31}(0) + 0.5\,[x_3 - w_{31}(0)] = 0.9 + 0.5(0 - 0.9) = 0.45$$
$$w_{41}(n) = w_{41}(0) + 0.5\,[x_4 - w_{41}(0)] = 0.45 + 0.5(1 - 0.95) = 0.475$$

The final weight obtained after the presentation of fourth input pattern is

$$w_{ij} = \begin{bmatrix} 0.025 & 0.95 \\ 0.3 & 0.35 \\ 0.45 & 0.25 \\ 0.475 & 0.15 \end{bmatrix}$$

Since all the four given input patterns are presented, this is end of first iteration or 1-epoch. Now the learning rate can be updated as

$$\alpha(t + 1) = 0.5\,\alpha(t)$$
$$\alpha(1) = 0.5\,\alpha(0) = 0.5 \times 0.5 = 0.25$$

With this learning rate you can proceed further up to 100 iterations or till radius becomes zero or the weight matrix reduces to a very negligible value. The net with updated weights is shown by Figure 3.

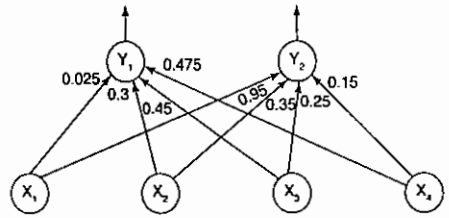


Figure 3 Net for problem 3.

4. For a given Kohonen self-organizing feature map with weights shown in Figure 4: (a) Use the square of the Euclidean distance to find the cluster unit Y_J closest to the input vector (0.2, 0.4). Using a learning rate of 0.2, find the new weights for unit Y_J. (b) For the input vector (0.6, 0.6) with learning rate 0.1, find the winning cluster unit and its new weights.

Solution: (a) For the input vector $(0.2, 0.4) = (x_1, x_2)$ and $\alpha = 0.2$, the weight vector W is given by

$$W = \begin{bmatrix} 0.3 & 0.2 & 0.1 & 0.8 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.9 & 0.2 \end{bmatrix}$$

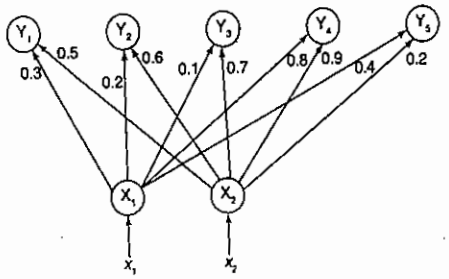


Figure 4 KSOFM net for problem 4.

Now we find the winner unit using square of Euclidean distance, i.e.,

$$D(j) = \sum_{i=1}^{2} (w_{ij} - x_i)^2 = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

For $j = 1$ to 5

$$\begin{aligned} D(1) &= (0.3 - 0.2)^2 + (0.5 - 0.4)^2 \\ &= 0.01 + 0.01 = 0.02 \\ D(2) &= (0.2 - 0.2)^2 + (0.6 - 0.4)^2 \\ &= 0 + 0.4 = 0.04 \\ D(3) &= (0.1 - 0.2)^2 + (0.7 - 0.4)^2 \\ &= 0.01 + 0.09 = 0.01 \\ D(4) &= (0.8 - 0.2)^2 + (0.9 - 0.4)^2 \\ &= 0.36 + 0.25 = 0.61 \\ D(5) &= (0.4 - 0.2)^2 + (0.2 - 0.4)^2 \\ &= 0.04 + 0.04 = 0.08 \end{aligned}$$

Since $D(1) = 0.2$ is the minimum value, the winner unit is $J = 1$. We now update the weights on the winner unit $J = 1$. The weight updation formula is given by

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$

Substituting $J = 1$ in the equation above, we obtain

$$w_{i1}(\text{new}) = w_{i1}(\text{old}) + \alpha[x_i - w_{i1}(\text{old})]$$

For $i = 1$ to 2,

$$\begin{aligned} w_{11}(n) &= w_{11}(0) + \alpha[x_1 - w_{11}(0)] \\ &= 0.3 + 0.2(0.2 - 0.3) = 0.28 \\ w_{21}(n) &= w_{21}(0) + \alpha[x_2 - w_{21}(0)] \\ &= 0.5 + 0.2(0.4 - 0.5) = 0.48 \end{aligned}$$

The updated weight matrix is given by

$$\mathbf{W} = \begin{bmatrix} 0.28 & 0.2 & 0.1 & 0.8 & 0.4 \\ 0.48 & 0.6 & 0.7 & 0.9 & 0.2 \end{bmatrix}$$

- For the input vector $(x_1, x_2) = (0.6, 0.6)$ and $\alpha = 0.1$, the weight matrix is initialized from Figure 4 as

$$\mathbf{W} = \begin{bmatrix} 0.3 & 0.2 & 0.1 & 0.8 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.9 & 0.2 \end{bmatrix}$$

Now we find the winner unit using square of Euclidean distance, i.e.,

$$D(j) = \sum_{i=1}^{2} (w_{ij} - x_i)^2 = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

For $j = 1$ to 5

$$\begin{aligned} D(1) &= (0.3 - 0.6)^2 + (0.5 - 0.6)^2 \\ &= 0.09 + 0.01 = 0.1 \\ D(2) &= (0.2 - 0.6)^2 + (0.6 - 0.6)^2 \\ &= 0.08 + 0 = 0.08 \\ D(3) &= (0.1 - 0.6)^2 + (0.7 - 0.6)^2 \\ &= 0.25 + 0.01 = 0.26 \\ D(4) &= (0.8 - 0.6)^2 + (0.9 - 0.6)^2 \\ &= 0.04 + 0.09 = 0.13 \\ D(5) &= (0.4 - 0.6)^2 + (0.2 - 0.6)^2 \\ &= 0.04 + 0.16 = 0.2 \end{aligned}$$

Since $D(2) = 0.08$ is the minimum value, the winner unit is $J = 2$. We now update the weights on the winner unit with $\alpha = 0.1$. The weight updation formula is given by

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$

Substituting $J = 2$ in the equation above, we obtain

$$w_{i2}(\text{new}) = w_{i2}(\text{old}) + \alpha[x_i - w_{i2}(\text{old})]$$

For $i = 1$ to 2,

$$\begin{aligned} w_{12}(n) &= w_{12}(0) + \alpha[x_1 - w_{12}(0)] \\ &= 0.2 + 0.1(0.6 - 0.2) = 0.24 \\ w_{22}(n) &= w_{22}(0) + \alpha[x_2 - w_{22}(0)] \\ &= 0.6 + 0.1(0.6 - 0.6) = 0.6 \end{aligned}$$

The new weight matrix is given by

$$\mathbf{W} = \begin{bmatrix} 0.3 & 0.24 & 0.1 & 0.8 & 0.4 \\ 0.5 & 0.6 & 0.7 & 0.9 & 0.2 \end{bmatrix}$$

5. Consider a Kohonen self-organizing net with two cluster units and five input units. The weight vectors for the cluster units are given by

$$w_1 = [1.0\ 0.9\ 0.7\ 0.5\ 0.3]$$
$$w_2 = [0.3\ 0.5\ 0.7\ 0.9\ 1.0]$$

Use the square of the Euclidean distance to find the winning cluster unit for the input pattern $x = [0.0\ 0.5\ 1.0\ 0.5\ 0.0]$. Using a learning rate of 0.25, find the new weights for the winning unit.

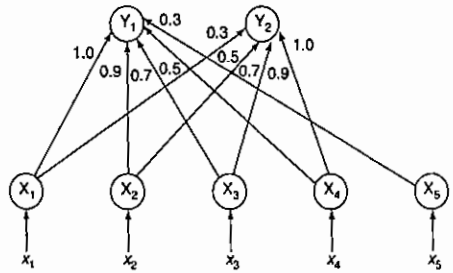


Figure 5 KSOFM net.

Solution: The net can be formed as shown in Figure 5. For the input vector $x = [0.0\ 0.5\ 1.0\ 0.5\ 0.0]$ and the learning rate $\alpha = 0.25$, the weight vector W is given by

$$\mathbf{W} = \begin{bmatrix} 1.0 & 0.3 \\ 0.9 & 0.5 \\ 0.7 & 0.7 \\ 0.5 & 0.9 \\ 0.3 & 1.0 \end{bmatrix}$$

Now we find the winner unit using square of Euclidean distance, i.e.,

$$D(j) = \sum_{i} (w_{ij} - x_i)^2$$

For $i = 1$ to 5 and $j = 1$ to 2,

$$\begin{aligned} D(1) &= (1 - 0)^2 + (0.9 - 0.5)^2 + (0.7 - 1)^2 \\ &\quad + (0.5 - 0.5)^2 + (0.3 - 0)^2 \\ &= 1 + 0.16 + 0.09 + 0 + 0.09 = 1.34 \\ D(2) &= (0.3 - 0)^2 + (0.5 - 0.5)^2 + (0.7 - 1)^2 \\ &\quad + (0.9 - 0.5)^2 + (1 - 0)^2 \\ &= 0.09 + 0 + 0.09 + 0.16 + 1 = 1.34 \end{aligned}$$

As we can see, in this case $D(1) = D(2)$, so the winner unit is the one with the smallest index. Thus, winner unit is Y_1, i.e., $J = 1$. We now update the weights on the winner unit with $\alpha = 0.25$. The weight updation formula is given by

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$

Substituting $J = 1$ in the equation above, we obtain

$$w_{i1}(\text{new}) = w_{i1}(\text{old}) + \alpha[x_i - w_{i1}(\text{old})]$$

For $i = 1$ to 5,

$$\begin{aligned} w_{11}(n) &= w_{11}(0) + \alpha[x_1 - w_{11}(0)] \\ &= 1 + 0.25(0 - 1) = 0.75 \\ w_{21}(n) &= w_{21}(0) + \alpha[x_2 - w_{21}(0)] \\ &= 0.9 + 0.25(0.5 - 0.9) = 0.8 \\ w_{31}(n) &= w_{31}(0) + \alpha[x_3 - w_{31}(0)] \\ &= 0.7 + 0.25(1 - 0.7) = 0.775 \\ w_{41}(n) &= w_{41}(0) + \alpha[x_4 - w_{41}(0)] \\ &= 0.5 + 0.25(0.5 - 0.5) = 0.5 \\ w_{51}(n) &= w_{51}(0) + \alpha[x_5 - w_{51}(0)] \\ &= 0.3 + 0.25(0 - 0.3) = 0.225 \end{aligned}$$

The updated weight matrix for the winning unit is given by

$$\mathbf{W} = \begin{bmatrix} 0.75 & 0.3 \\ 0.8 & 0.5 \\ 0.775 & 0.7 \\ 0.5 & 0.9 \\ 0.225 & 1.0 \end{bmatrix}$$

6. Construct and test an LVQ net with five vectors assigned to two classes. The given vectors along with the classes are as shown in Table 1.

Table 1

Vector	Class
[0 0 1 1]	1
[1 0 0 0]	2
[0 0 0 1]	2
[1 1 0 0]	1
[0 1 1 0]	1

Solution: In the given five vectors, first two vectors are used as initial weight vectors and the remaining three vectors are used as input vectors. Based on this, LVQ net is shown in Figure 6, along with initial weights.

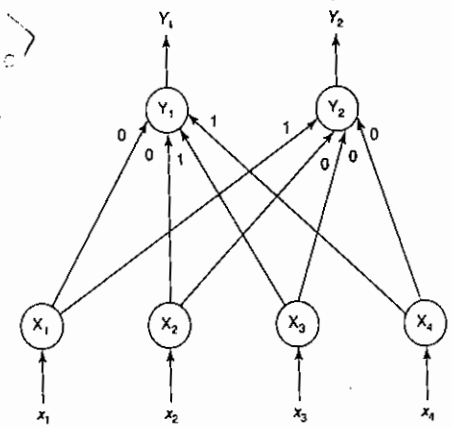


Figure 6 LVQ net.

Initialize the reference weight vectors as

$$w_1 = [0\ \ 0\ \ 1\ \ 1]; \quad w_2 = [1\ \ 0\ \ 0\ \ 0]$$

Let the learning rate be $\alpha = 0.1$.

First input vector

For [0 0 0 1] with $T = 2$, calculate the square of the Euclidean distance, i.e.,

$$D(j) = \sum_{i=1}^{4} (w_{ij} - x_i)^2$$

For $j = 1$ to 2,

$$D(1) = (0-0)^2 + (0-0)^2 + (1-0)^2 + (1-1)^2 = 1$$
$$D(2) = (1-0)^2 + (0-0)^2 + (0-0)^2 + (0-1)^2 = 2$$

Since $D(1) < D(2), D(1)$ is minimum; hence the winner unit index is $J = 1$. Now that $T \neq J$, the weight updation is performed as

$$w_J(\text{new}) = w_J(\text{old}) - \alpha[x - w_J(\text{old})]$$
$$w_{11}(n) = w_{11}(0) - \alpha[x_1 - w_{11}(0)] = 0 - 0.1(0-0) = 0$$
$$w_{21}(n) = w_{21}(0) - \alpha[x_2 - w_{21}(0)] = 0 - 0.1(0-0) = 0$$
$$w_{31}(n) = w_{31}(0) - \alpha[x_3 - w_{31}(0)] = 1 - 0.1(0-1) = 1.1$$
$$w_{41}(n) = w_{41}(0) - \alpha[x_4 - w_{41}(0)] = 1 - 0.1(1-1) = 1$$

After the presentation of first input pattern, the weight matrix becomes

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \\ 1.1 & 0 \\ 1 & 0 \end{bmatrix}$$

Second input vector

For [1 1 0 0] with $T = 1$, calculate the square of the Euclidean distance, i.e.,

$$D(j) = \sum_{i=1}^{4} (w_{ij} - x_i)^2$$

For $j = 1$ to 2,

$$D(1) = (0-1)^2 + (0-1)^2 + (1.1-0)^2 + (1-0)^2 = 4.21$$
$$D(2) = (1-1)^2 + (0-1)^2 + (0-0)^2 + (0-0)^2 = 1$$

Since $D(2) < D(1), D(2)$ is minimum; hence the winner unit index is $J = 2$. Again since $T \neq J$, the weight updation is performed as

$$w_J(\text{new}) = w_J(\text{old}) - \alpha[x - w_J(\text{old})]$$
$$w_{12}(n) = w_{12}(0) - \alpha[x_1 - w_{12}(0)] = 1 - 0.1(1-1) = 1$$
$$w_{22}(n) = w_{22}(0) - \alpha[x_2 - w_{22}(0)] = 0 - 0.1(1-0) = -0.1$$
$$w_{32}(n) = w_{32}(0) - \alpha[x_3 - w_{32}(0)] = 0 - 0.1(0-0) = 0$$
$$w_{42}(n) = w_{42}(0) - \alpha[x_4 - w_{42}(0)] = 0 - 0.1(0-0) = 0$$

After the presentation of second input pattern, the weight matrix becomes

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 0 & -0.1 \\ 1.1 & 0 \\ 1 & 0 \end{bmatrix}$$

Third input vector

For [0 1 1 0] with $T = 1$, calculate the square of the Euclidean distance as

$$D(j) = \sum_{i=1}^{4} (w_{ij} - x_i)^2$$

For $j = 1$ to 2,

$$D(1) = (0-0)^2 + (0-1)^2 + (1.1-1)^2 + (1-0)^2 = 2.01$$
$$D(2) = (1-0)^2 + (-0.1-1)^2 + (0-1)^2 + (0-0)^2 = 3.21$$

Since $D(1) < D(2), D(1)$ is minimum; hence the winner unit index is $J = 1$. Now that $T = J$, the weight updation is performed as

$$w_J(\text{new}) = w_J(\text{old}) + \alpha[x - w_J(\text{old})]$$

Updating the weights on the winner unit, we obtain

$$w_{11}(n) = w_{11}(0) + \alpha[x_1 - w_{11}(0)] = 0 + 0.1(0-0) = 0$$
$$w_{21}(n) = w_{21}(0) + \alpha[x_2 - w_{21}(0)] = 0 + 0.1(1-0) = 0.1$$
$$w_{31}(n) = w_{31}(0) + \alpha[x_3 - w_{31}(0)] = 1.1 + 0.1(1-1.1) = 1.09$$
$$w_{41}(n) = w_{41}(0) + \alpha[x_4 - w_{41}(0)] = 1 + 0.1(0-1) = 0.9$$

After the presentation of third input pattern, the weight matrix becomes

$$\mathbf{W} = \begin{bmatrix} 0 & 1 \\ 0.1 & -0.1 \\ 1.09 & 0 \\ 0.9 & 0 \end{bmatrix}$$

Thus the first epoch of the training has been completed. It is noted that if correct class is obtained for first and second input patterns, further epochs can be performed until all the winner units become equal to all the classes, i.e., all $T = J$.

7. Consider an LVQ net with two input units and four target classes: c_1, c_2, c_3 and c_4. There exist 16 classification units, with weight vectors indicated by the coordinates on the following chart, read in row–column order. For example, the unit with weight vector (0.2, 0.2), (0.2, 0.6) is assigned to represent class 1 and the classification units for class 2 have initial weight vectors of (0.4, 0.2), (0.4, 0.6), (0.8, 0.4) and (0.8, 0.8). The chart is given in Table 2.

Table 2

x_2						
1.0						
0.8	c_3	c_4	c_1	c_2		
0.6	c_1	c_2	c_3	c_4		
0.4	c_3	c_4	c_1	c_2		
0.2	c_1	c_2	c_3	c_4		
0.0						
0.0	0.2	0.4	0.6	0.8	1.0	x_1

Use square of Euclidean distance to measure the changes occurring.

- Given an input vector of (0.25, 0.25) representing class 1 and using a learning rat

of $\alpha = 0.25$, show which classification unit moves where (i.e., determine its new weight vector).

- Given the input vector of (0.4, 0.35) representing class 1, using initial weight vector and learning rate of $\alpha = 0.25$, note what happens?
- Given the input vector of (0.4, 0.45), determine the performance of the net. The input vector represents class 1.

Solution: The LVQ net for this problem with two input units and four cluster units is shown in Figure 7. The initial weight vectors for the respective classes are shown below.

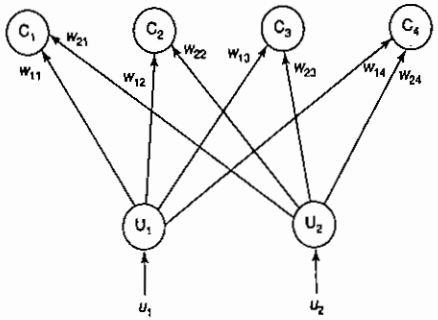


Figure 7 LVQ net (with two input units, four cluster units).

Class 1:

Initial weight vector

$$W_1 = \begin{bmatrix} 0.2 & 0.2 & 0.6 & 0.6 \\ 0.2 & 0.6 & 0.8 & 0.4 \end{bmatrix}$$

with target $t = 1$.

Class 2:

Initial weight vector

$$W_2 = \begin{bmatrix} 0.4 & 0.4 & 0.8 & 0.8 \\ 0.2 & 0.6 & 0.8 & 0.4 \end{bmatrix}$$

with target $t = 2$.

Class 3:

Initial weight vector

$$W_3 = \begin{bmatrix} 0.2 & 0.2 & 0.6 & 0.6 \\ 0.4 & 0.8 & 0.6 & 0.2 \end{bmatrix}$$

with target $t = 3$.

Class 4:

Initial weight vector

$$W_4 = \begin{bmatrix} 0.4 & 0.4 & 0.8 & 0.8 \\ 0.4 & 0.8 & 0.6 & 0.2 \end{bmatrix}$$

with target $t = 4$.

- For the given input vector $(u_1, u_2) = (0.25, 0.25)$ with $\alpha = 0.25$ and $t = 1$, we calculate the square of the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{2} (w_{ij} - x_i)^2 = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

For $j = 1$ to 4,

$$D(1) = (0.2 - 0.25)^2 + (0.2 - 0.25)^2 = 0.005$$
$$D(2) = (0.2 - 0.25)^2 + (0.6 - 0.25)^2 = 0.125$$
$$D(3) = (0.6 - 0.25)^2 + (0.8 - 0.25)^2 = 0.425$$
$$D(4) = (0.6 - 0.25)^2 + (0.4 - 0.25)^2 = 0.145$$

As $D(1)$ is minimum, therefore the winner unit index is $J = 1$. Now we update the weights on the winner unit, since $t = J = 1, \alpha = 0.25$, using the weight updation formula

$$w_J(\text{new}) = w_J(\text{old}) + \alpha[x - w_J(\text{old})]$$

Updating the weights on the winner unit, we obtain

$$w_{11}(\text{new}) = w_{11}(0) + \alpha[x_1 - w_{11}(\text{old})]$$
$$= 0.2 + 0.25(0.25 - 0.2) = 0.2125$$
$$w_{21}(\text{new}) = w_{21}(0) + \alpha[x_2 - w_{21}(\text{old})]$$
$$= 0.2 + 0.25(0.25 - 0.2) = 0.2125$$

Therefore, the new weight vector is

$$W_1 = \begin{bmatrix} 0.2125 & 0.2 & 0.6 & 0.6 \\ 0.2125 & 0.6 & 0.8 & 0.4 \end{bmatrix}$$

- For the given input vector $(u_1, u_2) = (0.4, 0.35)$ with $\alpha = 0.25$ and $t = 1$, we calculate the square of the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{2} (w_{ij} - x_i)^2 = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

For $j = 1$ to 4,

$$D(1) = (0.2 - 0.4)^2 + (0.2 - 0.35)^2 = 0.0625$$
$$D(2) = (0.2 - 0.4)^2 + (0.6 - 0.35)^2 = 0.1025$$
$$D(3) = (0.6 - 0.4)^2 + (0.8 - 0.35)^2 = 0.2425$$
$$D(4) = (0.6 - 0.4)^2 + (0.4 - 0.35)^2 = 0.0425$$

As $D(4)$ is minimum, therefore the winner unit index is $J = 4$. Thus, fourth unit is the winner unit that is closest to the input vector. Since $t \neq J$, the weight updation formula used is

$$w_J(\text{new}) = w_J(\text{old}) - \alpha[x - w_J(\text{old})]$$

Updating the weights on the winner unit, we obtain

$$w_{14}(\text{new}) = w_{14}(0) - \alpha[x_1 - w_{14}(\text{old})]$$
$$= 0.6 - 0.25(0.4 - 0.6) = 0.65$$
$$w_{24}(\text{new}) = w_{24}(0) - \alpha[x_2 - w_{24}(\text{old})]$$
$$= 0.4 - 0.25(0.35 - 0.4) = 0.4125$$

Therefore, the new weight vector is

$$W_1 = \begin{bmatrix} 0.2 & 0.2 & 0.6 & 0.65 \\ 0.2 & 0.6 & 0.8 & 0.4125 \end{bmatrix}$$

- For the given input vector $(u_1, u_2) = (0.4, 0.45)$ with $\alpha = 0.25$ and $t = 1$, we calculate the square of the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{2} (w_{ij} - x_i)^2 = (w_{1j} - x_1)^2 + (w_{2j} - x_2)^2$$

For $j = 1$ to 4,

$$D(1) = (0.2 - 0.4)^2 + (0.2 - 0.45)^2 = 0.1025$$
$$D(2) = (0.2 - 0.4)^2 + (0.6 - 0.45)^2 = 0.0625$$
$$D(3) = (0.6 - 0.4)^2 + (0.8 - 0.45)^2 = 0.1625$$
$$D(4) = (0.6 - 0.4)^2 + (0.4 - 0.45)^2 = 0.0425$$

As $D(4)$ is minimum, therefore in this case also the winner unit index is $J = 4$. Since $t \neq J$, the weight updation formula used is

$$w_J(\text{new}) = w_J(\text{old}) - \alpha[x - w_J(\text{old})]$$

Updating the weights on the winner unit, we obtain

$$w_{14}(\text{new}) = w_{14}(0) - \alpha[x_1 - w_{14}(\text{old})]$$
$$= 0.6 - 0.25(0.4 - 0.6) = 0.65$$
$$w_{24}(\text{new}) = w_{24}(0) - \alpha[x_2 - w_{24}(\text{old})]$$
$$= 0.4 - 0.25(0.45 - 0.4) = 0.3875$$

Therefore, the new weight vector is

$$W_1 = \begin{bmatrix} 0.2 & 0.2 & 0.6 & 0.65 \\ 0.2 & 0.6 & 0.8 & 0.3875 \end{bmatrix}$$

8. Consider the following full CPN shown in Figure 8. Using the input pair $x = [1\ 0\ 0\ 0]$ and $y = [1\ 0]$, perform the phase I of training (one step only). Find the activation of the cluster layer units and update the weights using learning rates $\alpha = \beta = 0.2$.

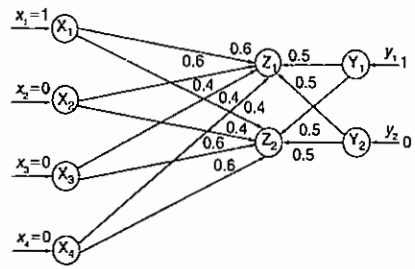


Figure 8 Instar model of CPN net.

Solution: The input pair is $x = [1\ 0\ 0\ 0]$ and $y = [1\ 0]$ and the learning rates are $\alpha = 0.2$ and $\beta = 0.2$.

Phase I of training: The initial weights are obtained from Figure 8 as

$$V = \begin{bmatrix} 0.6 & 0.4 \\ 0.6 & 0.4 \\ 0.4 & 0.6 \\ 0.4 & 0.6 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

Now we calculate the square of the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{4}(x_i - v_{ij})^2 + \sum_{k=1}^{2}(y_k - w_{kj})^2$$

For $j = 1$ to 2,

$$\begin{aligned} D(1) &= \sum_{i=1}^{4}(x_i - v_{i1})^2 + \sum_{k=1}^{2}(y_k - w_{k1})^2 \\ &= (x_1 - v_{11})^2 + (x_2 - v_{21})^2 + (x_3 - v_{31})^2 \\ &\quad + (x_4 - v_{41})^2 + (y_1 - w_{11})^2 + (y_2 - w_{21})^2 \\ &= (1 - 0.6)^2 + (0 - 0.6)^2 + (0 - 0.4)^2 \\ &\quad + (0 - 0.4)^2 + (1 - 0.5)^2 + (0 - 0.5)^2 \\ &= 0.16 + 0.36 + 0.16 + 0.16 + 0.25 + 0.25 \\ D(1) &= 1.34 \end{aligned}$$

$$\begin{aligned} D(2) &= \sum_{i=1}^{4}(x_i - v_{i2})^2 + \sum_{k=1}^{2}(y_k - w_{k2})^2 \\ &= (x_1 - v_{12})^2 + (x_2 - v_{22})^2 + (x_3 - v_{32})^2 \\ &\quad + (x_4 - v_{42})^2 + (y_1 - w_{12})^2 + (y_2 - w_{22})^2 \\ &= (1 - 0.4)^2 + (0 - 0.4)^2 + (0 - 0.6)^2 \\ &\quad + (0 - 0.6)^2 + (1 - 0.5)^2 + (0 - 0.5)^2 \\ &= 0.36 + 0.16 + 0.36 + 0.36 + 0.25 + 0.25 \\ D(2) &= 1.74 \end{aligned}$$

Since, $D(1) < D(2)$, therefore the winner unit index is $J = 1$. We now update the weights on the winner unit.

Weight updation: The weight updation between the x-input and cluster layer is performed as shown below:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

For $i = 1$ to 4 and $J = 1$, we obtain

$$\begin{aligned} v_{11}(\text{new}) &= v_{11}(0) + \alpha[x_1 - v_{11}(\text{old})] \\ &= 0.6 + 0.2(1 - 0.6) = 0.68 \\ v_{21}(\text{new}) &= v_{21}(0) + \alpha[x_2 - v_{21}(\text{old})] \\ &= 0.6 + 0.2(0 - 0.6) = 0.48 \\ v_{31}(\text{new}) &= v_{31}(0) + \alpha[x_3 - v_{31}(\text{old})] \\ &= 0.4 + 0.2(0 - 0.4) = 0.32 \\ v_{41}(\text{new}) &= v_{41}(0) + \alpha[x_4 - v_{41}(\text{old})] \\ &= 0.4 + 0.2(0 - 0.4) = 0.32 \end{aligned}$$

The weight updation between the y-input and cluster layer is performed as shown below:

$$w_{kj}(\text{new}) = w_{kj}(\text{old}) + \beta[y_k - w_{kj}(\text{old})]$$

For $k = 1$ to 2 and $J = 1$, we obtain

$$\begin{aligned} w_{11}(\text{new}) &= w_{11}(0) + \beta[y_1 - w_{11}(\text{old})] \\ &= 0.5 + 0.2(1 - 0.5) = 0.6 \\ w_{21}(\text{new}) &= w_{21}(0) + \beta[y_2 - w_{21}(\text{old})] \\ &= 0.5 + 0.2(0 - 0.5) = 0.4 \end{aligned}$$

Thus the updated weights are

$$V = \begin{bmatrix} 0.68 & 0.4 \\ 0.48 & 0.4 \\ 0.32 & 0.6 \\ 0.32 & 0.6 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{bmatrix}$$

9. Consider the CPN net shown in Figure 9. Using the input pair $x = [0\ 1\ 1\ 0]$ and $y = [0\ 1]$, perform phase I of training (one step only). Find the activation of the cluster layer units and update the weights using learning rates $\alpha = 0.2$ and $\beta = 0.3$.

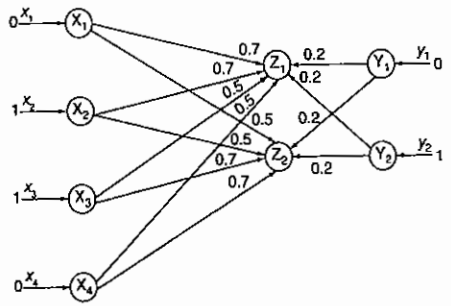


Figure 9 Instar model of CPN net.

Solution: The input pair is $x = [0\ 1\ 1\ 0]$, $y = [0\ 1]$ and learning rates are $\alpha = 0.2$ and $\beta = 0.3$.

Phase I of training: The initial weights obtained from Figure 9 are

$$V = \begin{bmatrix} 0.7 & 0.5 \\ 0.7 & 0.5 \\ 0.5 & 0.7 \\ 0.5 & 0.7 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.2 & 0.2 \\ 0.2 & 0.2 \end{bmatrix}$$

Now we calculate the square of the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{4}(x_i - v_{ij})^2 + \sum_{k=1}^{2}(y_k - w_{kj})^2$$

For $j = 1$ to 2,

$$\begin{aligned} D(1) &= \sum_{i=1}^{4}(x_i - v_{i1})^2 + \sum_{k=1}^{2}(y_k - w_{k1})^2 \\ &= (0 - 0.7)^2 + (1 - 0.7)^2 + (1 - 0.5)^2 \\ &\quad + (0 - 0.5)^2 + (0 - 0.2)^2 + (1 - 0.2)^2 \\ &= 0.49 + 0.09 + 0.25 + 0.25 + 0.04 + 0.64 \\ D(1) &= 1.76 \end{aligned}$$

$$\begin{aligned} D(2) &= \sum_{i=1}^{4}(x_i - v_{i2})^2 + \sum_{k=1}^{2}(y_k - w_{k2})^2 \\ &= (0 - 0.5)^2 + (1 - 0.5)^2 + (1 - 0.7)^2 \\ &\quad + (0 - 0.7)^2 + (0 - 0.2)^2 + (1 - 0.2)^2 \\ &= 0.25 + 0.25 + 0.09 + 0.49 + 0.04 + 0.64 \\ D(2) &= 1.76 \end{aligned}$$

In this case, $D(1) = D(2) = 1.76$, i.e., both the distances are equal. Hence the unit with the *smallest index* is chosen as the winner and weights are updated, i.e., we take $J = 1$ and update the weights on this winner unit.

Weight updation: The weight updation between the x-input and cluster layer is performed as shown below:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

For $i = 1$ to 4 and $J = 1$, we obtain

$$\begin{aligned} v_{11}(\text{new}) &= v_{11}(0) + \alpha[x_1 - v_{11}(\text{old})] \\ &= 0.7 + 0.2(0 - 0.7) = 0.56 \\ v_{21}(\text{new}) &= v_{21}(0) + \alpha[x_2 - v_{21}(\text{old})] \\ &= 0.7 + 0.2(1 - 0.7) = 0.76 \\ v_{31}(\text{new}) &= v_{31}(0) + \alpha[x_3 - v_{31}(\text{old})] \\ &= 0.5 + 0.2(1 - 0.5) = 0.60 \\ v_{41}(\text{new}) &= v_{41}(0) + \alpha[x_4 - v_{41}(\text{old})] \\ &= 0.5 + 0.2(0 - 0.5) = 0.40 \end{aligned}$$

The weight updation between the Y-input and cluster layer is performed as shown below:

$$w_{kj}(\text{new}) = w_{kj}(\text{old}) + \beta[y_k - w_{kj}(\text{old})]$$

For $k = 1$ to 2 and $J = 1$, we obtain

$$\begin{aligned} w_{11}(\text{new}) &= w_{11}(0) + \beta[y_1 - w_{11}(\text{old})] \\ &= 0.2 + 0.3(0 - 0.2) = 0.14 \\ w_{21}(\text{new}) &= w_{21}(0) + \beta[y_2 - w_{21}(\text{old})] \\ &= 0.2 + 0.3(1 - 0.2) = 0.44 \end{aligned}$$

Thus the updated weights are

$$V = \begin{bmatrix} 0.56 & 0.5 \\ 0.76 & 0.5 \\ 0.6 & 0.7 \\ 0.4 & 0.7 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.14 & 0.2 \\ 0.44 & 0.2 \end{bmatrix}$$

10. Consider the forward-only CPN net shown in Figure 10. Using the input pair $x = [1\ 0\ 0\ 0]$ and $y = [1\ 0]$, perform phases I and II of training and update the weights using learning rates $\alpha = a = 0.2$.

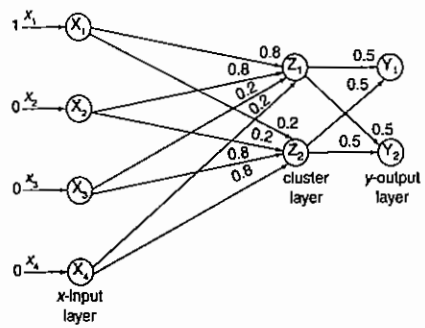


Figure 10 Forward-only CPN network.

Solution: The given input pair is $x = [1\ \ 0\ \ 0\ \ 0]$ and $y = [1\ \ 0]$ with learning rates $\alpha = a = 0.2$. The initial weights obtained from Figure 10 are

$$V = \begin{bmatrix} 0.8 & 0.2 \\ 0.8 & 0.2 \\ 0.2 & 0.8 \\ 0.2 & 0.8 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.5 & 0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

Phase I of training: We calculate the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{4} (x_i - v_{ij})^2$$

For $j = 1$ to 2,

$$\begin{aligned} D(1) &= \sum_{i=1}^{4} (x_i - v_{i1})^2 \\ &= (1 - 0.8)^2 + (0 - 0.8)^2 + (0 - 0.2)^2 \\ &\quad + (0 - 0.2)^2 \\ &= 0.04 + 0.64 + 0.04 + 0.04 \\ D(1) &= 0.76 \end{aligned}$$

$$\begin{aligned} D(2) &= \sum_{i=1}^{4} (x_i - v_{i2})^2 \\ &= (1 - 0.2)^2 + (0 - 0.2)^2 + (0 - 0.8)^2 \\ &\quad + (0 - 0.8)^2 \\ &= 0.64 + 0.04 + 0.64 + 0.64 \\ D(2) &= 1.96 \end{aligned}$$

Since $D(1) < D(2)$, the winner unit index is $J = 1$.

Weight updation: The weight updation on the winner unit is given by

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

For $i = 1$ to 4 and $J = 1$, we obtain

$$\begin{aligned} v_{11}(\text{new}) &= v_{11}(0) + \alpha[x_1 - v_{11}(\text{old})] \\ &= 0.8 + 0.2(1 - 0.8) = 0.84 \\ v_{21}(\text{new}) &= v_{21}(0) + \alpha[x_2 - v_{21}(\text{old})] \\ &= 0.8 + 0.2(0 - 0.8) = 0.64 \\ v_{31}(\text{new}) &= v_{31}(0) + \alpha[x_3 - v_{31}(\text{old})] \\ &= 0.2 + 0.2(0 - 0.2) = 0.16 \\ v_{41}(\text{new}) &= v_{41}(0) + \alpha[x_4 - v_{41}(\text{old})] \\ &= 0.2 + 0.2(0 - 0.2) = 0.16 \end{aligned}$$

The updated weight matrix is,

$$V = \begin{bmatrix} 0.84 & 0.2 \\ 0.64 & 0.2 \\ 0.16 & 0.8 \\ 0.16 & 0.8 \end{bmatrix}$$

Phase II of training: We calculate the Euclidean distance using the formula

$$D(j) = \sum_{i=1}^{4} (x_i - v_{ij})^2$$

For $j = 1$ to 2,

$$\begin{aligned} D(1) &= \sum_{i=1}^{4} (x_i - v_{i1})^2 \\ &= (1 - 0.84)^2 + (0 - 0.64)^2 + (0 - 0.16)^2 \\ &\quad + (0 - 0.16)^2 \\ &= 0.0256 + 0.4096 + 0.0256 + 0.0256 \\ D(1) &= 0.4864 \end{aligned}$$

$$\begin{aligned} D(2) &= \sum_{i=1}^{4} (x_i - v_{i2})^2 \\ &= (1 - 0.2)^2 + (0 - 0.2)^2 + (0 - 0.8)^2 \\ &\quad + (0 - 0.8)^2 \\ &= 0.64 + 0.04 + 0.64 + 0.64 \\ D(2) &= 1.96 \end{aligned}$$

Since $D(1) < D(2)$, the winner unit index is $J = 1$.

Weight updation on winner unit:

- Updating the weights into unit z_j:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \alpha[x_i - v_{ij}(\text{old})]$$

For $i = 1$ to 4 and $J = 1$, we obtain

$$\begin{aligned} v_{11}(\text{new}) &= v_{11}(0) + \alpha[x_1 - v_{11}(\text{old})] \\ &= 0.84 + 0.2(1 - 0.84) = 0.872 \\ v_{21}(\text{new}) &= v_{21}(0) + \alpha[x_2 - v_{21}(\text{old})] \\ &= 0.64 + 0.2(0 - 0.64) = 0.512 \\ v_{31}(\text{new}) &= v_{31}(0) + \alpha[x_3 - v_{31}(\text{old})] \\ &= 0.16 + 0.2(0 - 0.16) = 0.128 \\ v_{41}(\text{new}) &= v_{41}(0) + \alpha[x_4 - v_{41}(\text{old})] \\ &= 0.16 + 0.2(0 - 0.16) = 0.128 \end{aligned}$$

- Updating the weights from unit z_j to output layer:

$$w_{kj}(\text{new}) = w_{kj}(\text{old}) + a[y_k - w_{kj}(\text{old})]$$

For $k = 1$ to 2 and $J = 1$, we obtain

$$\begin{aligned} w_{11}(\text{new}) &= w_{11}(0) + a[y_1 - w_{11}(\text{old})] \\ &= 0.5 + 0.2(1 - 0.5) = 0.6 \\ w_{21}(\text{new}) &= w_{21}(0) + a[y_2 - w_{21}(\text{old})] \\ &= 0.5 + 0.2(0 - 0.5) = 0.4 \end{aligned}$$

Thus, the updated weights are after phase II of training are

$$V = \begin{bmatrix} 0.872 & 0.2 \\ 0.512 & 0.2 \\ 0.128 & 0.8 \\ 0.128 & 0.8 \end{bmatrix} \quad \text{and} \quad W = \begin{bmatrix} 0.6 & 0.5 \\ 0.4 & 0.5 \end{bmatrix}$$

11. Construct an ART 1 network for clustering four input vectors with low vigilance parameter of 0.4 into three clusters. The four input vectors are [0 0 0 1], [0 1 0 1], [0 0 1 1] and [1 0 0 0]. Assume the necessary parameters needed.

Solution: The values assumed in this case are $\rho = 0.4$, $\alpha = 2$. Also it can be noted that $n = 4$ and $m = 3$. Hence,

Bottom-up-weights, $b_{ij}(0) = 1/(1 + n) = 1/1 + 4 = 0.2$.

Top-down-weights $t_{ji}(0) = 1$.

For $i = 1$ to 4 and $j = 1$ to 2,

$$b_{ij} = \begin{bmatrix} 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \\ 0.2 & 0.2 & 0.2 \end{bmatrix}_{4\times 3}$$

$$\text{and} \quad t_{ji} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}_{3\times 4}$$

Step 0: Initialize the parameters:

$$\rho = 0.4; \quad \alpha = 2$$

Initialize weights:

$$b_{ij}(0) = 0.2; \quad t_{ji}(0) = 1$$

Step 1: Start computation.

Step 2: For the first input vector [0 0 0 1], perform Steps 3–12.

Step 3: Set activations of all F_2 units to zero. Set activations of $F_1(a)$ units to input vector $s = [0\ 0\ 0\ 1]$.

Step 4: Compute norm of s:

$$\|s\| = 0 + 0 + 0 + 1 = 1$$

Step 5: Compute activations for each node in the F_1 layer:

$$x = [0\ 0\ 0\ 1]$$

Step 6: Compute net input to each node in the F_2 layer:

$$y_j = \sum_{i=1}^{4} b_{ij} x_i$$

For $j = 1$ to 3,

$$\begin{aligned} y_1 &= 0.2(0) + 0.2(0) + 0.2(0) + 0.2(1) \\ &= 0.2 \\ y_2 &= 0.2(0) + 0.2(0) + 0.2(0) + 0.2(1) \\ &= 0.2 \\ y_3 &= 0.2(0) + 0.2(0) + 0.2(0) + 0.2(1) \\ &= 0.2 \end{aligned}$$

Step 7: When reset is true, perform Steps 8–11.

Step 8: Since all the inputs pose same net input, there exists a tie and the unit with the smallest index is the winner, i.e., $J = 1$.

Step 9: Recompute the F_1 activations (for $J = 1$):

$$x_i = s_i t_{ji}$$

$$x_1 = s_1 t_{j1} = [0\ \ 0\ \ 0\ \ 1][1\ \ 1\ \ 1\ \ 1]$$

$$x_1 = [0\ \ 0\ \ 0\ \ 1]$$

Step 10: Calculate norm of x:

$$\|x\| = 1$$

Step 11: Test for reset condition:

$$\frac{\|x\|}{\|s\|} = \frac{1}{1} = 1.0 \geq 0.4\ (\rho)$$

Hence reset is false. Proceed to Step 12.

Step 12: • Update bottom-up weights for $\alpha = 2$:

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|} = \frac{2x_i}{2 - 1 + \|x\|} = \frac{2x_i}{1 + \|x\|}$$

$$b_{11} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{21} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{31} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{41} = \frac{2 \times 1}{1+1} = \frac{2}{2} = 1$$

Therefore, the bottom-up weight matrix b_{ij} becomes

$$b_{ij} = \begin{bmatrix} 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 1 & 0.2 & 0.2 \end{bmatrix}$$

• Update the top-down weights:

$$t_{ji}(\text{new}) = x_i$$

$$t_{ji} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Steps 0 and 1 remain the same.

Step 2: For the second input vector [0 1 0 1], perform Steps 3–12.

Step 3: Set activations of all F_2 units to zero. Set activations of F_1(a) units to input vector s = [0 1 0 1].

Step 4: Compute norm of s:

$$\|s\| = 0 + 1 + 0 + 1 = 2$$

Step 5: Compute activations of each node in the F_1 layer:

$$x = [0\ 1\ 0\ 1]$$

Step 6: Compute net input to each node in the F_2 layer:

$$y_j = \sum_{i=1}^{4} b_{ij} x_i$$

For j = 1 to 3,

$$y_1 = 0(0) + 0(1) + 0(0) + 1(1) = 1$$

$$y_2 = 0.2(0) + 0.2(1) + 0.2(0) + 0.2(1) = 0.4$$

$$y_3 = 0.2(0) + 0.2(1) + 0.2(0) + 0.2(1) = 0.4$$

Step 7: When reset is true, perform Steps 8–11.

Step 8: The unit with largest net input is the winner, i.e., $J = 1$.

Step 9: Recompute F_1 activations (for $J = 1$):

$$x_i = s_i t_{ji} = [0\ 1\ 0\ 1][0\ 0\ 0\ 1] = [0\ 0\ 0\ 1]$$

Step 10: Calculate norm of x:

$$\|x\| = 0 + 0 + 0 + 1 = 1$$

Step 11: Test for reset condition:

$$\frac{\|x\|}{\|s\|} = \frac{1}{2} = 0.5 \geq 0.4\ (\rho)$$

Hence reset is false. Proceed to Step 12.

Step 12: • Update bottom-up weights for $\alpha = 2$:

$$b_{ij}(\text{new}) = \frac{2x_i}{1 + \|x\|}$$

$$b_{11} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{21} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{31} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{41} = \frac{2 \times 1}{1+1} = \frac{2}{2} = 1$$

Therefore, the bottom-up weight matrix b_{ij} becomes

$$b_{ij} = \begin{bmatrix} 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 1 & 0.2 & 0.2 \end{bmatrix}$$

• Update the top-down weights:

$$t_{ji}(\text{new}) = x_i$$

$$t_{ji} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Steps 0 and 1 remain the same.

Step 2: For the third input vector [0 0 1 1], perform Steps 3–12.

Step 3: Set activations of all F_2 units to zero. Set activations of F_1(a) units to input vector s = [0 0 1 1].

Step 4: Compute norm of s:

$$\|s\| = 0 + 0 + 1 + 1 = 2$$

Step 5: Compute activations of each node in the F_1 layer:

$$x = [0\ 0\ 1\ 1]$$

Step 6: Compute net input to each node in the F_2 layer:

$$y_j = \sum_{i=1}^{4} b_{ij} x_i$$

For j = 1 to 3,

$$y_1 = 0(0) + 0(0) + 0(1) + 1(1) = 1$$

$$y_2 = 0.2(0) + 0.2(0) + 0.2(1) + 0.2(1) = 0.4$$

$$y_3 = 0.2(0) + 0.2(0) + 0.2(1) + 0.2(1) = 0.4$$

Step 7: When reset is true, perform Steps 8–11.

Step 8: The unit with largest net input is the winner, i.e., $J = 1$.

Step 9: Recompute F_1 activations (for $J = 1$):

$$x_i = s_i t_{ji} = [0\ 0\ 1\ 1][0\ 0\ 0\ 1] = [0\ 0\ 0\ 1]$$

Step 10: Calculate norm of x:

$$\|x\| = 0 + 0 + 0 + 1 = 1$$

Step 11: Test for reset condition:

$$\frac{\|x\|}{\|s\|} = \frac{1}{2} = 0.5 \geq 0.4\ (\rho)$$

Hence reset is false. Proceed to Step 12.

Step 12: • Update bottom-up weights for $\alpha = 2$:

$$b_{ij}(\text{new}) = \frac{2x_i}{1 + \|x\|}$$

$$b_{11} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{21} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{31} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{41} = \frac{2 \times 1}{1+1} = \frac{2}{2} = 1$$

Therefore, the bottom-up weight matrix b_{ij} becomes

$$b_{ij} = \begin{bmatrix} 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 0 & 0.2 & 0.2 \\ 1 & 0.2 & 0.2 \end{bmatrix}$$

• Update the top-down weights:

$$t_{ji}(\text{new}) = x_i$$

$$t_{ji} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Steps 0 and 1 remain the same.

Step 2: For the fourth input vector [1 0 0 0], perform Steps 3–12.

Step 3: Set activations of all F_2 units to zero. Set activations of F_1(a) units to input vector $s = [1\ 0\ 0\ 0]$.

Step 4: Compute norm of s:

$$\|s\| = 1 + 0 + 0 + 0 = 1$$

Step 5: Compute activations of each node in the F_1 layer:

$$x = [1\ 0\ 0\ 0]$$

Step 6: Compute net input to each node in the F_2 layer:

$$y_j = \sum_{i=1}^{4} b_{ij} x_i$$

For $j = 1$ to 3,

$$y_1 = 0(1) + 0(0) + 0(0) + 1(0) = 0$$
$$y_2 = 0.2(1) + 0.2(0) + 0.2(0) + 0.2(0) = 0.2$$
$$y_3 = 0.2(1) + 0.2(0) + 0.2(0) + 0.2(0) = 0.2$$

Step 7: When reset is true, perform Steps 8–11.

Step 8: The unit with largest net input is the winner, i.e., $J = 2$.

Step 9: Recompute F_1 activations:

$$x_i = s_i t_{ji} = [1\ 0\ 0\ 0][1\ 1\ 1\ 1] = [1\ 0\ 0\ 0]$$

Step 10: Calculate norm of x:

$$\|x\| = 1 + 0 + 0 + 0 = 1$$

Step 11: Test for reset condition:

$$\frac{\|x\|}{\|s\|} = \frac{1}{1} = 1 \geq 0.4\ (\rho)$$

Hence reset is false. Proceed to Step 12.

Step 12: • Update bottom-up weights for $\alpha = 2$:

$$b_{ij}(\text{new}) = \frac{2x_i}{1 + \|x\|}$$

$$b_{11} = \frac{2 \times 1}{1+1} = \frac{2}{2} = 1;$$

$$b_{21} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{31} = \frac{2 \times 0}{1+1} = 0;$$

$$b_{41} = \frac{2 \times 0}{1+1} = 0$$

Therefore, the bottom-up weight matrix b_{ij} becomes

$$b_{ij} = \begin{bmatrix} 0 & 1 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 1 & 0 & 0.2 \end{bmatrix}$$

• Update the top-down weights:

$$t_{ji}(\text{new}) = x_i$$

$$t_{ji} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Step 13: Test for stopping condition. (This completes one epoch of training.)

The network may be trained for a particular number of epochs on the basis of the stopping condition.

12. Consider an ART 1 neural net with four F_1 units and three F_2 units. After some training, the weights are as follows:

Bottom-up weights

$$b_{ij} = \begin{bmatrix} 0.67 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 0.67 & 0.2 \end{bmatrix}_{4\times3}$$

Top-down weights

$$t_{ji} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}_{3\times4}$$

Determine the new weight matrices after the vector [0, 0, 1, 1] is presented if

- The vigilance parameter is 0.3.
- The vigilance parameter is 0.7.

Solution: It can be noted that $n = 4$, $m = 3$ (clusters) and $\alpha = 2$.

- Vigilance parameter $\rho = 0.3$.
  Bottom-up weight,

$$b_{ij}(0) = \frac{1}{1+n} = \frac{1}{1+4} = 0.2$$

Top-down weight

$$t_{ji}(0) = 1$$

Note: These are not necessary as b_{ij} and t_{ji} weights are already given.

We now compute norm of $s = [0\ 0\ 1\ 1]$:

$$\|s\| = 0 + 0 + 1 + 1 = 2$$

Then we compute the activations of F_1 layer:

$$x = [0\ 0\ 1\ 1]$$

Now, calculate the net input:

$$y_j = \sum_{i=1}^{4} x_i b_{ij}$$

$$y_1 = \sum_{i=1}^{4} x_i b_{i1} = 0.67 \times 0 + 0 \times 0 + 1 \times 0 + 1 \times 0 = 0$$

$$y_2 = \sum_{i=1}^{4} x_i b_{i2} = 0 \times 0 + 0 \times 0 + 1 \times 0 + 1 \times 0.67 = 0.67$$

$$y_3 = \sum_{i=1}^{4} x_i b_{i3} = 0 \times 0.2 + 0 \times 0.2 + 1 \times 0.2 + 1 \times 0.2 = 0.4$$

Since y_2 is the largest, hence the winner unit is $J = 2$.

Compute F_1 activations again,

$$x_i = s_i t_{ji} = [0\ 0\ 1\ 1][0\ 0\ 0\ 1] = [0\ 0\ 0\ 1]$$

Computing the norm of x we obtain

$$\|x\| = 0 + 0 + 0 + 1 = 1$$

We now test for reset. Since

$$\frac{\|x\|}{\|s\|} = \frac{1}{2} = 0.5 \geq 0.3(\rho)$$

we update the weights. Update bottom-up weights (b_{ij}),

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|}\ (\alpha = 2)$$

$$b_{12} = \frac{2 \times 0}{2 - 1 + 1} = 0;$$

$$b_{22} = \frac{2 \times 0}{2 - 1 + 1} = 0;$$

$$b_{32} = \frac{2 \times 0}{2 - 1 + 1} = 0;$$

$$b_{42} = \frac{2 \times 1}{2 - 1 + 1} = 1$$

Update top-down weights, $t_{ji}(\text{new}) = x_i$.
The new top-down weights are

$$t_{ji} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The new bottom-up weights are

$$b_{ij} = \begin{bmatrix} 0.67 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 1 & 0.2 \end{bmatrix}$$

- Vigilance parameter $\rho = 0.7$. The input vector is $s = [0\ 0\ 1\ 1]$. Now calculate norm of s,

$$\|s\| = 0 + 0 + 1 + 1 = 2$$

Set activations of F_1 layer as

$$x = [0\ 0\ 1\ 1]$$

Calculating the net input we obtain

$$y_j = \sum_{i=1}^{4} x_i b_{ij}$$

$$y_1 = \sum_{i=1}^{4} x_i b_{i1} = 0.67 \times 0 + 0 \times 0 + 1 \times 0 + 1 \times 0 = 0$$

$$y_2 = \sum_{i=1}^{4} x_i b_{i2}$$
$$= 0 \times 0 + 0 \times 0 + 1 \times 0 + 1 \times 0.67 = 0.67$$
$$y_3 = \sum_{i=1}^{4} x_i b_{i3}$$
$$= 0 \times 0.2 + 0 \times 0 + 1 \times 0.2 + 1 \times 0.2$$
$$= 0.4$$

As y_2 is the largest, therefore the winner unit index is $J = 2$.
Recomputing the activations of F_1 layer we get $(J = 2)$

$$x_i = s_i t_{ji} = [0\ \ 0\ \ 1\ \ 1][0\ \ 0\ \ 0\ \ 1]$$
$$= [0\ \ 0\ \ 0\ \ 1]$$

The norm of x is

$$\|x\| = 0 + 0 + 0 + 1 = 1$$

We now test for reset condition. Since

$$\frac{\|x\|}{\|s\|} = \frac{1}{2} = 0.5 < 0.7\,(\rho)$$

$y_2 = -1$ (inhibit node 2). Therefore, the net input becomes

$$y_1 = 0; \quad y_2 = -1; \quad y_3 = 0.4$$

As the largest is y_3, the winner unit index is $J = 3$. Recomputing F_1 layer activations, we get

$$x_i = s_i t_{ji} = [0\ \ 0\ \ 1\ \ 1][1\ \ 1\ \ 1\ \ 1]$$
$$= [0\ \ 0\ \ 1\ \ 1]$$

From this we get that the norm of $\|x\| = 2$. Testing for reset we obtain

$$\frac{\|x\|}{\|s\|} = \frac{2}{2} = 1 > 0.7\,(\rho)$$

Hence we update the weights. The bottom-up weights are $(x_i = [0\ 0\ 1\ 1], J = 3)$

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|}$$
$$b_{13} = \frac{2 \times 0}{2 - 1 + 2} = 0;$$
$$b_{23} = \frac{2 \times 0}{2 - 1 + 2} = 0;$$
$$b_{33} = \frac{2 \times 1}{2 - 1 + 2} = 0.67;$$
$$b_{43} = \frac{2 \times 1}{2 - 1 + 2} = 0.67$$

The updated bottom-up weights are

$$b_{ij} = \begin{bmatrix} 0.67 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.67 \\ 0 & 0.67 & 0.67 \end{bmatrix}$$

The top-down weights are given by $t_{ji}(\text{new}) = x_i$. Hence the updated top-down weights are

$$t_{ji} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

13. Consider an ART 1 network with nine input F_1 units and two cluster F_2 units. After some training, the bottom-up weights b_{ij} and top-down weights t_{ji} have the following values:
The bottom-up weights

$$b_{ij} = \begin{bmatrix} 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \end{bmatrix}$$

Top-down weights t_{ji}

$$t_{ji} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The pattern (1 1 1 1 0 1 1 1 1) is presented to the network. Compute the action of the network if

- the vigilance parameter is 0.5;
- the vigilance parameter is 0.8.

Solution: Here $n = 9$, $m = 2$.

- Vigilance parameter $\rho = 0.5$. The initial weight matrices are

$$b_{ij} = \begin{bmatrix} 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \\ 0 & 1/10 \\ 1/3 & 1/10 \end{bmatrix}$$
$$t_{ji} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The input pattern is $s = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$. Calculating the norm of s, we obtain

$$\|s\| = \sum_{i=1}^{9} s_i = 8$$

We now compute the activations of F_1 layer,

$$x = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$$

Calculating the net input, we obtain

$$y_j = \sum_{i=1}^{9} x_i b_{ij}$$
$$y_1 = \sum_{i=1}^{9} x_i b_{i1}$$
$$= 1(1/3) + 1(0) + 1(1/3) + 1(0) + 0(1/3) + 1(0) + 1(1/3) + 1(0) + 1(1/3)$$
$$= \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} = \frac{4}{3} = 1.3$$
$$y_2 = \sum_{i=1}^{9} x_i b_{i2}$$
$$= 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 0(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10)$$
$$= \frac{8}{10} = 0.8$$

It can be seen that $y_1 > y_2$, so the winner unit index is $J = 1$. Recomputing the activations of F_1 layer, we obtain (for $J = 1$)

$$x_i = s_i t_{ji} = [1\ \ 1\ \ 1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1]$$
$$[1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1]$$
$$x = [1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 1]$$

Calculating the norm of x, we obtain

$$\|x\| = 1 + 0 + 1 + 0 + 0 + 0 + 1 + 0 + 1 = 4$$

Testing for reset implies

$$\frac{\|x\|}{\|s\|} = \frac{4}{8} = \frac{1}{2} = 0.5 \geqslant 0.5(\rho)$$

Reset is false. Hence we update the weights. Updating bottom-up weights for $\alpha = 2$ we get

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|} = \frac{2x_i}{2 - 1 + \|x\|}$$
$$= \frac{2x_i}{1 + \|x\|}$$
$$b_{11} = \frac{2(1)}{1+4} = \frac{2}{5}; \quad b_{21} = \frac{2(0)}{1+4} = 0;$$
$$b_{31} = \frac{2(1)}{1+4} = \frac{2}{5}; \quad b_{41} = \frac{2(0)}{1+4} = 0;$$
$$b_{51} = \frac{2(0)}{1+4} = 0; \quad b_{61} = \frac{2(0)}{1+4} = 0;$$
$$b_{71} = \frac{2(1)}{1+4} = \frac{2}{5}; \quad b_{81} = \frac{2(0)}{1+4} = 0;$$
$$b_{91} = \frac{2(1)}{1+4} = \frac{2}{5}$$

The updated bottom-up weights are

$$b_{ij} = \begin{bmatrix} 2/5 & 1/10 \\ 0 & 1/10 \\ 2/5 & 1/10 \\ 0 & 1/10 \\ 0 & 1/10 \\ 0 & 1/10 \\ 2/5 & 1/10 \\ 0 & 1/10 \\ 2/5 & 1/10 \end{bmatrix}$$

We now update top-down weights using t_{ji} (new) $= x_i$. The new updated top-down weight are

$$t_{ji} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

- Vigilance parameter $\rho = 0.8$. The input pattern is $s = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$. Calculating the norm of s using the formula as in (1), we obtain

$$\|s\| = 8$$

We now compute the activations of F_1 layer,

$$x = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$$

Calculating the net input, we get

$$y_j = \sum_{i=1}^{9} x_i b_{ij}$$

$$y_1 = \sum_{i=1}^{9} x_i b_{i1}$$

$$= 1(1/3) + 1(0) + 1(1/3) + 1(0) + 0(1/3) + 1(0) + 1(1/3) + 1(0) + 1(1/3)$$

$$= \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{1}{3} + \frac{4}{3} = 1.3$$

$$y_2 = \sum_{i=1}^{9} x_i b_{i2}$$

$$= 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10) + 1(1/10)$$

$$= \frac{8}{10} = 0.8$$

It can be seen that $y_1 > y_2$. Hence the winner unit index is $J = 1$.
Recomputing the activations of F_1 layer, we obtain (for $J = 1$)

$$x_i = s_i t_{ji} = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1] \\ [1\ 0\ 1\ 0\ 1\ 0\ 1\ 0\ 1]$$

$$x = [1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 1]$$

Computing the norm of x, we have

$$\|x\| = 1 + 0 + 1 + 0 + 0 + 0 + 1 + 0 + 1 = 4$$

Testing for reset we obtain

$$\frac{\|x\|}{\|s\|} = \frac{4}{8} = \frac{1}{2} = 0.5 < 0.8$$

Hence $y_1 = -1$ (inhibit node 1). Therefore, the dot products become

$$y_1 = -1; \quad y_2 = 0.8$$

Since $y_2 > y_1$, the winner unit index is $J = 2$. Recomputing activations of F_1 layer (for $J = 2$) gives

$$x_i = s_i t_{ji} = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1] \\ \times [1\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ 1]$$

$$x = [1\ 1\ 1\ 1\ 0\ 1\ 1\ 1\ 1]$$

Again computing the norm of x, we get

$$\|x\| = 1 + 1 + 1 + 1 + 0 + 1 + 1 + 1 + 1 = 8$$

Testing for reset gives

$$\frac{\|x\|}{\|s\|} = \frac{8}{8} = 1 > 0.8$$

Hence we update the weights.
Update bottom-up weights, for $\alpha = 2$, using the formula

$$b_{ij}(\text{new}) = \frac{\alpha x_i}{\alpha - 1 + \|x\|} = \frac{2x_i}{2 - 1 + \|x\|} = \frac{2x_i}{1 + \|x\|}$$

For all $x_i = 1$ (where $i = 1$ to 4 and 6 to 9, and $J = 2$),

$$b_{ij}(\text{new}) = \frac{2 \times 1}{1 + 8} = \frac{2}{9}$$

For $x_i = 0$ (where $i = 5$ and $J = 2$)

$$b_{ij}(\text{new}) = \frac{2 \times 0}{1 + 8} = 0$$

The updated bottom-up weights are

$$b_{ij} = \begin{bmatrix} 1/3 & 2/9 \\ 0 & 2/9 \\ 1/3 & 2/9 \\ 0 & 2/9 \\ 1/3 & 0 \\ 0 & 2/9 \\ 1/3 & 2/9 \\ 0 & 2/9 \\ 1/3 & 2/9 \end{bmatrix}$$

Updated top-down weights can be calculated using the formula $t_{ji}(\text{new}) = x_i$.
The new updated top-down weights are

$$t_{ji} = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

14. Consider an ART 2 network with two input units. ($n = 2$). Show that using $\theta = 0.7$ will force the input patterns (0.71, 0.69) and (0.69, 0.71) to different clusters. What role does the vigilance parameters play in this case? (Do not calculate the weights, stop with checking of reset condition.) Assume the necessary parameters.

Solution: The parameters are assumed to be

$a = b = 10$, $c = 0.1$, $d = 0.9$, $e = 0$, $\alpha = 0.6$, $\rho = 0.9$, $\theta = 0.7$, $n = 2$, $t_j = (0, 0)$,

$$b_j = \frac{1}{(1 - n) e} = (7.0, 7.0)$$

1st *pattern*:

$$s = (0.71, 0.69)$$

$$u = \frac{v}{e + \|v\|} = (0, 0)$$

$$w = s + au = (0.71, 0.69)$$

$$p = u + d t_j = (0, 0)$$

$$x = \frac{w}{e + \|w\|} = \frac{(0.71, 0.69)}{0.99} = (0.717, 0.697)$$

(where $\|w\| = \sqrt{(0.71)^2 + (0.69)^2} = 0.99$)

$$q = \frac{p}{e + \|p\|} = (0, 0)$$

$$v = f(x_i) + b f(q_i)$$

$$= f(0.717, 0.697)$$

$$v = (0.72, 0)$$

Here the activation function is

$$f(x) = \begin{cases} x & f(x) \geq \theta \\ 0 & f(x) < \theta \end{cases}$$

Since $\theta = 0.7$, the output $v = (0.72, 0)$. Update F_1 activations again:

$$u = \frac{v}{e + \|v\|} = \frac{(0.72, 0)}{0.72} = (1, 0)$$

$$w = s + au = (0.71, 0.69) + 10(1, 0) = (10.71, 0.69)$$

$$p = u + d t_j = (1, 0)$$

$$x = \frac{w}{e + \|w\|} = \frac{(10.71, 0.69)}{10.732} = (0.998, 0.064)$$

$$q = \frac{p}{e + \|p\|} = \frac{(1, 0)}{1} = (1, 0)$$

$$v_i = f(x_i) + b f(q_i)$$

$$= f(0.998, 0.064) + 10 f(1, 0)$$

$$= (0.998, 0) + 10(1, 0)$$

$$v_i = (10.998, 0)$$

Calculating signals to F_2 units, we get

$$y_j = \Sigma\, b_{ij}\, p_i = (7, 7) \times (1, 0) = (7, 0)$$

The winner unit is $J = 1$, since y_1 has a larger value, i.e. 7, than $y_2 = 0$.

Check for reset:

$$u = \frac{v}{e + \|v\|} = \frac{(10.998, 0)}{10.998} = (1, 0)$$

$$p = u + d t_j = (1, 0) + 0.9(0, 0) = (1, 0)$$

$$r = \frac{u + cp}{e + \|u\| + c\|p\|} = \frac{(1, 0) + 0.1(1, 0)}{0 + 1 + 0.1 \times 1}$$

$$= \frac{(1.1, 0)}{1.1} = (1, 0)$$

Computing norm of r, we get

$$\|r\| = 1 > (\rho - e) = 1 > 0.9$$

This implies

$$w = s + au = (0.71, 0.69) + 10(1, 0)$$
$$= (10.71, 0.69)$$
$$x = \frac{w}{e + \|w\|} = (0.998, 0.064)$$
$$q = \frac{\rho}{e + \|\rho\|} = (1, 0)$$
$$v = f(x) + bf(q) = (10.998, 0)$$

2nd pattern

$$s = (0.69, 0.71)$$
$$u = \frac{v}{e + \|v\|} = 0$$
$$w = s + au = (0.69, 0.71)$$
$$\rho = u + dt_J = (0, 0)$$
$$x = \frac{w}{e + \|w\|} = \frac{(0.69, 0.71)}{0.99} = (0.697, 0.717)$$
$$q = \frac{\rho}{e + \|\rho\|} = (0, 0)$$
$$v = f(x) + bf(q) = f(0.697, 0.717)$$
$$= (0, 0.717) = (0, 0.71)$$

Update F_1 activations again:

$$u = \frac{v}{e + \|v\|} = \frac{(0, 0.72)}{0.72} = (0.1)$$
$$w = s + au = (0.69, 0.71) + 10(0, 1)$$
$$= (0.69, 10.71)$$
$$\rho = u + dt_J = (0, 1)$$
$$x = \frac{w}{e + \|w\|} = \frac{(0.69, 10.71)}{10.732} = (0.064, 0.998)$$
$$q = \frac{\rho}{e + \|\rho\|} = \frac{(0, 1)}{1} = (0, 1)$$
$$v = f(x) + bf(q) = f(0.064, 0.998) + 10f(0, 1)$$
$$= (0, 0.998) + 10(0, 1)$$
$$v = (0, 10.998)$$

Calculating signals to F_2 units, we get

$$y_j = \Sigma\, b_{ij}\, \rho_i = (7, 7) \times (0, 1) = (0, 7)$$

The winner unit is $J = 2$, since $y_2 > y_1$ [i.e., $7 > 0$]

Check for reset

$$u = \frac{v}{e + \|v\|} = \frac{(0, 10.998)}{10.998} = (0, 1)$$
$$\rho = u + dt_J = (0, 1)$$
$$r = \frac{u + c\rho}{e + \|u\| + c\|\rho\|} = \frac{(0, 1) + 0.1(0, 1)}{0 + 1 + 0.1 \times 1}$$
$$= \frac{(0, 1.1)}{1.1} = (0, 1)$$
$$\|r\| = 1 > (\rho - e) = 0.9.$$

Then,

$$w = s + au = (0.69, 0.71) + 10(0, 1)$$
$$= (0.67, 10.71)$$
$$x = \frac{w}{e + \|w\|} = \frac{(0.69, 10.71)}{10.732} = (0.064, 0.998)$$
$$q = \frac{\rho}{e + \|\rho\|} = (0, 1)$$
$$v = f(x) + bf(q) = (0, 10.998)$$

Thus the vigilance parameter assumed $\rho = 0.9$ does not affect the solutions for first and second pattern. It activates the same for both the input patterns.

15. Consider an ART 2 network to cluster the input vectors (0.6, 0.8, 0.0) and (0.8, 0.6, 0.0) together? When will it place (0.6, 0.2, 0.0) together with (0.0, 1.0, 0.0)? Use the noise suppression parameter value $\theta = 1\sqrt{3} = 0.577$ and consider different values of the vigilance and different initial weights. Assume necessary parameters.

Solution: Case (i) Taking $\rho = 0.9$, presenting (0.6, 0.8, 0) and (0.8, 0.6, 0)

$$a = 10, b = 10, c = 0.1, d = 0.9, e = 0, \rho = 0.9,$$
$$\theta = 0.577, b_j = \frac{1}{(1 - d)\sqrt{n}} = \frac{1}{(1 - 0.9)\sqrt{3}}$$
$$= (5.0, 5.0, 5.0),\ t_j = (0, 0, 0),\ \alpha = 0.6$$

1st pattern:

$$s = (0.6, 0.8, 0.0)$$

$$u = \frac{v}{e + \|v\|} = (0, 0, 0)$$
$$w = s + an = (0.6, 0.8, 0.0)$$
$$\rho = u + dt_J = (0, 0, 0)$$
$$x = \frac{w}{e + \|w\|} = \frac{(0.6, 0.8, 0.0)}{0 + 1} = (0.6, 0.8, 0.0)$$
$$q = \frac{\rho}{e + \|\rho\|} = (0, 0, 0)$$
$$v_i = f(x_i) + bf(q_i)$$
$$= f(0.6, 0.8, 0) + 10f(0, 0, 0)$$
$$\begin{bmatrix} \theta = 0.55 \\ f(x) = \begin{cases} x, & f(x) \geq 0 \\ 0, & f(x) < \theta \end{cases} \end{bmatrix}$$
$$v_i = (0.6, 0.8, 0)$$

Update F_1 activations again:

$$u = \frac{v_i}{e + \|v\|} = \frac{(0.6, 0.8, 0)}{0 + 1} = (0.6, 0.8, 0)$$
$$w = s + au = (0.6, 0.8, 0) + 10(0.6, 0.8, 0)$$
$$= (6.6, 8.8, 0)$$
$$\rho = u + dt_J = (0.6, 0.8, 0)$$
$$x = \frac{w}{e + \|w\|} = \frac{(0.6, 8.8, 0)}{0 + 11} = (0.6, 0.8, 0)$$
$$q = \frac{\rho}{e + \|\rho\|} = \frac{(0.6, 0.8, 0)}{0 + 1} = (0.6, 0.8, 0)$$
$$v_i = f(x_i) + bf(q_i)$$
$$= f(0.6, 0.8, 0) + 10f(0.6, 0.8, 0)$$
$$= (0.6, 0.8, 0) + (6, 8, 0)$$
$$v_i = (6.6, 8.8, 0)$$

Calculate the signals to F_2 units:

$$y_j = \Sigma\, b_{ij}\, \rho_i = (5.0 \times 0.6, 5.0 \times 0.8, 5.0 \times 0)$$
$$= (3.0, 4.0, 0)$$

The winner unit index is $J = 2$, since the largest net input is 4.0.

Check for reset:

$$u = \frac{v}{e + \|v\|} = \frac{(6.6, 8.8, 0)}{0 + 11} = (0.6, 0.8, 0)$$
$$\rho_i = u_i + dt_J$$
$$= (0.6, 0.8, 0) + 0.9(0, 0, 0) = (0.6, 0.8, 0)$$
$$r = \frac{u_i + c\rho_i}{e + \|u\| + c\|\rho\|}$$
$$= \frac{(0.6, 0.8, 0) + 0.1(0.6, 0.8, 0)}{0 + 1 + 0.1 \times 1}$$
$$= \frac{(0.66, 0.88, 0)}{1.1} = (0.6, 0.8, 0)$$

Computing norm of r, we get

$$\|r\| = 1 > (\rho - e) = 0.9$$
$$q = \frac{\rho}{e + \|\rho\|} = \frac{(0.6, 0.8, 0)}{0 + 1} = (0.6, 0.8, 0)$$
$$w = \frac{w}{e + \|w\|}$$
$$= (0.6, 0.8, 0) + 10(0.6, 0.8, 0) = (6.6, 8.8, 0)$$
$$x = \frac{w}{e + \|w\|} = \frac{(6.6, 8.8, 0)}{0 + 11} = (0.6, 0.8, 0)$$
$$v = f(x) + bf(q) = (6.6, 8.8, 0)$$

Updation of weights: Weights are updated for winning unit $J = 2$.

$$t_{ji}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\} t_{ji}(\text{old})$$
$$= 0.6 \times 0.9 \times 0.8$$
$$+ \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 0$$
$$= 0.432$$
$$b_{ij}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\} b_{ij}(\text{old})$$
$$= 0.6 \times 0.9 \times 0.8$$
$$+ \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 5.0$$
$$= 5.162$$

2nd pattern:

$$s = (0.8, 0.6, 0)$$
$$u = \frac{v}{e + \|v\|} = (0, 0)$$
$$w = s + au = (0.8, 0.6, 0)$$
$$\rho = u + d_{ij} = (0, 0)$$

$$x = \frac{w}{e + \|w\|} = (0.8, 0.6, 0.0)$$

$$q = \frac{p}{e + \|p\|} = (0, 0, 0)$$

$$v = f(x_i) + bf(q_i)$$

$$= f(0.8, 0.6, 0.0) + 0 = (0.8, 0.6, 0)$$

Update F_1 activations again:

$$u = \frac{v}{e + \|v\|} = \frac{(0.8, 0.6, 0)}{0 + 1} = (0.8, 0.6, 0)$$

$$w = s + qu = (0.8, 0.6, 0) + 10\,(0.8, 0.6, 0)$$

$$= (8.8, 6.6, 0)$$

$$p = u + dt_J = (0.8, 0.6, 0)$$

$$x = \frac{w}{e + \|w\|} = (0.8, 0.6, 0)$$

$$q = \frac{p}{e + \|p\|} = (0.8, 0.6, 0)$$

$$v = f(x) + bf(q)$$

$$= f(0.8, 0.6, 0) + 10 f(0.8, 0.6, 0)$$

$$= (0.8, 0.6, 0) + (8, 6, 0)$$

$$v = (8.8, 6.6, 0)$$

Calculate signals to F_2 units:

$$y_j = \Sigma\, b_{ij} p_i = (5, 5, 0) \times (0.8, 0.6, 0) = (4, 3, 0)$$

The winner unit index is $J = 1$, since the largest net in put is 4.0.

Check for reset:

$$u = \frac{v}{e + \|v\|} = \frac{(8.8, 6.6, 0)}{0 + 11} = (0.8, 0.6, 0)$$

$$p_i = u + dt_J = (0.8, 0.6, 0)$$

$$r_i = \frac{u_i + cp_i}{e + \|u\| + c\|p\|} = (0.8, 0.6, 0)$$

Computing norm of r, we get

$$\|r\| = 1 > (p - e) = 0.9$$

$$q = \frac{p}{e + \|p\|} = \frac{(0.8, 0.6, 0)}{0 + 1} = (0.8, 0.6, 0)$$

$$w = s + au = (8.8, 6.6, 0)$$

$$x = \frac{w}{e + \|w\|} = (0.8, 0.6, 0)$$

$$v = f(x) + bf(q) = (8.8, 6.6, 0)$$

Updation of weights: Weights are updated for winning unit $J = 1$.

$$t_{ji}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\} t_{ji}(\text{old})$$

$$= 0.6 \times 0.9 \times 0.8 + \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 0$$

$$= 0.432$$

$$b_{ij}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\}\, b_{ij}(\text{old})$$

$$= 0.6 \times 0.9 \times 0.8 + \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 5.0$$

$$= 5.162$$

Thus from the results for presenting the patterns (0.6, 0.8, 0) and (0.6, 0.8, 0) with the assumption of $\rho = 0.9$, even though the winning cluster unit is different, due to the components of input vector, output weights remains same. Thus they are placed together–since both the weights are same, both the patterns will be placed at the same location.

Case (ii): Taking $\rho = 0.7$, presenting (0.6, 0.8, 0) and (0, 1, 0).

$a = 10, b = 10, c = 0.1, d = 0.9, e = 0, \rho = 0.7,$

$\theta = 0.577, \alpha = 0.6, b_j = (6.0, 6.0, 6.0),$

$t_j = (0, 0, 0)$

1*st pattern*

$$s = (0.6, 0.8, 0.0)$$

$$u = \frac{v}{e + \|v\|} = (0, 0, 0)$$

$$w = s + au = (0.6, 0.8, 0)$$

$$p = u + dt_J = (0, 0, 0)$$

$$x = \frac{w}{e + \|0\|} = (0.6, 0.8, 0.0)$$

$$q = \frac{p}{e + \|p\|} = (0, 0, 0)$$

$$v_i = f(x_i) + bf(q_i) = (0.6, 0.8, 0)$$

Update F_1 activations again:

$$u = \frac{v}{e + \|v\|} = (0.6, 0.8, 0)$$

$$w = s + au = (6.6, 8.8, 0)$$

$$p = u + dt_J = (0.6, 0.8, 0)$$

$$x = \frac{w}{e + \|w\|} = (0.6, 0.8, 0)$$

$$q = \frac{p}{e + \|p\|} = (0.6, 0.8, 0)$$

$$v = f(x_i) + bf(q_i) = (6.6, 8.8, 0)$$

Calculate the signals to F_2 units:

$$y_j = \Sigma\, b_{ij} p_i = (6 \times 0.6, 6 \times 0.8, 6 \times 0)$$

$$= (3.6, 4.8, 0)$$

The winner unit index is $J = 2$, since the largest net input is 4.8.

Check for reset:

$$u = \frac{v}{e + \|v\|} = \frac{(6.6, 8.8, 0)}{0 + 11} = (0.6, 0.8, 0)$$

$$p_i = u + dt_J = (0.6, 0.8, 0)$$

$$r = \frac{u + cp}{e + \|w\| + c\|p\|} = (0.6, 0.8, 0)$$

$$\|r\| = 1 > (p - e) = 0.7$$

$$q = \frac{p}{e + \|p\|} = \frac{(0.0, 0.8, 0)}{0 + 1} = (0.6, 0.8, 0)$$

$$w = s + au = (6.6, 8.8, 0)$$

$$x = \frac{w}{e + \|u\|} = (0.6, 0.8, 0.0)$$

$$v = f(x) + bf(q) = (6.6, 8.8, 0)$$

Updation of weights: Weights are updated for winning unit $J = 2$.

$$t_{ji}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\} t_{ji}(\text{old})$$

$$= 0.6 \times 0.9 \times 0.8 + 0$$

$$= 0.432$$

$$b_{ij}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\}\, b_{ij}(\text{old})$$

$$= 0.6 \times 0.9 \times 0.8 + \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 6.0$$

$$= 6.108$$

2*nd pattern:*

$$s = (0, 1, 0)$$

$$u = \frac{v}{e + \|v\|} = (0, 0, 0)$$

$$w = s + au = (0, 1, 0)$$

$$p = u + dt_J = (0, 1, 0)$$

$$x = \frac{w}{e + \|w\|} = (0.0, 1.0, 0.0)$$

$$q = \frac{p}{e + \|p\|} = (0.0, 1.0, 0.0)$$

$$v = f(x_i) + bf(q_i) = (0.0, 1.0, 0.0)$$

Update F_1 activations again:

$$u = \frac{v}{e + \|v\|} = (0.0, 1.0, 0.0)$$

$$w = s + au = (0, 1, 0) + 10(0, 1, 0) = (0, 11, 0)$$

$$p = u + dt_J = (0, 1, 0)$$

$$x = \frac{w}{e + \|w\|} = (0, 1, 0)$$

$$q = \frac{p}{e + \|p\|} = (0, 1, 0)$$

$$v = f(x) + bf(q) = (0, 1, 0) + (0, 10, 0)$$

$$= (0, 11, 0)$$

Calculating signals to F_2 units, we get

$$y_j = \Sigma\, b_{ij} p_i = (6.0, 6.0, 6.0)(0.0, 1.0, 0.0)$$

$$= (0.0, 6.0, 0.0)$$

The winner unit index is $J = 2$, since the largest net input is 6.0.

Check for reset:

$$u = \frac{v}{e + \|v\|} = \frac{(0, 11, 0)}{0 + 11} = (0, 1, 0)$$

$$p = u + d\,tj = (0, 1, 0)$$

$$r = \frac{u + c\,p}{e + \|u\| + c\|p\|} = \frac{(0, 1, 0) + 0.1\,(0, 1, 0)}{0 + 1 + 0.1}$$

$$r = \frac{(0.0, 1.1, 0.0)}{0 + 1.1} = (0.0, 1.0, 0.0)$$

$$\|r\| = 1 > (p - e) = 0.7$$

$$q = \frac{p}{e + \|p\|} = (0.0, 1.0, 0.0)$$

$$w = s + au = (0.0, 1.0, 0.0) + 10\,(0.0, 1.0, 0.0)$$

$$= (0, 11, 0)$$

$$x = \frac{w}{e + \|w\|} = (0, 1, 0)$$

$$v = f(x) + bf(q) = f(0, 1, 0) + 10f(0, 1, 0)$$

$$= (0, 11, 0)$$

Updation of weights: Weights are updated for winning unit $J = 2$.

$$t_{ji}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\} t_{ji}(\text{old})$$

$$= 0.6 \times 0.9 \times 1 + 0 = 0.54$$

$$b_{ij}(\text{new}) = \alpha d u_i + \{1 + \alpha d(d - 1)\}\, b_{ij}(\text{old})$$

$$= 0.6 \times 0.9 \times 1$$

$$+ \{1 + 0.6 \times 0.9(0.9 - 1)\} \times 6 = 6.216$$

Thus the two inputs may be clustered together only when their weights become equal, this can be achieved by proper selection of initial weights.

5.9 Review Questions

1. What is meant by unsupervised learning?
2. Define exemplar vector or code book vector.
3. List the fixed weight competitive nets.
4. Draw the architecture of Mexican hat and state its activation function.
5. What is winner-takes-all or clustering principle or competitive learning?
6. Why inhibitory weights are used in Max net?
7. What are "topology preserving" maps?
8. Define Euclidean distance.
9. Briefly discuss about Hamming net.
10. How is competition performed for clustering of the vectors?
11. State the application of Kohonen self-organizing maps.
12. With neat architecture, explain the training algorithm of Kohonen self-organizing feature maps.
13. Discuss the important features of Kohonen self-organizing maps.
14. Write the principle involved in learning vector quantization.
15. What is the purpose of LVQ net?
16. How are the initial weights determined for LVQ net?
17. With architecture, describe how LVQ nets are trained.
18. List the variants of LVQ net.
19. State Kohonen's learning rule and Grossberg learning rule.
20. Discuss the applications of counterpropagation network.
21. How many stages are needed for training a CPN network?
22. Mention the importance of in-star model and out-star model.
23. Sketch the architecture of full Counter Propagation Network.
24. How are CPN nets used for function approximation?

25. Write the training algorithms and testing algorithms used in full counterpropagation network.
26. Compare full counterpropagation network and forward-only counterpropagation network.
27. What is the principle strength of competitive learning?
28. State the merits and demerits of Kohonen self-organizing feature maps.
29. What are called as similarity maps?
30. Define stability and plasticity.
31. Differentiate between ART networks and CPN networks.
32. List the type of input patterns given to ART 1 and ART 2 network.
33. Mention the three main components of an ART network.
34. Define bottom-up weight and top-down weight.
35. What is vigilance parameter and noise suppression parameter?
36. Illustrate with neat figure, the two basic units of an ART 1 network.
37. Discuss the importance of supplemental units in ART 1 network.
38. Differentiate fast learning and slow learning.
39. List the advantages and disadvantages of ART network.
40. What are the applications of ART networks?
41. Sketch the architecture of ART 1 network and discuss its training algorithm.
42. State the significance of ART 2 network.
43. Why more complexity is involved in the F_1 layer of ART 2 network?
44. How slow learning and fast learning is achieved in ART 2 network?
45. What is the activation function used in ART 2 networks?
46. List the characteristics of ART network.
47. Why reset mechanism is essential in ART networks?
48. With neat architecture, explain the training algorithm used in ART network.
49. State the assumptions made in ART 2 network.
50. Mention the limitation of ART 1 network and how is it overcome in ART 2 network.

5.10 Exercise Problems

1. Construct a Max net with four neurons and inhibitory weights $E = 0.25$ when given the initial activations (input signals). The initial activations are $a_1(0) = 0.1$, $a_2(0) = 0.3$, $a_3(0) = 0.4$, $a_4(0) = 0.7$.
2. Construct a Kohonen self-organizing feature map to cluster four vectors [0 0 1 1], [1 0 0 1], [0 1 0 1], [1 1 1 1]. The maximum number of clusters to be formed is 2 and assume learning rate as 0.5. Assume random initial weights.
3. Given a Kohonen self-organizing map with weights as shown in the following figure, use square of euclidean distance to find the cluster unit that is close to the input vector (0.35, 0.05). Using learning rate of 0.25, find the new weights.

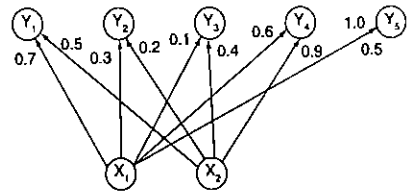


Figure 11 KSOFM net.

4. Repeat the preceding exercise problem for input vector [0.4, 0.4] with $\alpha = 0.15$.

5. Consider a Kohonen net with two cluster units and five input units. The weights vectors for the cluster units are

$$w_1 = (1.0\ 0.9\ 0.7\ 0.3\ 0.2)$$
$$w_2 = (0.6\ 0.7\ 0.5\ 0.4\ 1.0)$$

Use the square of the Euclidean distance to find the winning cluster unit for the input pattern $x = (0.0\ 0.2\ 0.1\ 0.2\ 0.0)$. Using a learning rate of 0.2, find the new weights for the winning unit.

6. Construct an LVQ net to cluster five vectors assigned to two classes. The following input vectors represent two classes 1 and 2.

Vectors	Class
(1 0 0 1)	1
(1 1 0 0)	2
(0 1 1 0)	1
(1 0 0 0)	2
(0 0 1 1)	1

Perform only one epoch of training.

7. Consider an LVQ net with two input units and four target classes: c_1, c_2, c_3 and c_4. There are 16 classification units, with weight vectors indicated by the coordinates on the following chart, read in row-column order.

x_2							
1.0							
0.9		c_4	c_3	c_2	c_1		
0.7		c_2	c_1	c_4	c_3		
0.5		c_4	c_3	c_2	c_1		
0.3		c_2	c_1	c_4	c_3		
0.0							
	0.0	0.3	0.5	0.7	0.9	1.0	x_1

Using the square of the euclidean distance, perform the following.

- Present an input vector of (0.35, 0.35) representing class 3. Using a learning rate of $\alpha = 0.4$, show which classification unit moves where.
- Present an input vector of (0.6, 0.75) representing class 1. What happens to the network performance?
- Present the vector (0.4, 0.55) representing class 1. Note what happens.

8. Implement a counterpropagation network for approximating the functions:
   - $f(x) = 1/x$
   - $f(x) = 7x^2$
9. Consider the following full CPN net:

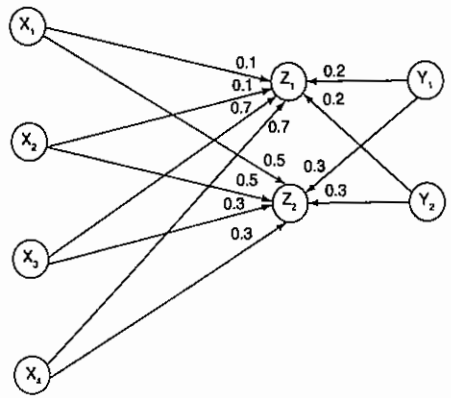


Figure 12 Full CPN Net.

Using the input pair $x = (1, 0, 0, 0), y = (0, 1)$, perform the first phase of training (one step only). Find the activation of the cluster layer units. Update the weights using a learning rate of 0.25.

10. Repeat Problem 9 using $x = (0\ 1\ 1\ 1)$ and $y = (1, 0)$ with a learning rate of 0.3.
11. Modify Problem 9 to implement forward-only CPN.
12. Construct an ART 1 network to cluster four vectors (1, 0, 1, 1), (1, 1, 1, 0), (1, 0, 0, 0) and (0, 1, 0, 1) in at most three clusters using very low vigilance parameter. Assume necessary parameters.
13. Consider an ART 1 neural net with four F_1 units and three F_2 units. After some training, the weights are as follows:

Bottom-up weights b_{ij}

$$\begin{bmatrix} 0.57 & 0 & 0.2 \\ 0 & 0 & 0.2 \\ 0 & 0.37 & 0.2 \\ 0 & 0.37 & 0.2 \end{bmatrix}$$

Top-down weights t_{ji}

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Determine the new weight matrices after the vector (0, 0, 1, 1) is presented if

- the vigilance parameter is 0.4;
- the vigilance parameter is 0.8.

14. Consider an ART 1 network with eight input (F_1) units and two cluster (F_2) units. After some training, the bottom-up weight (b_{ij}) and top-down weights (t_{ji}) are the following values:

Bottom-up weights b_{ij}

$$\begin{bmatrix} 1/2 & 1/8 \\ 0 & 1/8 \\ 1/2 & 1/8 \\ 0 & 1/8 \\ 1/2 & 1/8 \\ 0 & 1/8 \\ 1/2 & 1/8 \\ 0 & 1/8 \end{bmatrix}$$

Top down weights t_{ji}

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

The pattern [1 1 1 0 0 1 1 1] is presented to the network. Compute the action of the network if

- the vigilance parameter is 0.3;
- the vigilance parameter is 0.7.

15. Consider an ART 2 network with two input units ($n = 2$). Show that using $\theta = 0.7$ will force the input patterns (0.61, 0.59) and (0.59, 0.61) to different clusters. What role does vigilance parameter play here? Determine the new weights.

5.11 Projects

1. Write a computer program to implement Kohonen self-organizing map. Take suitable application. Use 2 input units and 25 cluster units and a linear topology for the cluster units. Perform 20 epochs of training.
2. Write a computer program to implement the LVQ net absorbed in Problem 7. Train the net with several sets of data. Experiment with different learning rates and different numbers of classification units.
3. Write a program for counterpropagation network to approximate the function $f(x) = 1/x$.
4. Implement counterpropagation network for performing data compression. Take data sets like heart disease data, cancer data and credit card data.
5. Write a program to approximate the function $f(x) = 7/x$ using forward-only counterpropagation net.
6. Let the digits 0, 1, 2, . . . , 7 be represented as

0 : 0 0 0 0 0 0 0 1
1 : 0 0 0 0 0 0 1 0
2 : 0 0 0 0 0 1 0 0
3 : 0 0 0 0 1 0 0 0
4 : 0 0 0 1 0 0 0 0
5 : 0 0 1 0 0 0 0 0
6 : 0 1 0 0 0 0 0 0
7 : 1 0 0 0 0 0 0 0

Use forward-only and full counterpropagation nets to map digits to their binary representations

0 : 0 0 0
1 : 0 0 1
2 : 0 1 0
3 : 0 1 1
4 : 1 0 0
5 : 1 0 1
6 : 1 1 0
7 : 1 1 1

Assume the necessary parameters involved.

7. Writer a computer program to implement full counterpropagation network for approximating the function $f(x) = 3x + 2$.
8. Build a computer program to implement ART 1 neural net.
9. Write a computer program to implement the ART 2 neural network, allowing for either fast learning or slow learning, based on the number of epochs of training and the number of weight update iterations performed on each learning trial.
10. Write a compute program to implement ART 2 network for Problem 15.

Special Networks

6

Learning Objectives

- The other special networks apart from supervised learning, unsupervised learning and association networks.
- A simulated annealing network.
- How Boltzmann machine can be used to solve optimization problems.
- An introduction to Cauchy and Gaussian machine.
- A probabilistic neural network.
- The feature of cascade correlation network to fluid its own architecture during training progresses.
- A cognitron and neocognitron nets with their basic features.
- Apart from these, cellular NN, logicon NN, STCNN are also discussed.
- List of neuroprocessor chips that are currently in use.

6.1 Introduction

In this chapter, we will discuss some specialized networks in more detail. Among the networks to be discussed are Boltzmann network, cascade correlation net, probabilistic neural net, Cauchy and Gaussian net, cognitron and neocognitron nets, spatio-temporal network, optical neural net, simulated annealing network, cellular neural net and logicon neural net. Besides, neuroprocessor chip has also been discussed for the benefit of the reader. Boltzmann network is designed for optimization problems, such as traveling salesman problem. In this network, fixed weights are used based on the constraints and quantity to be optimized. Probabilistic neural net is designed using the probability theory to classify the input data (Bayesian method). Cascade correlation net is designed depending on the hierarchical arrangement of the hidden units. Cauchy and Gaussian net is the variation of fixed weight optimization net. Cognitron and neocognitron nets were designed for recognition of handwritten digits.

6.2 Simulated Annealing Network

The concept of simulated annealing has it origin in the physical annealing process performed over metals and other substances. In metallurgical annealing, a metal body is heated almost to its melting point and then cooled back slowly to room temperature. This process eventually makes the metal's global energy function reach an absolute minimum value. If the metal's temperature is reduced quickly, the energy of the metallic lattice will be higher than this minimum value because of the existence of frozen lattice dislocations that would otherwise disappear due to thermal agitation. Analogous to the physical annealing behavior, simulated annealing can make a system change its state to a higher energy state having a chance to jump from local

minima to global minima. There exists a *cooling procedure* in the simulated annealing process such that the system has a higher probability of changing to an increasing energy state in the beginning phase of convergence. Then, as time goes by, the system becomes stable and always moves in the direction of decreasing energy state as in the case of normal minimization produce.

With simulated annealing, a system changes its state from the original state SA^{old} to a new state SA^{new} with a probability P given by

$$P = \frac{1}{1 + \exp(-\Delta E/T)}$$

where $\Delta E = E^{old} - E^{new}$ (energy change = difference in new energy and old energy) and T is the non-negative parameter (acts like temperature of a physical system). The probability P as a function of change in energy (ΔE) obtained for different values of the temperature T is shown in Figure 6-1. From Figure 6-1, it can be noticed that the probability when $\Delta E > 0$ is always higher than the probability when $\Delta E < 0$ for any temperature.

An optimization problem seeks to find some configuration of parameters $\dot{X} = (X_1, \ldots, X_n)$ that minimizes some function $f(\dot{X})$ called cost function. In an artificial neural network, configuration parameters are associated with the set of weights and the cost function is associated with the error function.

The simulated annealing concept is used in statistical mechanics and is called Metropolis algorithm. As discussed earlier, this algorithm is based on a material that anneals into a solid as temperature is slowly decreased. To understand this, consider the slope of a hill having local valleys. A stone is moving down the hill. Here, the local valleys are local minima, and the bottom of the hill is going to be the global or universal minimum. It is possible that the stone may stop at a local minimum and never reaches the global minimum. In neural nets, this would correspond to a set of weights that correspond to that of local minimum, but this is not the desired solution. Hence, to overcome this kind of situation, simulated annealing perturbs the stone such that if it is trapped in a local minimum, it escapes from it and continues falling till it reaches its global minimum (optimal solution). At that point, further perturbations cannot move the stone to a lower position. Figure 6-2 shows the simulated annealing between a stone and a hill.

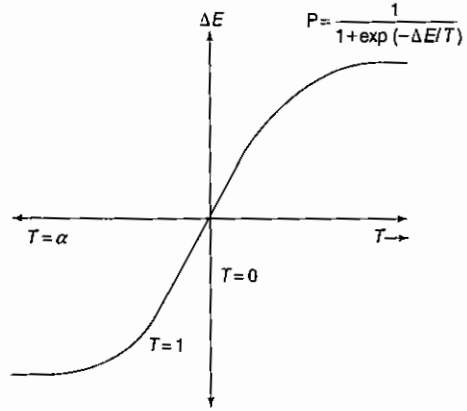


Figure 6-1 Probability "P" as a function of change in energy (ΔE) for different values of temperature T.

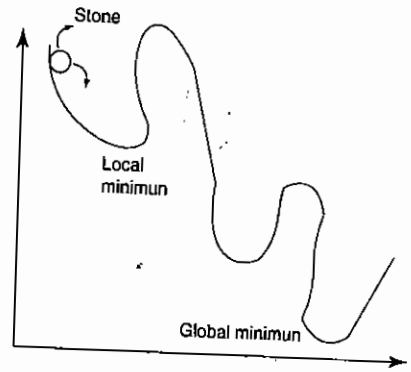


Figure 6-2 Simulated annealing—stone and hill.

The components required for annealing algorithm are the following.

1. *A basic system configuration:* The possible solution of a problem over which we search for a best (optimal) answer. (In a neural net, this is optimum steady-state weight.)
2. *The move set:* A set of allowable moves that permit us to escape from local minima and reach all possible configurations.
3. *A cost function* associated with the error function.
4. *A cooling schedule:* Starting of the cost function and rules to determine when it should be lowered and by how much, and when annealing should be terminated.

Simulated annealing networks can be used to make a network converge to its global minimum.

6.3 Boltzmann Machine

The early optimization technique used in artificial neural networks is based on the Boltzmann machine. When the simulated annealing process is applied to the discrete Hopfield network, it becomes a Boltzmann machine. The network is configured as the vector of the states of the units, and the states of the units are binary valued with probabilistic state transitions. The Boltzmann machine described in this section has fixed weights w_{ij}. On applying the Boltzmann machine to a constrained optimization problem, the weights represent the constraints of the problem and the quantity to be optimized. The discussion here is based on the fact of maximization of a consensus function (CF).

The Boltzmann machine consists of a set of units (X_i and X_j) and a set of bi-directional connections between pairs of units. This machine can be used as an associative memory. If the units X_i and X_j are connected, then $w_{ij} \neq 0$. There exists symmetry in the weighted interconnections based on the directional nature. It can be represented as $w_{ij} = w_{ji}$. There also may exist a self-connection for a unit (w_{ii}). For unit X_i, its state x_i may be either 1 or 0. The objective of the neural net is to maximize the CF given by

$$CF = \sum_i \sum_{j \leq i} w_{ij} x_i x_j$$

The maximum of the CF can be obtained by letting each unit attempt to change its state (alter between "1" and "0" or "0" and "1"). The change of state can be done either in parallel or sequential manner. However, in this case all the description is based on sequential manner. The consensus change when unit X_i changes its state is given by

$$\Delta \text{CF}(i) = (1 - 2x_i)\left(w_{ij} + \sum_{j \neq i} w_{ij}x_i\right)$$

where x_i is the current state of unit X_i. The variation in coefficient $(1 - 2x_i)$ is given by

$$(1 - 2x_i) = \begin{cases} +1, & X_i \text{ is currently off} \\ -1, & X_i \text{ is currently on} \end{cases}$$

If unit X_i were to change its activations then the resulting change in the CF can be obtained from the information that is local to unit X_i. Generally, X_i does not change its state, but if the states are changed, then this increases the consensus of the net. The probability of the network that accepts a change in the state for unit X_i is given by

$$\text{AF}(i, T) = \frac{1}{1 + \exp[-\Delta\, \text{CF}(i)/T]}$$

where T (temperature) is the controlling parameter and it will gradually decrease as the CF reaches the maximum value. Low values of T are acceptable because they increase the net consensus since the net accepts a change in state. To help the net not to stick with the local maximum, probabilistic functions are used widely.

6.3.1 Architecture

The architecture of a Boltzmann machine is represented through a two-dimensional array of the units in Figure 6-3. The units within each row and column are fully interconnected. The weights on the interconnections are given by $-p$ where $(p > 0)$. Also, there exists a self-connection for each unit, with weight $b > 0$. Unit X_{ij} is the common unit on which our discussion is based. The weights present on the interconnections are inhibitory.

6.3.2 Algorithm

6.3.2.1 Setting the Weights of the Network

The weights of a Boltzmann machine are fixed; hence there is no specific training algorithm for updation of weights. (For a Boltzmann machine with learning, there exists a training procedure.) With the Boltzmann machine weights remaining fixed, the net makes its transition toward maximum of the CF.

As shown in Figure 6-3, each unit is connected to every other unit in the same row and same column by the weights $-p(p > 0)$. The weights indicate the penalties obtained due to the violation that occurs when more than one unit is "on" in each row and column. There also exists a self-connection for each unit given by weight $b > 0$. The self-connection is a bonus given to the unit to turn on if it can do so without causing more than one unit to be on in each row and column. The net function will be desired if $p > b$. If unit X_{ij} is "off" and no other unit in its row or column is "on" then changing the status of X_{ij} to on increases the consensus of the net by amount b. This is an acceptable change because this increases the consensus. As a result, the net accepts it instead of rejecting it.

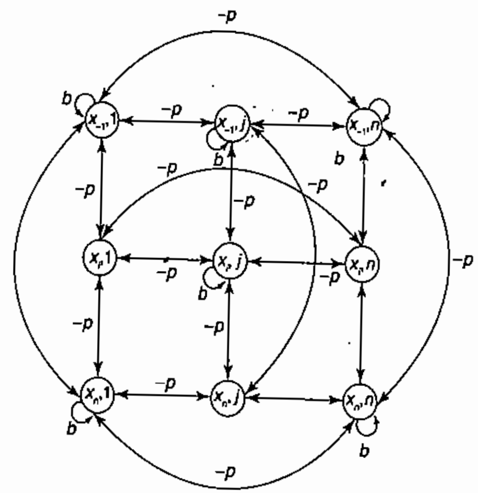


Figure 6-3 Architecture of Boltzmann machine.

On the other hand, if any one unit in each row or column of X_{ij} is on then changing the state of X_{ij} to on effects a change in consensus by $(b - p)$. Hence $(b - p) < 0$ makes $p > b$, i.e., the effect decreases the consensus. The net rejects this situation.

6.3.2.2 Testing Algorithm

It is assumed here that the units are arranged in a two-dimensional array. There are n^2 units. So, the weights between unit $X_{i,j}$ and unit $X_{I,J}$ are denoted by $w(i, j : I, J)$.

$$w(i, j : I, J) = \begin{cases} -p & \text{if } i = I \text{ or } j = J \text{ (but not both)} \\ b & \text{otherwise} \end{cases}$$

The testing algorithm is as follows.

Step 0: Initialize the weights representing the constraints of the problem. Also initialize control parameter T and activate the units.

Step 1: When stopping condition is false, perform Steps 2–8.

Step 2: Perform Steps 3–6 n^2 times. (This forms an epoch.)

Step 3: Integers I and J are chosen random values between 1 and n. (Unit $U_{I,J}$ is the current victim to change its state.)

Step 4: Calculate the change in consensus:

$$\Delta \text{CF} = (1 - 2X_{I,J})\left[w(I,J : I,J) + \sum_{i,j \neq} \sum_{I,J} w(i,j : I,J)X_{I,J}\right]$$

Step 5: Calculate the probability of acceptance of the change in state:

$$AF(T) = 1/1 + \exp[-(\Delta CF/T)]$$

Step 6: Decide whether to accept the change or not. Let R be a random number between 0 and 1. If $R < AF$, accept the change:
$X_{I,J} = 1 - X_{I,J}$ (This changes the state $U_{I,J}$.)
If $R \geq AF$, reject the change.

Step 7: Reduce the control parameter T.

$$T(\text{new}) = 0.95\ T(\text{old})$$

Step 8: Test for stopping condition, which is:
If the temperature reaches a specified value or if there is no change of state for specified number of epochs then stop, else continue.

The initial temperature should be taken large enough for accepting the change of state quickly. Also remember that the Boltzmann machine can be applied for various optimization problems such as traveling salesman problem.

6.4 Gaussian Machine

Gaussian machine is one which includes Boltzmann machine, Hopfield net and other neural networks. The Gaussian machine is based on the following three parameters: (a) a slope parameter of sigmoidal function α, (b) a time step Δt, (c) temperature T.

The steps involved in the operation of the Gaussian net are the following:

Step 1: Compute the net input to unit X_i:

$$\text{net}_i = \sum_{j=1}^{N} w_{ij} v_j + \theta_i + \epsilon$$

where θ_i is the threshold and ϵ the random noise which depends on temperature T.

Step 2: Change the activity level of unit X_i:

$$\frac{\Delta x_i}{\Delta t} = -\frac{x_i}{t} + \text{net}_i$$

Step 3: Apply the activation function:

$$v_i = f(x_i) = 0.5[1 + \tanh(x_i)]$$

where the binary step function corresponds to $\alpha = \infty$ (infinity).

The Gaussian machine with $T = 0$ corresponds the Hopfield net. The Boltzmann machine can be obtained by setting $\Delta t = \tau = 1$ to get

$$\Delta x_i = -x_i + \text{net}_i$$

$$\text{or} \quad x_i(\text{new}) = \text{net}_i = \sum_{j=1}^{N} w_{ij} v_j + \theta_i + \epsilon$$

The approximate Boltzmann acceptance function is obtained by integrating the Gaussian noise distribution

$$\int_0^{\infty} \frac{1}{\sqrt{2\pi\sigma^2}} \exp \frac{(x - x_i^2)}{2\sigma^2} dx \approx AF(t, T) = \frac{1}{1 + \exp(-x_i/T)}$$

where $x_i = \Delta CF(i)$. The noise which is found to obey a logistic rather than a Gaussian distribution produces a Gaussian machine that is identical to Boltzmann machine having Metropolis acceptance function, i.e., the output set to 1 with probability,

$$AF(i, T) = \frac{1}{1 + \exp(-x_i/T)}$$

This does not bother about the unit's original state. When noise is added to the net input of a unit then using probabilistic state transition gives a method for extending the Gaussian machine into Cauchy machine.

6.5 Cauchy Machine

Cauchy machine can be called fast simulated annealing, and it is based on including more noise to the net input for increasing the likelihood of a unit escaping from a neighborhood of local minimum. Larger changes in the system's configuration can be obtained due to the unbounded variance of the Cauchy distribution. Noise involved in Cauchy distribution is called "colored noise" and the noise involved in the Gaussian distribution is called "white noise."

By setting $\Delta t = \tau = 1$, the Cauchy machine can be extended into the Gaussian machine, to obtain

$$\Delta x_i = -x_i + \text{net}_i$$

$$\text{or} \quad x_i(\text{new}) = \text{net}_i = \sum_{j=1}^{N} w_{ij} v_j + \theta_i + \epsilon$$

The Cauchy acceptance function can be obtained by integrating the Cauchy noise distribution:

$$\int_0^{\infty} \frac{1}{\pi} \frac{T\,dx}{T^2 + (x - x_i)^2} = \frac{1}{2} + \frac{1}{\pi} \arctan\left(\frac{x_i}{T}\right) = AF(i, T)$$

where $x_i = \Delta CF(i)$. The cooling schedule and temperature have to be considered in both Cauchy and Gaussian machines.

6.6 Probabilistic Neural Net

The probabilistic neural net is based on the idea of conventional probability theory, such as Bayesian classification and other estimators for probability density functions, to construct a neural net for classification. This net instantly approximates optimal boundaries between categories. It assumes that the training

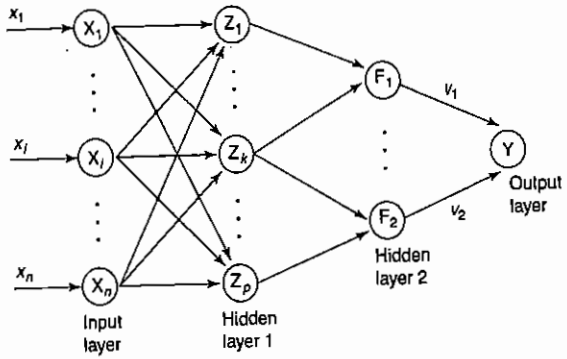


Figure 6-4 Probabilistic neural network.

data are original representative samples. The probabilistic neural net consists of two hidden layers as shown in Figure 6-4. The first hidden layer contains a dedicated node for each training pattern and the second hidden layer contains a dedicated node for each class. The two hidden layers are connected on a class-by-class basis, that is, the several examples of the class in the first hidden layer are connected only to a single matching unit in the second hidden layer.

During training process, the probabilistic neural net uses the training patterns for estimating the class probability distributions; each new input is classified according to the weighted average of the training sample which is very closer. The probabilistic neural net avoids the iterative process by simply storing the training patterns. Owing to this, probabilistic neural net learns very fast, but large networks are needed for large data sets.

The algorithm for the construction of the net is as follows:

Step 0: For each training input pattern $x(p), p = 1$ to P, perform Steps 1 and 2.

Step 1: Create pattern unit z_k (hidden-layer-1 unit). Weight vector for unit z_k is given by

$$w_k = x(p)$$

Unit z_k is either z-class-1 unit or z-class-2 unit.

Step 2: Connect the hidden-layer-1 unit to the hidden-layer-2 unit.
If $x(p)$ belongs to class 1, then connect the hidden layer unit z_k to the hidden layer unit F_1.
Otherwise, connect pattern hidden layer unit z_k to the hidden layer unit F_2.

The net can be used for classification when an example of a pattern from each class has been presented to it. The net's ability for generalization improves when it is trained on more examples.

6.7 Cascade Correlation Network

Cascade correlation is a network which builds its own architecture as the training progresses. This algorithm was proposed by Fahlman and Lebiere in 1990. Figure 6-5 shows the cascade correlation architecture. The network begins with some inputs and one or more output nodes, but it has no hidden nodes. Each and every input

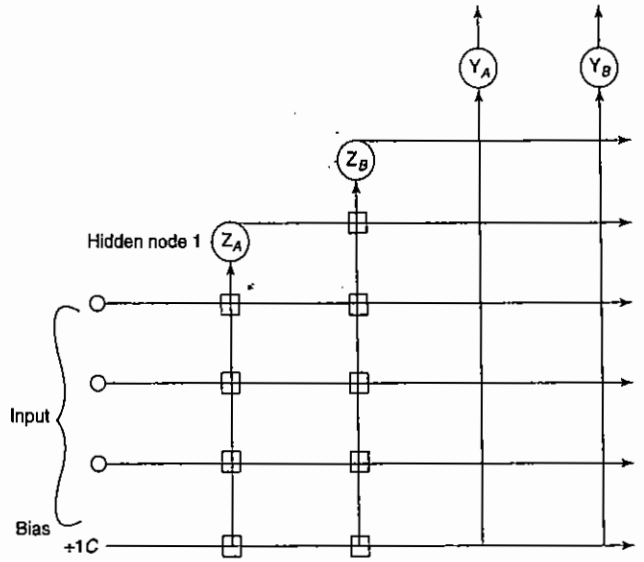


Figure 6-5 Cascade architecture after two hidden nodes have been added.

is connected to every output node. There may be linear units or some nonlinear activation function such as bipolar sigmoidal activation function in the output nodes. During training process, new hidden nodes are added to the network one by one. For each new hidden node, the correlation magnitude between the new node's output and the residual error signal is maximized. The connection is made to each node from each of the network's original inputs and also from every preexisting hidden node. During the time when the node is being added to the network, the input weights of the hidden nodes are frozen, and only the output connections are trained repeatedly. Each new node thus adds a new one-node layer to the network.

In Figure 6-5, the vertical lines sum all incoming activations. The rectangular boxed connections are frozen and "O" connections are trained continuously. In the beginning of the training, there are no hidden nodes, and the network is trained over the complete training set. Since there is no hidden node, a simple learning rule, Widrow–Hoff learning rule, is used for training. After a certain number of training cycles, when there is no significant error reduction and the final error obtained is unsatisfactory, we try to reduce the residual errors further by adding a new hidden node. For performing this task, we begin with a candidate node that receives trainable input connections from the network's external inputs and from all pre-existing hidden nodes. The output of this candidate node is not yet connected to the active network. After this, we run several numbers of epochs for the training set. We adjust the candidate node's input weights after each epoch to maximize C which is defined as

$$C = \sum_i \left| \sum_j (v_j - \bar{v})(E_{j,i} - \bar{E}_o) \right|$$

where i is the network output at which error is measured, j the training pattern, v the candidate node's output value, E_o the residual output error at node o, $\bar{v}$ the value of v averaged over all patterns, $\bar{E}_o$ the value of E_o

averaged over all patterns. The value "C" measures the correlation between the candidate node's output value and the calculated residual output error. For maximizing C, the gradient $\partial c/\partial w_i$ is obtained as

$$\frac{\partial c}{\partial w_i} = \sum_{j,i} \sigma_i (E_{j,i} - \bar{E}_i) d_j' I_{m,j}$$

where σ_i is the sign of the correlation between the candidate's value and output i; d_j' the derivative for pattern j of the candidate node's activation function with respect to sum of its inputs; $I_{m,j}$ the input the candidate node receives from node m for pattern j. When gradient $\partial c/\partial w_i$ is calculated, perform gradient ascent to maximize C. As we are training only a single layer of weights, simple delta learning rule can be applied. When C stops improving, again a new candidate can be brought in as a node in the active network and its input weights are frozen. Once again all the output weights are trained by the delta learning rule as done previously, and the whole cycle repeats itself until the error becomes acceptably small.

On the basis of this cascade correlation network, Fahlman (1991) proposed another training method for creating a recurrent network called the recurrent cascade correlation network. Its structure is same as shown in Figure 6-5, but each hidden node is a recurrent node, i.e., each hidden node has a connection to itself. Cascade correlation network is mainly suitable for classification problems. Even if modified, it can be used for approximation of functions.

6.8 Cognitron Network

Cognitron network was proposed by Fukushima in 1975. The learning hypothesis put forth by him is given in the following paragraphs.

The synaptic strength from cell X to cell Y is reinforced if and only if the following two conditions are true:

1. Cell X – presynaptic cell fires.
2. None of the postsynaptic cells present near cell Y fire stronger than Y.

The model developed by Fukushima was called cognitron as a successor to the perceptron which can perform cognizance of symbols from any alphabet after training. Figure 6-6 shows the connection between presynaptic cell and postsynaptic cell.

The cognitron network is a self-organizing multilayer neural network. Its nodes receive input from the defined areas of the previous layer and also from units within its own area. The input and output neural elements can take the form of positive analog values, which are proportional to the pulse density of firing biological neurons. The cells in the cognitron model use a mechanism of shunting inhibition, i.e., a cell is bound in terms of a maximum and minimum activities and is driven toward these extremities. The area from which the cell receives input is called connectable area. The area formed by the inhibitory cluster is called the vicinity area. Figure 6-7 shows the model of a cognitron. Since the connectable areas for cells in the same vicinity are defined to overlap, but are not exactly the same, there will be a slight difference appearing between the cells which is reinforced so that the gap becomes more apparent. Like this, each cell is allowed to develop its own characteristics.

Cognitron network can be used in neurophysiology and psychology. Since this network closely resembles the natural characteristics of a biological neuron, this is best suited for various kinds of visual and auditory information processing systems. However, a major drawback of cognitron net is that it cannot deal with the problems of orientation or distortion. To overcome this drawback, an improved version called neocognitron was developed.

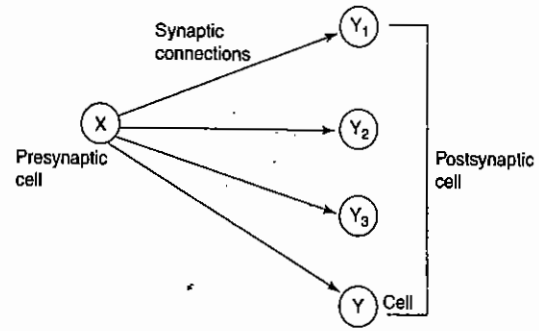


Figure 6-6 Connection between presynaptic cell and postsynaptic cell.

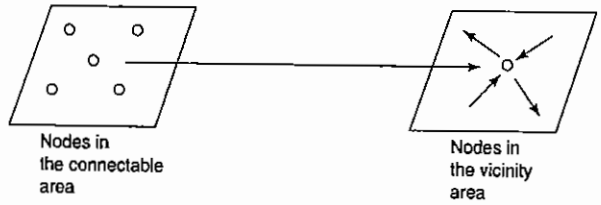


Figure 6-7 Model of a cognitron network.

6.9 Neocognitron Network

Neocognitron is a multilayer feed-forward network model for visual pattern recognition. It is a hierarchical net comprising many layers and there is a localized pattern of connectivity between the layers. It is an extension of cognitron network. Neocognitron net can be used for recognizing hand-written characters. A neocognitron model is shown in Figure 6-8.

The algorithm used in cognitron and neocognitron is same, except that neocognitron model can recognize patterns that are position-shifted or shape-distorted. The cells used in neocognitron are of two types:

1. *S-cell:* Cells that are trained suitably to respond to only certain features in the previous layer.

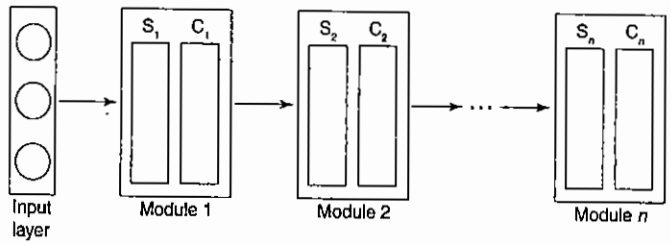


Figure 6-8 Neocognitron models.

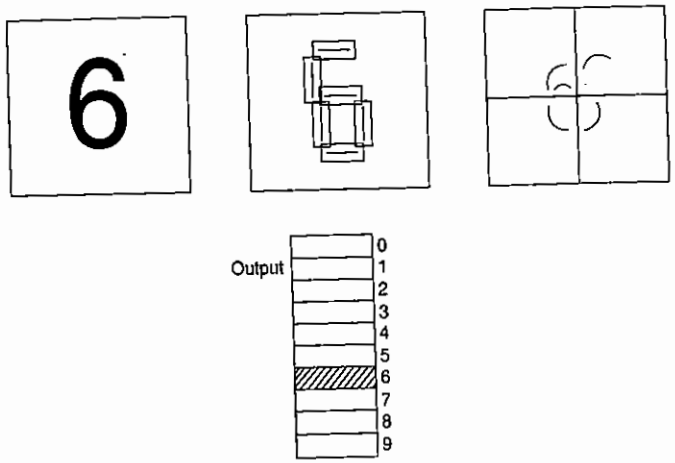


Figure 6-9 Spreading effect in neocognitron.

2. *C-cell:* A C-cell displaces the result of an S-cell in space, i.e., sort of "spreads" the features recognized by the S-cell.

Neocognitron net consists of many modules with the layered arrangement of S-cells and C-cells. The S-cells receive the input from the previous layer, while C-cells receive the input from the S-layer. During training, only the inputs to the S-layer are modified. The S-layer helps in the detection of specific features and their complexities. The feature recognized in the S_1 layer may be a horizontal bar or a vertical bar but the feature in the S_n layer may be more complex. Each unit in the C-layer corresponds to one relative position independent feature. For the independent feature, C-node receives the inputs from a subset of S-layer nodes. For instance, if one node in C-layer detects a vertical line and if four nodes in the preceding S-layer detect a vertical line, then these four nodes will give the input to the specific node in C-layer to spatially distribute the extracted features. Modules present near the input layer (lower in hierarchy) will be trained before the modules that are higher in hierarchy, i.e., module 1 will be trained before module 2 and so on.

The users have to fix the "receptive field" of each C-node before training starts because the inputs to C-node cannot be modified. The lower level modules have smaller receptive fields while the higher level modules indicate complex independent features present in the hidden layer. The spreading effect used in neocognitron is shown in Figure 6-9.

The S-layers are trained to respond to a particular pattern or group of patterns. The C-arrays then combine the results from related S-arrays and correspondingly thin out the number of units in each array. Training is found to progress layer by layer. The weights from the input units to the first layer are first trained and then frozen. Then the next trainable weights are adjusted and so on. When the net is designed, the weights between some layers are fixed as they are connection patterns.

6.10 Cellular Neural Network

The cellular neural network (CNN), introduced in 1988, is based on cellular automata, i.e., every cell in the network is connected only to its neighbor cells. Figures 6-10(A) and (B) show 2 × 2 CNN and 3 × 3 CNN,

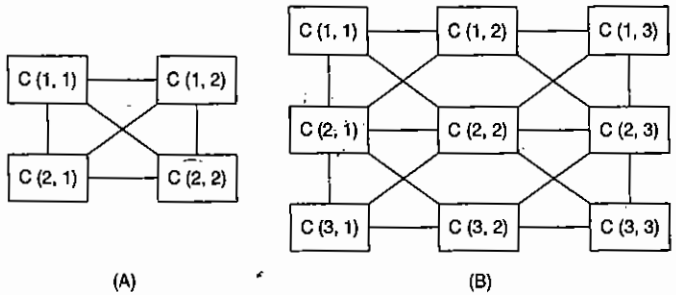


Figure 6-10 (A) A 2 × 2 CNN; (B) a 3 × 3 CNN.

respectively. The basic unit of a CNN is a cell. In Figures 6-10(A) and (B), C(1, 1) and C(2, 1) are called as cells.

Even if the cells are not directly connected with each other, they affect each other indirectly due to propagation effects of the network dynamics. The CNN can be implemented by means of a hardware model. This is achieved by replacing each cell with linear capacitors and resistors, linear and nonlinear controlled sources, and independent sources. An electronic circuit model can be constructed for a CNN. The CNNs are used in a wide variety of applications including image processing, pattern recognition and array computers.

6.11 Logicon Projection Network Model

Logicon projection network model (LPNM) is a learning process developed by researchers at Logicon. This model combines supervised and unsupervised training during the learning phases. When the unsupervised learning is used, the network learns quickly but not accurately. On the other hand, with supervised learning, it learns slowly but the error is minimized. The learning phase uses a feed-forward network with a hidden layer in between input and output layers. At the beginning of the learning phase, an unsupervised method such as Kohonen or ART is used to quickly initialize the weights of the network to some gross values, and then a supervised method like BPN may be used to finetune weight values. As the supervised method starts from "almost acceptable" solution, the network is claimed to converge quickly to a global minimum. Logicon claims that LPNM method is best than other methods. This network does not have to be reinitialized if more knowledge is to be added. Also, a network with some knowledge can be added to another network with different knowledge to obtain the sum of both.

6.12 Spatio-Temporal Connectionist Neural Network

Spatio-temporal connectionist neural network (STCNN) characterizes connectionist approaches for learning input–output relationships in which the data is distributed across space and time–spatio-temporal patterns. An STCNN is defined as a parallel distributed information processing structure which is capable of dealing with input data presented across both time and space. In STCNN, input and output patterns vary across time as well as space. For analyzing the network's performance, it is useful to discretize the temporal dimension by sampling at regular intervals. The system considered here produces the response when the time proceeds by intervals of Δt. Symbol "t" may be used to represent a particular point in time. Here Δt can be considered

as the unit measure for quantity t or some small variations in t. In STCNN, even a continuous time system is converted into a set of first-order difference equations, making it to be in the form of discrete time systems.

The time dimension in STCNN differs from the spatial dimensions in conventional connectionist networks. Components of an input pattern distributed across space can be accessed at the same time. However, only the current components of patterns distributed along the time are accessible at any given instant. The input vector for an STCNN at instant t is denoted by input vector $\bar{x}(t)$. This vector is supplied to STCNN at time t by setting the activation values of the input units of the STCNN to the components of the vector. Hence, input vector can be considered as a stimulus.

The conventional and spatio-temporal networks are equipped with memory in the form of connection weights denoted by one or more matrices depending on the number of layers of connections present in the network. These are updated after each training step and constitute a memory of all previous training. Assuming this memory extends back past the current input pattern, all the way to the first training step, we refer to the weights as long-term memory. After a connectionist network has been successfully trained, this long-term memory remains fixed during the operation of the network.

Along with these weight matrices, some networks also use other trainable parameters. These parameters may represent either of the three mentioned below:

1. connectivity scheme of the network;
2. types of transmission delays associated with connections;
3. initial activation values of the internal processing elements.

These parameters also form a part of the long-term memory. We define "W" to denote an n-tuple representing all the adaptable parameters of the network. This n-tuple includes one or more weight matrices, and also may contain connectivity scheme parameters, transmission delay parameters and initial activation values depending on the type of the network.

The STCNNs also include a short-term memory. This memory allows these networks to deal with input and output patterns that vary across time and thus defines them as STCNNs. Conventional connectionist networks compute the activation values of all the nodes at time t based only on the input at time t. On the other hand, in STCNNs the activations of some nodes at time t are computed on the basis of the activations at time $(t-1)$ or earlier. These activations serve as short-term memory. The state vector $\bar{s}(t-1)$ is used here to represent the activations at time $(t-1)$ of those nodes that are used to compute the activations of other nodes at a time t, i.e., state nodes. The long-term memory is stored in connection weights (which are updated only during training) while the short-term memory is represented by node activations (which are computed with each time step even after training).

STCNNs encode their output responses in the activations of a special set of units called output units. The output of STCNN is represented by vector $\bar{y}(t)$. Most connectionist networks learn by computing the difference between their response and desired (ideal) response and adjusting their long-term memory suitably. The desired response is denoted by $\bar{y}_d(t)$. The difference between the desired output vector and the actual output vector is the error vector $\bar{E}(t) = \bar{y}_d(t) - \bar{y}(t)$, and the total network error, ε, is defined as the one-half of the square of the magnitude of this vector, i.e.,

$$\varepsilon = \sum_{t \geq 0} \left\{ \frac{1}{2} \left\| \bar{E}(t) \right\|^2 \right\}$$

The total error, given by ε, is the measure of overall performance. It is this quantity that is minimized via gradient descent during training. STCNNs are applicable to dynamical system identification and control, syntactic pattern recognition and grammatical induction. There are various taxonomies that are being developed for STCNNs.

6.13 Optical Neural Networks

Optical neural networks interconnect neurons with light beams. Owing to this interconnection, no insulation is required between signal paths and the light rays can pass through each other without interacting. The path of the signal travels in three dimensions. The transmission path density is limited by the spacing of light sources, the divergence effect and the spacing of detectors. As a result, all signal paths operate simultaneously, and true data rate results are produced. In holograms with high density, the weighted strengths are stored. These stored weights can be modified during training for producing a fully adaptive system. There are two classes of this optical neural network. They are:

1. electro-optical multipliers;
2. holographic correlators.

6.13.1 Electro-Optical Multipliers

Electro-optical multipliers, also called electro-optical matrix multipliers, perform matrix multiplication in parallel. The network speed is limited only by the available electro-optical components; here the computation time is potentially in the nanosecond range. A model of electro-optical matrix multiplier is shown in Figure 6-11.

Figure 6-11 shows a system which can multiply a nine-element input vector by a 9×7 matrix, which produces a seven-element NET vector. There exists a column of light sources that passes its rays through a lens; each light illuminates a single row of weight shield. The weight shield is a photographic film where transmittance of each square (as shown in Figure 6-11) is proportional to the weight. There is another

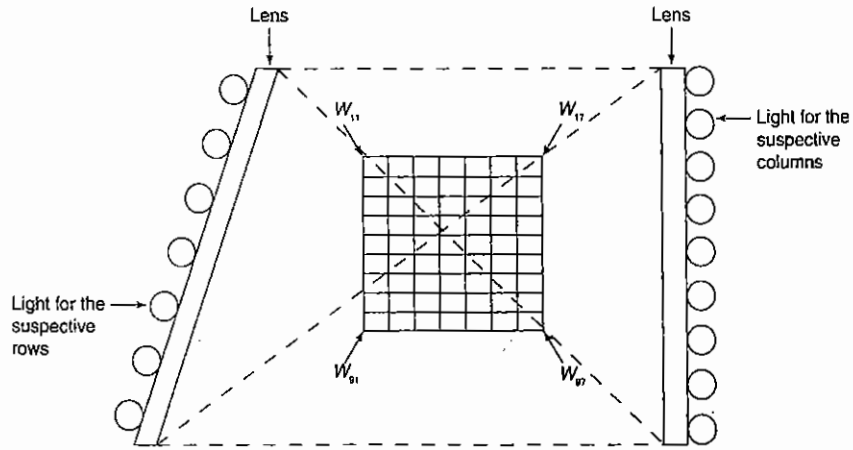


Figure 6-11 Electro-optical multiplier.

lens that focuses the light from each column of the shield to a corresponding photoelector. The NET is calculated as

$$\text{NET}_k = \sum_i w_{ik} x_i$$

where NET_k is the net output of neuron k; w_{ik} the weight from neuron i to neuron k; x_i the input vector component i. The output of each photodetector represents the dot product between the input vector and a column of the weight matrix. The output vector set is equal to the product of the input vector with weight matrix. Hence, matrix multiplication is performed parallely. The speed is independent of the size of the array. So, the network is sealed up without increasing the time required for computation. Variable weights may be designed for use in the adaptive system. A liquid crystal light valve instead of photographic film may be used for weights. This makes the weights to get adjusted electronically. This type of electro-optical multiplier can be used in Hopfield net and bidirectional associative memory.

6.13.2 Holographic Correlators

In holographic correlators, the reference images are stored in a thin hologram and are retrieved in a coherently illuminated feedback loop. The input signal, either noisy or incomplete, may be applied to the system and can simultaneously be correlated optically with all the stored reference images. These correlations can be threshold and are fed back to the input, where the strongest correlation reinforces the input image. The enhanced image passes around the loop repeatedly, which approaches the stored image more closely on each pass, until the system gets stabilized on the desired image. The best performance of optical correlators is obtained when they are used for image recognition. A generalized optical image recognition system with holograms is shown in Figure 6-12.

The system input is an image from a laser beam. This passes through a beam splitter, which sends it to the threshold device. The image is reflected, then gets reflected from the threshold device, passes back to the beam splitter, then goes to lens 1, which makes it fall on the first hologram. There are several stored images

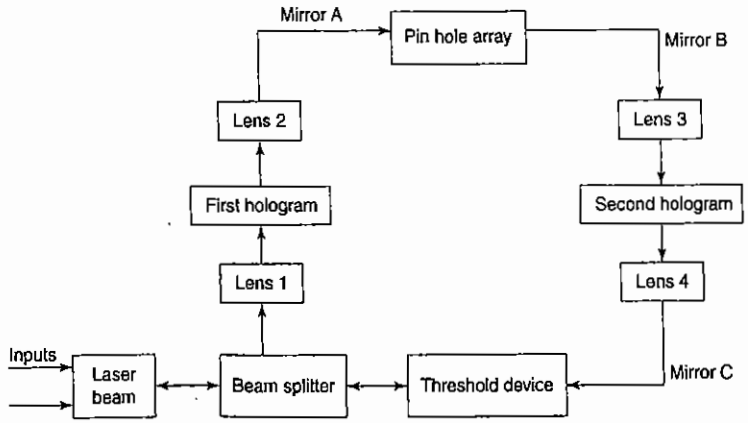


Figure 6-12 Optical image recognition system.

in first hologram. The image then gets correlated with each stored image. This correlation produces light patterns. The brightness of the patterns varies with the degree of correlation. The projected images from lens 2 and mirror A pass through pinhole array, where they are spatially separated. From this array, light patterns go to mirror B through lens 3 and then are applied to the second hologram. Lens 4 and mirror C then produce superposition of the multiple correlated images onto the back side of the threshold device.

The front surface of the threshold device reflects most strongly that pattern which is brightest on its rear surface. Its rear surface has projected on it the set of four correlations of each of the four stored images with the input image. The stored image that is similar to the input image possesses highest correlation. This reflected image again passes through the beam splitter and re-enters the loop for further enhancement. The system gets converged on the stored patterns most like the input pattern.

Here we have discussed the basic operation of the holographic optical image recognition system. Employing hologram correlator, we can design Hopfield network. Optical neural networks are more advantageous in terms of speed and interconnect density. They can virtually construct any network architecture.

6.14 Neuroprocessor Chips

Neural networks implemented in hardware can take advantage of their inherent parallelism and run orders of magnitude faster than software simulations. There exists a wide variety of commercial neural network chips and neurocomputers.

The probabilistic RAM, pRAM-256 Very Large Scale Integrated (VLSI) neural network processor, was developed by the Electrical Engineering Department of King's College, London. pRAM has 256 reconfigurable neurons, each with six inputs. Its on-chip learning unit utilizes reinforced learning where learning can be global, local or competitive. The external static RAM of pRAM stores the synaptic weights. The p-RAM possesses both stochastic and nonlinear aspects of biological neurons in a typical manner, which allows exploitation of hardware.

The Neuro Accelerator Chip (NAC) was developed in 1992 by the Information Defence Division, Australian Defence Science and Technology Organization. It is made up of an array of 16-component 10-bit integer processing elements that can be cascaded in two dimensions with necessary control signals. Each processing element multiplies its input by one of 16 weights preloaded in dual port registers and accumulates the results to 23-bit precision at a rate of 500 million operations per second. The NAC can be hard wired to implement various neural networks.

Neural Network Processor (NNP), developed by Accurate Automation Corporation, uses a multiple instruction multiple data architecture capable of running multiple chips in parallel without performance degradation. Each chip houses high-speed 16-bit processor with on-chip storage for synaptic weights. Only nine assembly language instructions are executed by the processor. Communication among multiple NNPs is performed by interprocessor. NNP can be programmed to implement any particular neural network training algorithms. Its performance is 140 MCPS for a single chip and up to 1.4 GCPS for a 10-processor system.

The CNAPS system, developed by Adaptive Solutions, is mainly based on CNAPS-1064 digital parallel processor chip that has 64 sub-processors operating in SIMD mode. Each sub-processor can emulate one or more neurons, and multiple chips can be ganged together. The CNAPS/PC 1SA card uses 1, 2 or 4 of new CNAPS-1016 parallel processor chips or two of the 1064 chips to obtain 16, 32, 64 or 128 CNAPS processors. Learning algorithms can be programmed. It can be noted that back propagation and several other algorithms come in the Build Net package. Here, back propagation feed-forward performs 1.16 billion multiply/accumulates/second and 293 million weight updates/second with 1 chip and

5.80 1.16 billion multiply/accumulates/second /1.95 million weight updates/second, respectively, with four chips.

The IBM ZISC 036 is a digital chip with 64-component 8-bit inputs and 36 radial basis function neurons. Multiple chips can be easily cascaded to create networks of arbitrary size. Here input vector V is compared to store prototype vector ϕ for each neuron. It takes 3.5 μs to load 64 elements and another 0.5 μs for the classification signal to appear. Learning processing of a vector takes about another 2 μs beyond 4 μs for loading and evaluation. Its performance at 16 MHz, 4 μs classification of a 64-component 8-bit vector.

The INTEL 80170NX Electrically Trainable Analog Neural Network (ETANN) is one with 64 inputs (0-3v), 16 internal biases, and 64 neurons with sigmoidal transfer functions. Two-layer feed-forward networks can be implemented with 64 inputs, 64 hidden neurons, and 64 output neurons using the two 80 × 64 weight matrices. Hidden layer outputs are clocked back through second weight matrix to perform output layer processing. Instead of this, a single 64-layer network with 128 inputs can be implemented using both matrices and clocking in two sets of 64 inputs. Weights possess 6-bit precision and are stored in nonvolatile floating gate synapses. There is no on-chip learning. Emulation is performed in software and the weights are downloaded to the chip. In this case, about 8 μs propagation time is taken for a two-layer network. This is equivalent to roughly 2 billion multiply/accumulates per second.

MCE MT 19003 Neural Instruction Set Processor is a digital processor chip using signed 12-bit internal neuron values, with 16-bit multiplier and 35-bit accumulator. Network input values, bias values, synapse and neuron values are held in off-chip memory. The network processing is also guided by a given program in off-chip memory using seven-element instruction set. Neuron values can be sealed by a transfer function using four available tables. This processor also has no on-chip learning. Its performance is 1 synapse per clock cycle.

RC Module Neuro Processor NM6403 is a high-performance microprocessor with super scalar architecture. The architecture includes control unit, address calculation and scalar processing units, node to support vector operations with elements of variable bit length. There is no on-chip learning in this processor. Its performance is

1. *Scalar operations:* 50 MIPS, 50 MIPS for 32 bit data.
2. *Vector operations:* 1.2 billion multiplications and additions/second.

Nestor N11000 is a network with Radial Basis function neurons. During its learning, prototype vectors are stored under the assumption that they are picked randomly from original parent distributions. Here up to 1024 prototypes can be stored. Each prototype is then assigned to a given middle layer neuron. This middle layer neuron is assigned to an output neuron that represents the particular class for that vector. All middle layer neurons that correspond to the same class are designed to same output neuron. In recall stage, an input vector is compared to each prototype parallely, and if the distance between them is above a given threshold, it fires, leading to firing of the corresponding output, or class, neuron. Here two on-chip learning algorithms are available:

1. Probabilistic neural net (PNN);
2. Restricted Coulomb energy (RCE).

Also, microcoding can be modified for user-defined algorithms. Its performance is 40K, 256 element patterns per second. There is National Semiconductors NeuFuz/COP8 Microcontroller processor, which uses a combination of neural network and fuzzy logic software to generate code for National's COP8 microcontrollers. Neural network can be used to learn the fuzzy based rules and membership functions. There exist several packages of it. Some are listed below:

1. *NeuFuz Learning Kit (NF2 – C8A – Kit):* A neural network PC/AT software (2 inputs, 1 output) and fuzzy rule and membership function generator (max 3 membership functions), COP8 code generator and COP8 assembler/linker.
2. *NeuFuz 4 (NF2 – C8A):* Neural network PC/AT software (4 inputs, 1 output) and fuzzy rule and membership function generator (max 3 member functions), COP8 code generator and COP8 assm/linker.
3. *NeuFuz 4 Development System (NF2 – C8A):* Neural network PC/AT software (4 inputs, 1 output) and fuzzy rule membership function generators (max 3 member functions), COP8 code generator, COP8 assembler/linker and COP8 in-circuit emulator with PROM programming.

Learning performed here is only software learning. Apart from the above listed chips, there are several other neuroprocessor chips. Besides, a wide variety of research is going on for further development of neural network hardware.

6.15 Summary

In this chapter, we have discussed certain specific networks based on their special characteristics and performance. The networks are designed for optimization problems and classifications. Certain nets discussed use Bayesian decision making method and hierarchical arrangement of units. The variations of Boltzmann machine, which include Gaussian and Cauchy nets were also discussed. Besides, our discussion focused on the cognitron and neocognitron networks, which are used for recognition of hand written characters. Other networks discussed include spatio-temporal neural network, annealing network, optical neural nets, cellular neural nets, and Logicon neural nets. To give the reader an idea of neural network hardware, a few neuroprocessor chips have also been listed.

6.16 Review Questions

1. List a few special neural networks designed for typical applications.
2. What is the principle behind simulated annealing network?
3. How is Boltzmann machine used in constrained optimization problems?
4. With a neat architectural diagram explain the application procedure used in Boltzmann machine.
5. Write short note on Gaussian and Cauchy machines.
6. What is the importance of probabilistic neural network?
7. With an architectural diagram, explain the probabilistic neural network.
8. Discuss the algorithm used in probabilistic neural network.
9. How does cascade correlation network build its network as the training progress?
10. Justify that cascade correlation network is a hierarchical network.
11. Compare and contrast presynaptic and postsynaptic cells in cognitron model.
12. Explain the working principle of cognitron network.
13. What is the drawback of cognitron net?
14. Write short note on neocognitron model, stating how does it overcome the drawback of cognitron model.

15. Describe the working methodology in cellular neural network.
16. In what way are the supervised and unsupervised learning methods combined to obtain high performance in Logicon projection network?
17. Discuss in detail the spatio-temporal connectiorist neural network.
18. State the principle of optical neural networks.
19. Briefly explain the concept involved in electro-multiplier networks and holographic correlators.
20. Mention a few latest neuroprocessor chips.

Introduction to Fuzzy Logic, Classical Sets and Fuzzy Sets

7

Learning Objectives

- Definition of classical sets and fuzzy sets.
- The various operations and properties of classical and fuzzy sets.
- How functional mapping of crisp set can be carried out.
- Solved problems performing the operations and properties of fuzzy sets.

7.1 Introduction to Fuzzy Logic

In general, the entire real world is complex, and the complexity arises from uncertainty in the form of ambiguity. One should closely look into the real-world complex problems to find an accurate solution, amidst the existing uncertainties, using certain methodologies. Henceforth, the growth of fuzzy logic approach, to handle ambiguity and uncertainty existing in the complex problems. In general, fuzzy logic is a form of multi-valued logic to deal with reasoning that is approximate rather than precise. This is in contradiction with "crisp logic" that deals with precise values. Also, binary sets have binary or Boolean logic (either 0 or 1), which finds solution to a particular set of problems. Fuzzy logic variables may have a truth value that ranges between 0 and 1 and is not constrained to the two truth values of classic propositional logic. Also, as linguistic variables are used in fuzzy logic, these degrees have to be managed by specific functions.

As the complexity of a system increases, it becomes more difficult and eventually impossible to make a precise statement about its behavior, eventually arriving at a point of complexity where the fuzzy logic method born in humans is the only way to get at the problem.

(Originally identified and set forth by Lotfi A. Zadeh, Ph.D., University of California, Berkeley)

Fuzzy logic, introduced in the year 1965 by Lotfi A. Zadeh, is a mathematical tool for dealing with uncertainty. Dr. Zadeh states that the Principle of complexity and imprecision are correlated: "The closer one looks at a real world problem, the fuzzier becomes its solution." Fuzzy logic offers soft computing paradigm the important concept of computing with words. It provides a technique to deal with imprecision and information granularity. The fuzzy theory provides a mechanism for representing linguistic constructs such as "high," "low," "medium," "tall," "many." In general, fuzzy logic provides an inference structure that enables appropriate human reasoning capabilities. On the contrary, the traditional binary set theory describes crisp events, that is, events that either do or do not occur. It uses probability theory to explain if an event will occur, measuring the chance with which a given event is expected to occur. The theory of fuzzy logic is based upon the notion of relative graded membership and so are the functions of cognitive processes. The utility

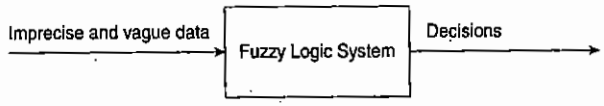


Figure 7-1 A fuzzy logic system accepting imprecise data and providing a decision.

of fuzzy sets lies in their ability to model uncertain or ambiguous data and to provide suitable decisions as in Figure 7-1.

Though fuzzy logic has been applied to many fields, from control theory to artificial intelligence, it still remains controversial among most statisticians, who prefer Bayesian logic, and some control engineers, who prefer traditional two-valued logic. In fuzzy systems, values are indicated by a number (called a truth value) ranging from 0 to 1, where 0.0 represents absolute falseness and 1.0 represents absolute truth. While this range evokes the idea of probability, fuzzy logic and fuzzy sets operate quite differently from probability.

Fuzzy sets that represent fuzzy logic provide means to model the uncertainty associated with vagueness, imprecision and lack of information regarding a problem or a plant or a system, etc. Consider the meaning of a "short person". For an individual X, a short person may be one whose height is below 4'25". For other individual Y, a short person may be one whose height is below or equal to 3'90". The word "short" is called a linguistic descriptor. The term "short" provides the same meaning to individuals X and Y, but it can be seen that they both do not provide a unique definition. The term "short" would be conveyed effectively only when a computer compares the given height value with the pre-assigned value of "short". This variable "short" is called as *linguistic variable* which represents the imprecision existing in the system.

The basis of the theory lies in making the membership function lie over a range of real numbers from 0.0 to 1.0. The fuzzy set is characterized by (0.0,0,1.0). Real world is vague and assigning rigid values to linguistic variables means that some of the meaning and semantic value is invariably lost. The uncertainty is found to arise from ignorance, from chance and randomness, due to lack of knowledge, from vagueness (unclear), like the fuzziness existing in our natural language. Dr. Zadeh proposed the *set membership* idea to make suitable decisions when uncertainty occurs. Consider the "short" example discussed previously. If we take "short" as a height equal to or less than 4 feet, then 3'90" would easily become the member of the set "short" and 4'25" will not be a member of the set "short." The membership value is "1" if it belongs to the set and "0" if it is not a member of the set. Thus membership in a set is found to be binary, that is, either the element is a member of a set or not. It can be indicated as

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

where $\chi_A(x)$ is the membership of element x in the set A and A is the entire set on the universe.

If it is said that the height is 5'6" (or 168 cm), one might think a bit before deciding whether to consider it as short or not short (i.e., tall). Moreover, one might reckon it as short for a man but tall for a woman. Let's make the statement "John is short", and give it a truth value of 0.70. If 0.70 represented a probability value, it would be read as "There is a 70% chance that John is short," meaning that it is still believed that John is either short or not short, and there exists 70% chance of knowing which group he belongs to. But fuzzy terminology actually translates to "John's degree of membership in the set of short people is 0.70," by which it is meant that if all the (fuzzy set of) short people are considered and lined up, John is positioned 70% of the way to the shortest. In conversation, it is generally said that John is "kind of" short and recognize that there is no definite demarcation between short and tall. This could be stated mathematically as μSHORT(Russell) = 0.70, where μ is the membership function.

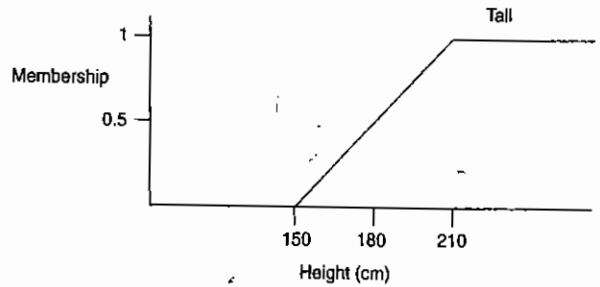


Figure 7-2 Graph showing membership functions for fuzzy set "tall."

Fuzzy logic operates on the concept of membership. For example, the statement "Elizabeth is old" can be translated as Elizabeth is a member of the set of old people and can be written symbolically as μ(OLD), where μ is the membership function that can return a value between 0.0 and 0.1 depending on the degree of membership. In Figure 7-2, the objective term "tall" has been assigned fuzzy values. At 150 cm and below, a person does not belong to the fuzzy class while for above 180, the person certainly belongs to category "tall." However, between 150 and 180 cm, the degree of membership for the class "tall" can be assigned from the curve varying linearly between 0 and 1. The fuzzy concept "tallness" can be extended into "short," "medium" and "tall" as shown in Figure 7-3. This is different from the probability approach that gives the degree of probability of an occurrence of an event (Elizabeth being old, in this instance).

The membership was extended to possess various "degrees of membership" on the real continuous interval [0,1]. Zadeh formed *fuzzy sets* as the sets on the universe X which can accommodate "degrees of membership." The concept of a fuzzy set contrasts with the classical concept of a bivalent set (crisp set) whose boundary is required to be precise, that is, a crisp set is a collection of things for which it is known irrespective of whether any given thing is inside it or not. Zadeh generalized the idea of a crisp set by extending a valuation set {1,0} (definitely in/definitely out) to the interval of real values (degrees of membership) between 1 and 0, denoted as [0,1]. We can say that the degree of membership of any particular element of a fuzzy set expresses the degree of compatibility of the element with a concept represented by fuzzy set. It means that a fuzzy set A contains an object x to degree $a(x)$, that is, $a(x) = \text{Degree}(x \in A)$, and the map $a: X \rightarrow$ {Membership Degrees} is called a *set function* or a *membership function*. The fuzzy set A can be expressed as $A = \{(x, a(x))\}$, $x \in X$; it imposes an elastic constrain of the possible values of elements $x \in X$, called the *possibility distribution*. Fuzzy sets tend to

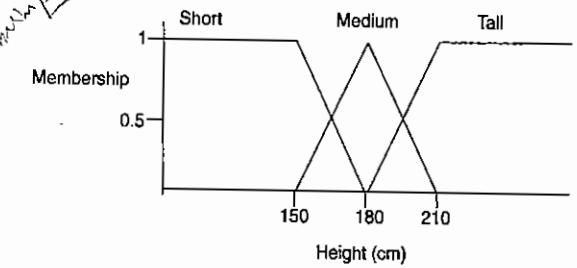


Figure 7-3 Graph showing membership functions for fuzzy sets "short," "medium" and "tall."

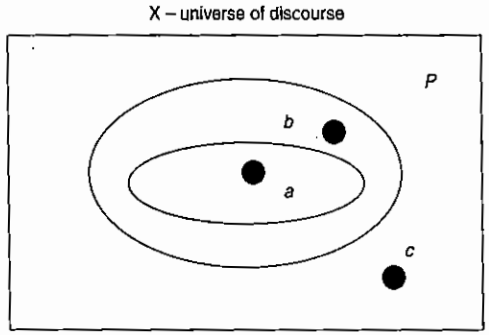


Figure 7-4 Boundaty region of a fuzzy set.

capture vagueness exclusively via membership functions that are mappings from a given universe of discourse X to a unit interval containing membership values. It is important to note that membership can take values between 0 and 1.

Fuzziness describes the ambiguity of an event and randomness describes the uncertainty in the occurrence of an event. It can be generally seen in classical sets that there is no uncertainty, hence they have crisp boundaries, but in the case of a fuzzy set, since uncertainty occurs, the boundaries may be ambiguously specified.

From Figure 7-4 it can be noted that "a" is clearly a member of fuzzy set P, "c" is clearly not a member of fuzzy set P and the membership of "b" is found to be vague. Hence "a" can take membership value 1, "c" can take membership value 0 and "b" can take membership value between 0 and 1 [0 to 1], say 0.4, 0.7, etc. This is said to be a partial membership of fuzzy set P.

The membership function for a set maps each element of the set to a membership value between 0 and 1 and uniquely describes that set. The values 0 and 1 describe "not belonging to" and "belonging to" a conventional set, respectively; values in between represent "fuzziness." Determining the membership function is subjective to varying degrees depending on the situation. It depends on an individual's perception of the data in question and does not depend on randomness. This concept is important and distinguishes fuzzy set theory from probability theory.

Fuzzy logic also consists of fuzzy inference engine or fuzzy rule-base to perform approximate reasoning somewhat similar to (but much more primitive than) that of the human brain. Computing with words seems to be a slightly futuristic phrase today since only certain aspects of natural language can be represented by the calculus of fuzzy sets; still fuzzy logic remains one of the most practical ways to mimic human expertise in a realistic manner. The fuzzy approach uses a premise that humans don't represent classes of objects (e.g. *"class of bald men"* or the *"class of numbers which are much greater than 50"*) as fully disjoint sets but rather as sets in which there may be grades of membership intermediate between full membership and non-membership. Thus, a fuzzy set works as a concept that makes it possible to *treat fuzziness in a quantitative manner*.

Fuzzy sets form the building blocks for fuzzy *IF–THEN* rules which have the general form *"IF X is A THEN Y is B,"* where A and B are fuzzy sets. The term "fuzzy systems" refers mostly to systems that are governed by fuzzy IF–THEN rules. The IF part of an implication is called the *antecedent* whereas the THEN part is called a *consequent*. A fuzzy system is a set of fuzzy rules that converts inputs to outputs. The basic configuration of a pure fuzzy system is shown in Figure 7-5. The fuzzy inference engine (algorithm) combines fuzzy *IF–THEN* rules into a mapping from fuzzy sets in the input space X to fuzzy sets in the output space

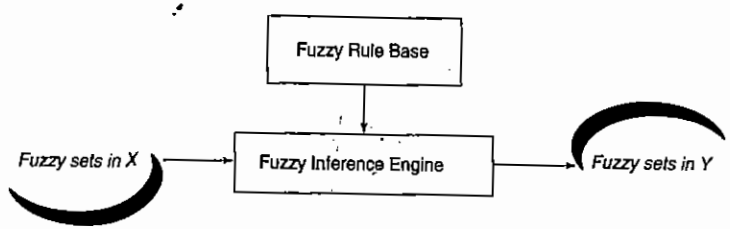


Figure 7-5 Configuration of a pure fuzzy system.

Y based on fuzzy logic principles. From a knowledge representation viewpoint, a fuzzy *IF–THEN* rule is a scheme for capturing knowledge that involves imprecision. The main feature of reasoning using these rules is its *partial matching* capability, which enables an inference to be made from a fuzzy rule even when the rule's condition is only partially satisfied.

Fuzzy systems, on one hand, are rule-based systems that are constructed from a collection of linguistic rules; on the other hand, fuzzy systems are nonlinear mappings of inputs (stimuli) to outputs (responses), that is, certain types of fuzzy systems can be written as compact nonlinear formulas. The inputs and outputs can be numbers or vectors of numbers. These rule-based systems can in theory model any system with arbitrary accuracy, that is, they work as *universal approximators*.

The Achilles' heel of a fuzzy system is its rules; smart rules give smart systems and other rules give less smart or even dumb systems. The *number of rules* increases exponentially with the dimension of the input space (number of system variables). This rule explosion is called the *curse of dimensionality* and is a general problem for mathematical models. For the last 5 years several approaches based on decomposition, (cluster) merging and fusing have been proposed to overcome this problem.

Hence, fuzzy models are not replacements for probability models. The fuzzy models are sometimes found to work better and sometimes they do not. But mostly fuzzy logic has evidently proved that it provides better solutions for complex problems.

7.2 Classical Sets (Crisp Sets)

Basically, a set is defined as a collection of objects, which share certain characteristics. A classical set is a collection of distinct objects. For example, the user may define a classical set of negative integers, a set of persons with height less than 6 feet, and a set of students with passing grades. Each individual entity in a set is called a member or an element of the set. The classical set is defined in such a way that the universe of discourse is splitted into two groups: members and nonmembers. Consider an object x in a crisp set A. This object x is either a member or a nonmember of the given set A. In case of crisp sets, no partial membership exists. A crisp set is defined by its characteristic function.

Let universe of discourse be U. The collection of elements in the universe is called whole set. The total number of elements in universe U is called cardinal number denoted by $n(U)$. Collections of elements within a universe are called sets, and collections of elements within a set are called subsets.

We know that for a crisp set A in universe U:

1. An object x is a member of given set A ($x \in A$), i.e., x belongs to A.
2. An object x is not a member of given set A ($x \notin A$), i.e., x does not belong to A.

There are several ways for defining a set. A set may be defined using one of the following:

1. The list of all the members of a set may be given. Example

$$A = \{2, 4, 6, 8, 10\}$$

2. The properties of the set elements may be specified. Example

$$A = \{x | x \text{ is prime number} < 20\}$$

3. The formula for the definition of a set may be mentioned. Example

$$A = \left\{ x_i = \frac{x_i + 1}{5}, i = 1 \text{ to } 10, \quad \text{where } x_i = 1 \right\}$$

4. The set may be defined on the basis of the results of a logical operation. Example

$$A = \{x | x \text{ is an element belonging to } P \text{ AND } Q\}$$

5. There exists a membership function, which may also be used to define a set. The membership is denoted by the letter μ and the membership function for a set A is given by (for all values of x)

$$\mu_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{if } x \notin A \end{cases}$$

The set with no elements is defined as an empty set or null set. It is denoted by symbol ϕ. The occurrence of an impossible event is denoted by a null set, and the occurrence of a certain event indicates a whole set. The set which consists of all possible subsets of a given set A is called a power set and is denoted as

$$P(A) = \{x | x \subseteq A\}$$

For crisp sets A and B containing some elements in universe X, the notations used are given below:

$$x \in A \Rightarrow x \text{ belongs to } A$$
$$x \notin A \Rightarrow x \text{ does not belong to } A$$
$$x \in X \Rightarrow x \text{ belongs to universe } X$$

For classical sets A and B on X, we also have some notations:

$$A \subset B \Rightarrow A \text{ is completely contained in } B \text{ (i.e., if } x \in A, \text{ then } x \in B)$$
$$A \subseteq B \Rightarrow A \text{ is contained in or is equivalent to } B$$
$$A = B \Rightarrow A \subseteq B \text{ and } B \subseteq A$$

7.2.1 Operations on Classical Sets

Classical sets can be manipulated through numerous operations such as union, intersection, complement and difference. All these operations are defined and explained in the following sections.

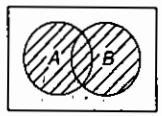


Figure 7-6 Union of two sets.

7.2.1.1 Union

The union between two sets gives all those elements in the universe that belong to either set A or set B or both sets A and B. The union operation can be termed as a logical OR operation. The union of two sets A and B is given as

$$A \cup B = \{x | x \in A \text{ or } x \in B\}$$

The union of sets A and B is illustrated by the Venn diagram shown in Figure 7-6.

7.2.1.2 Intersection

The intersection between two sets represents all those elements in the universe that simultaneously belong to both the sets. The intersection operation can be termed as a logical AND operation. The intersection of sets A and B is given by

$$A \cap B = \{x | x \in A \text{ and } x \in B\}$$

The intersection of sets A and B is represented by the Venn diagram shown in Figure 7-7.

7.2.1.3 Complement

The complement of set A is defined as the collection of all elements in universe X that do not reside in set A, i.e., the entities that do not belong to A. It is denoted by $\overline{A}$ and is defined as

$$\overline{A} = \{x | x \notin A, x \in X\}$$

where X is the universal set and A is a given set formed from universe X. The complement operation of set A is shown in Figure 7-8.

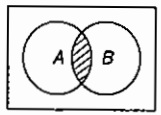


Figure 7-7 Intersection of two sets.

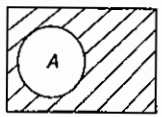


Figure 7-8 Complement of set A.

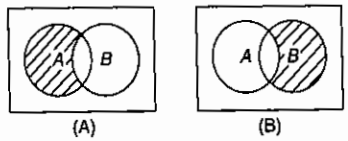


Figure 7-9 (A) Difference $A|B$ or $(A - B)$; (B) difference $B|A$ or $(B - A)$.

7.2.1.4 *Difference (Subtraction)*

The difference of set A with respect to set B is the collection of all elements in the universe that belong to A but do not belong to B, i.e., the difference set consists of all elements that belong to A but do not belong to B. It is denoted by $A|B$ or $A - B$ and is given by

$$A|B \text{ or } (A - B) = \{x | x \in A \text{ and } x \notin B\} = A - (A \cap B)$$

The vice versa of it also can be performed

$$B|A \text{ or } (B - A) = B - (B \cap A) = \{x | x \in B \text{ and } x \notin A\}$$

The above operations are shown in Figures 7-9(A) and (B).

7.2.2 Properties of Classical Sets

The important properties that define classical sets and show their similarity to fuzzy sets are as follows:

1. Commutativity

$$A \cup B = B \cup A; \quad A \cap B = B \cap A$$

2. Associativity

$$A \cup (B \cup C) = (A \cup B) \cup C; \quad A \cap (B \cap C) = (A \cap B) \cap C$$

3. Distributivity

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

4. Idempotency

$$A \cup A = A; \quad A \cap A = A$$

5. Transitivity

If $A \subseteq B \subseteq C$, then $A \subseteq C$

6. Identity

$$A \cup \phi = A, \quad A \cap \phi = \phi$$
$$A \cup X = X, \quad A \cap X = X$$

7. Involution (double negation)

$$\overline{\overline{A}} = A$$

8. Law of excluded middle

$$A \cup \overline{A} = X$$

9. Law of contradiction

$$A \cap \overline{A} = \phi$$

10. DeMorgan's law

$$\overline{A \cap B} = \overline{A} \cup \overline{B}; \quad \overline{A \cup B} = \overline{A} \cap \overline{B}$$

From the properties mentioned above, we can observe the duality existing by replacing ϕ, $\cup$, $\cap$ with X, $\cap$, $\cup$, respectively. It is important to know the law of excluded middle and the law of contradiction.

7.2.3 Function Mapping of Classical Sets

Mapping is a rule of correspondence between set-theoretic forms and function theoretic forms. A classical set is represented by its characteristic function $\chi(x)$, where x is the element in the universe.

Now consider X and Y as two different universes of discourse. If an element x contained in X corresponds to an element y contained in Y, it is called mapping from X to Y, i.e., $f: X \rightarrow Y$. On the basis of this mapping, the characteristic function is defined as

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$

where χ_A is the membership in set A for element x in the universe. The membership concept represents mapping from an element x in universe X to one of the two elements in universe Y (either to element 0 or 1). There exists a function-theoretic set called value set $V(A)$ for any set A defined on universe X, based on the mapping of characteristic function. The whole set is assigned a membership value 1, and the null set is assigned a membership value 0.

Let A and B be two sets on universe X. The function-theoretic forms of operations performed between these two sets are given as follows:

1. Union $(A \cup B)$

$$\chi_{A \cup B}(x) = \chi_A(x) \vee \chi_B(x) = \max\{\chi_A(x), \chi_B(x)\}$$

Here $\vee$ is the maximum operator.

2. Intersection $(A \cap B)$

$$\chi_{A \cap B}(x) = \chi_A(x) \wedge \chi_B(x) = \min\{\chi_A(x), \chi_B(x)\}$$

Here $\wedge$ is the minimum operator.

3. Complement $(\overline{A})$

$$\chi_{\overline{A}}(x) = 1 - \chi_A(x)$$

4. Containment

If $A \subseteq B$, then $\chi_A(x) \leq \chi_B(x)$

7.3 Fuzzy Sets

Fuzzy sets may be viewed as an extension and generalization of the basic concepts of crisp sets. An important property of fuzzy set is that it allows partial membership. A fuzzy set is a set having degrees of membership between 1 and 0. The membership in a fuzzy set need not be complete, i.e., member of one fuzzy set can also be member of other fuzzy sets in the same universe. Fuzzy sets can be analogous to the thinking of intelligent people. If a person has to be classified as friend or enemy, intelligent people will not resort to absolute classification as friend or enemy. Rather, they will classify the person somewhere between two extremes of friendship and enmity. Similarly, vagueness is introduced in fuzzy set by eliminating the sharp boundaries that divide members from nonmembers in the group. There is a gradual transition between full membership and nonmembership, not abrupt transition.

A fuzzy set $\underline{A}$ in the universe of discourse U can be defined as a set of ordered-pairs and it is given by

$$\underline{A} = \{(x, \mu_{\underline{A}}(x)) \mid x \in U\}$$

where $\mu_{\underline{A}}(x)$ is the degree of membership of x in $\underline{A}$ and it indicates the degree that x belongs to $\underline{A}$. The degree of membership $\mu_{\underline{A}}(x)$ assumes values in the range from 0 to 1, i.e., the membership is set to unit interval [0, 1] or $\mu_{\underline{A}}(x) \in [0, 1]$.

There are other ways of representation of fuzzy sets; all representations allow partial membership to be expressed. When the universe of discourse U is discrete and finite, fuzzy set $\underline{A}$ is given as follows:

$$\underline{A} = \left\{ \frac{\mu_{\underline{A}}(x_1)}{x_1} + \frac{\mu_{\underline{A}}(x_2)}{x_2} + \frac{\mu_{\underline{A}}(x_3)}{x_3} + \cdots \right\} = \left\{ \sum_{i=1}^{n} \frac{\mu_{\underline{A}}(x_i)}{x_i} \right\}$$

where "n" is a finite value. When the universe of discourse U is continuous and infinite, fuzzy set $\underline{A}$ is given by

$$\underline{A} = \left\{ \int \frac{\mu_{\underline{A}}(x)}{x} \right\}$$

In the above two representations of fuzzy sets for discrete and continuous universe, the horizontal bar is not a quotient but a delimiter. The numerator in each representation is the membership value in set $\underline{A}$ that is associated with the element of the universe present in the denominator. For discrete and finite universe of discourse U, the summation symbol in the representation of fuzzy set $\underline{A}$ does not denote algebraic summation but indicates the collection of each element. Thus the summation sign ("+") used is not the algebraic "add" but rather it is a discrete-function-theoretic union. Also, for continuous and infinite universe of discourse U, the integral sign in the representation of fuzzy set $\underline{A}$ is not an algebraic integral but is a continuous function-theoretic union for continuous variables.

A fuzzy set is universal fuzzy set if and only if the value of the membership function is 1 for all the members under consideration. Any fuzzy set $\underline{A}$ defined on a universe U is a subset of that universe. Two fuzzy sets $\underline{A}$ and $\underline{B}$ are said to be equal fuzzy sets if $\mu_{\underline{A}}(x) = \mu_{\underline{B}}(x)$ for all $x \in U$. A fuzzy set $\underline{A}$ is said to be empty fuzzy set if and only if the value of the membership function is 0 for all possible members considered. The universal fuzzy set can also be called whole fuzzy set.

The collection of all fuzzy sets and fuzzy subsets on universe U is called *fuzzy power set* $P(U)$. Since all the fuzzy sets can overlap, the cardinality of the fuzzy power set, $n_{P(U)}$ is infinite, i.e., $n_{P(U)} = \infty$.

On the basis of the above discussion we have

$$\underline{A} \subseteq U \Rightarrow \mu_{\underline{A}}(x) \leq \mu_U(x)$$

Also, for all $x \in U$

$$\mu_\phi(x) = 0; \quad \mu_U(x) = 1$$

7.3.1 Fuzzy Set Operations

The generalization of operations on classical sets to operations on fuzzy sets is not unique. The fuzzy set operations being discussed in this section are termed standard fuzzy set operations. These are the operations widely used in engineering applications. Let A and B be fuzzy sets in the universe of discourse U. For a given element x on the universe, the following function theoretic operations of union, intersection and complement are defined for fuzzy sets $\underline{A}$ and $\underline{B}$ on U.

7.3.1.1 Union

The union of fuzzy sets $\underline{A}$ and $\underline{B}$, denoted by $\underline{A} \cup \underline{B}$, is defined as

$$\mu_{\underline{A} \cup \underline{B}}(x) = \max[\mu_{\underline{A}}(x), \mu_{\underline{B}}(x)] = \mu_{\underline{A}}(x) \vee \mu_{\underline{B}}(x) \quad \text{for all } x \in U$$

where $\vee$ indicates max operation. The Venn diagram for union operation of fuzzy sets $\underline{A}$ and $\underline{B}$ is shown in Figure 7-10.

7.3.1.2 Intersection

The intersection of fuzzy sets $\underline{A}$ and $\underline{B}$, denoted by $\underline{A} \cap \underline{B}$, is defined by

$$\mu_{\underline{A} \cap \underline{B}}(x) = \min[\mu_{\underline{A}}(x), \mu_{\underline{B}}(x)] = \mu_{\underline{A}}(x) \wedge \mu_{\underline{B}}(x) \quad \text{for all } x \in U$$

where $\wedge$ indicates min operator. The Venn diagram for intersection operation of fuzzy sets $\underline{A}$ and $\underline{B}$ is shown in Figure 7-11.

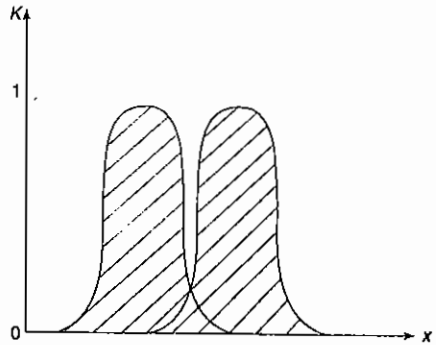


Figure 7-10 Union of fuzzy sets $\underline{A}$ and $\underline{B}$.

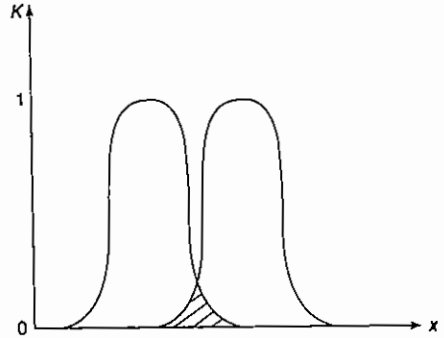


Figure 7-11 Intersection of fuzzy sets $\underline{A}$ and $\underline{B}$.

7.3.1.3 Complement

When $\mu_{\underline{A}}(x) \in [0, 1]$, the complement of $\underline{A}$, denoted as $\overline{\underline{A}}$ is defined by

$$\mu_{\overline{\underline{A}}}(x) = 1 - \mu_{\underline{A}}(x) \quad \text{for all } x \in U$$

The Venn diagram for complement operation of fuzzy set $\underline{A}$ is shown in Figure 7-12.

7.3.1.4 More Operations on Fuzzy Sets

1. *Algebraic sum:* The algebraic sum $(\underline{A} + \underline{B})$ of fuzzy sets, fuzzy sets $\underline{A}$ and $\underline{B}$ is defined as

$$\mu_{\underline{A}+\underline{B}}(x) = \mu_{\underline{A}}(x) + \mu_{\underline{B}}(x) - \mu_{\underline{A}}(x) \cdot \mu_{\underline{B}}(x)$$

2. *Algebraic product:* The algebraic product $(\underline{A} \cdot \underline{B})$ of two fuzzy sets $\underline{A}$ and $\underline{B}$ is defined as

$$\mu_{\underline{A}\cdot\underline{B}}(x) = \mu_{\underline{A}}(x) \cdot \mu_{\underline{B}}(x)$$

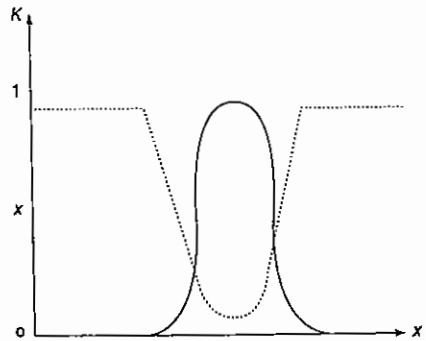


Figure 7-12 Complement of fuzzy set $\underline{A}$.

3. *Bounded sum:* The bounded sum $(\underline{A} \oplus \underline{B})$ of two fuzzy sets $\underline{A}$ and $\underline{B}$ is defined as

$$\mu_{\underline{A}\oplus\underline{B}}(x) = \min\{1, \mu_{\underline{A}}(x) + \mu_{\underline{B}}(x)\}$$

4. *Bounded difference:* The bounded difference $(\underline{A} \ominus \underline{B})$ of two fuzzy sets $\underline{A}$ and $\underline{B}$ is defined as

$$\mu_{\underline{A}\ominus\underline{B}}(x) = \max\{0, \mu_{\underline{A}}(x) - \mu_{\underline{B}}(x)\}$$

7.3.2 Properties of Fuzzy Sets

Fuzzy sets follow the same properties as crisp sets except for the law of excluded middle and law of contradiction. That is, for fuzzy set $\underline{A}$

law of excluded middle $\rightarrow \underline{A} \cup \overline{\underline{A}} \neq U$; $\underline{A} \cap \overline{\underline{A}} \neq \phi \leftarrow$ law of contradiction

Frequently used properties of fuzzy sets are given as follows:

1. Commutativity

$$\underline{A} \cup \underline{B} = \underline{B} \cup \underline{A}; \quad \underline{A} \cap \underline{B} = \underline{B} \cap \underline{A}$$

2. Associativity

$$\underline{A} \cup (\underline{B} \cup \underline{C}) = (\underline{A} \cup \underline{B}) \cup \underline{C}$$
$$\underline{A} \cap (\underline{B} \cap \underline{C}) = (\underline{A} \cap \underline{B}) \cap \underline{C}$$

3. Distributivity

$$\underline{A} \cup (\underline{B} \cap \underline{C}) = (\underline{A} \cup \underline{B}) \cap (\underline{A} \cup \underline{C})$$
$$\underline{A} \cap (\underline{B} \cup \underline{C}) = (\underline{A} \cap \underline{B}) \cup (\underline{A} \cap \underline{C})$$

4. Idempotency

$$\underline{A} \cup \underline{A} = \underline{A}; \quad \underline{A} \cap \underline{A} = \underline{A}$$

5. Identity

$$\underline{A} \cup \phi = \underline{A} \quad \text{and} \quad \underline{A} \cup U = U \text{(universal set)}$$
$$\underline{A} \cap \phi = \phi \quad \text{and} \quad \underline{A} \cap U = \underline{A}$$

6. Involution (double negation)

$$\overline{\overline{\underline{A}}} = \underline{A}$$

7. Transitivity

$$\text{If } \underline{A} \subseteq \underline{B} \subseteq \underline{C}, \text{ then } \underline{A} \subseteq \underline{C}$$

8. De Morgan's law

$$\overline{\underline{A} \cup \underline{B}} = \overline{\underline{A}} \cap \overline{\underline{B}}; \overline{\underline{A} \cap \underline{B}} = \overline{\underline{A}} \cup \overline{\underline{B}}$$

7.4 Summary

In this chapter, we have discussed the basic definitions, properties and operations on classical sets and fuzzy sets. Fuzzy sets are the tools that convert the concept of fuzzy logic into algorithms. Since fuzzy sets allow partial membership, they provide computer with such algorithms that extend binary logic and enable it to take human-like decisions. In other words, fuzzy sets can be thought of as a media through which the human thinking is transferred to a computer. One difference between fuzzy sets and classical sets is that the former do not follow the law of excluded middle and law of contradiction. Hence, if we want to choose fuzzy intersection and union operations which satisfy these laws, then the operations will not satisfy distributivity and idempotency. Except the difference of set membership being an infinite valued quantity instead of a binary valued quantity, fuzzy sets are treated in the same mathematical form as classical sets.

7.5 Solved Problems

1. Find the power set and cardinality of the given set $X = \{2, 4, 6\}$. Also find cardinality of power set.

Solution: Since set X contains three elements, so its cardinal number is

$$n_X = 3$$

The power set of X is given by

$$P(X) = \{\phi, \{2\}, \{4\}, \{6\}, \{2,4\}, \{4,6\}, \{2,6\}, \{2,4,6\}\}$$

The cardinality of power set $P(X)$, denoted by $n_{P(X)}$, is found as

$$n_{P(X)} = 2^{n_X} = 2^3 = 8$$

2. Consider two given fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{1}{2} + \frac{0.3}{4} + \frac{0.5}{6} + \frac{0.2}{8}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{0.5}{2} + \frac{0.4}{4} + \frac{0.1}{6} + \frac{1}{8}\right\}$$

Perform union, intersection, difference and complement over fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$.

Solution: For the given fuzzy sets we have the following

(a) Union

$$\underset{\sim}{A} \cup \underset{\sim}{B} = \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\} = \left\{\frac{1}{2} + \frac{0.4}{4} + \frac{0.5}{6} + \frac{1}{8}\right\}$$

(b) Intersection

$$\underset{\sim}{A} \cap \underset{\sim}{B} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\} = \left\{\frac{0.5}{2} + \frac{0.3}{4} + \frac{0.1}{6} + \frac{0.2}{8}\right\}$$

(c) Complement

$$\underset{\sim}{A} = 1 - \mu_{\underset{\sim}{A}}(x) = \left\{\frac{0}{2} + \frac{0.7}{4} + \frac{0.5}{6} + \frac{0.8}{8}\right\}$$

$$\underset{\sim}{B} = 1 - \mu_{\underset{\sim}{B}}(x) = \left\{\frac{0.5}{2} + \frac{0.6}{4} + \frac{0.9}{6} + \frac{0}{8}\right\}$$

(d) Difference

$$\underset{\sim}{A} | \underset{\sim}{B} = \underset{\sim}{A} \cap \overline{\underset{\sim}{B}} = \left\{\frac{0.5}{2} + \frac{0.3}{4} + \frac{0.5}{6} + \frac{0}{8}\right\}$$

$$\underset{\sim}{B} | \underset{\sim}{A} = \underset{\sim}{B} \cap \overline{\underset{\sim}{A}} = \left\{\frac{0}{2} + \frac{0.4}{4} + \frac{0.1}{6} + \frac{0.8}{8}\right\}$$

3. Given the two fuzzy sets

$$\underset{\sim}{B}_1 = \left\{\frac{1}{1.0} + \frac{0.75}{1.5} + \frac{0.3}{2.0} + \frac{0.15}{2.5} + \frac{0}{3.0}\right\}$$

$$\underset{\sim}{B}_2 = \left\{\frac{1}{1.0} + \frac{0.6}{1.5} + \frac{0.2}{2.0} + \frac{0.1}{2.5} + \frac{0}{3.0}\right\}$$

find the following:

(a) $\underset{\sim}{B}_1 \cup \underset{\sim}{B}_2$; (b) $\underset{\sim}{B}_1 \cap \underset{\sim}{B}_2$; (c) $\overline{\underset{\sim}{B}_1}$;
(d) $\overline{\underset{\sim}{B}_2}$; (e) $\underset{\sim}{B}_1 | \underset{\sim}{B}_2$; (f) $\overline{\underset{\sim}{B}_1 \cup \underset{\sim}{B}_2}$;
(g) $\overline{\underset{\sim}{B}_1 \cap \underset{\sim}{B}_2}$; (h) $\underset{\sim}{B}_1 \cap \overline{\underset{\sim}{B}_1}$; (i) $\underset{\sim}{B}_1 \cup \overline{\underset{\sim}{B}_1}$;
(j) $\underset{\sim}{B}_2 \cap \overline{\underset{\sim}{B}_2}$; (k) $\underset{\sim}{B}_2 \cup \overline{\underset{\sim}{B}_2}$

Solution: For the given fuzzy sets, we have the following:

(a) $\underset{\sim}{B}_1 \cup \underset{\sim}{B}_2 = \left\{\frac{1}{1.0} + \frac{0.75}{1.5} + \frac{0.3}{2.0} + \frac{0.15}{2.5} + \frac{0}{3.0}\right\}$

(b) $\underset{\sim}{B}_1 \cap \underset{\sim}{B}_2 = \left\{\frac{1}{1.0} + \frac{0.6}{1.5} + \frac{0.2}{2.0} + \frac{0.1}{2.5} + \frac{0}{3.0}\right\}$

(c) $\overline{\underset{\sim}{B}_1} = \left\{\frac{0}{1.0} + \frac{0.25}{1.5} + \frac{0.7}{2.0} + \frac{0.85}{2.5} + \frac{1}{3.0}\right\}$

(d) $\overline{\underset{\sim}{B}_2} = \left\{\frac{0}{1.0} + \frac{0.4}{1.5} + \frac{0.8}{2.0} + \frac{0.9}{2.5} + \frac{1}{3.0}\right\}$

(e) $\underset{\sim}{B}_1 | \underset{\sim}{B}_2 = \underset{\sim}{B}_1 \cap \overline{\underset{\sim}{B}_2} = \left\{\frac{0}{1.0} + \frac{0.4}{1.5} + \frac{0.3}{2.0} + \frac{0.15}{2.5} + \frac{0}{3.0}\right\}$

(f) $\overline{\underset{\sim}{B}_1 \cup \underset{\sim}{B}_2} = \left\{\frac{0}{1.0} + \frac{0.25}{1.5} + \frac{0.7}{2.0} + \frac{0.85}{2.5} + \frac{1}{3.0}\right\}$

(g) $\overline{\underset{\sim}{B}_1 \cap \underset{\sim}{B}_2} = \left\{\frac{0}{1.0} + \frac{0.4}{1.5} + \frac{0.8}{2.0} + \frac{0.9}{2.5} + \frac{1}{3.0}\right\}$

(h) $\underset{\sim}{B}_1 \cap \overline{\underset{\sim}{B}_1} = \left\{\frac{0}{1.0} + \frac{0.25}{1.5} + \frac{0.3}{2.0} + \frac{0.15}{2.5} + \frac{0}{3.0}\right\}$

(i) $\underset{\sim}{B}_1 \cup \overline{\underset{\sim}{B}_1} = \left\{\frac{1}{1.0} + \frac{0.75}{1.5} + \frac{0.7}{2.0} + \frac{0.85}{2.5} + \frac{1}{3.0}\right\}$

(j) $\underset{\sim}{B}_2 \cap \overline{\underset{\sim}{B}_2} = \left\{\frac{0}{1.0} + \frac{0.4}{1.5} + \frac{0.2}{2.0} + \frac{0.1}{2.5} + \frac{0}{3.0}\right\}$

(k) $\underset{\sim}{B}_2 \cup \overline{\underset{\sim}{B}_2} = \left\{\frac{1}{1.0} + \frac{0.6}{1.5} + \frac{0.8}{2.0} + \frac{0.9}{2.5} + \frac{1}{3.0}\right\}$

4. It is necessary to compare two sensors based upon their detection levels and gain settings. The table of gain settings and sensor detection levels with a standard item being monitored providing typical membership values to represent the detection levels for each sensor is given in Table 1.

Table 1

Gain setting	Detection level of sensor 1	Detection level of sensor 2
0	0	0
10	0.2	0.35
20	0.35	0.25
30	0.65	0.8
40	0.85	0.95
50	1	1

Now given the universe of discourse $X = \{0, 10, 20, 30, 40, 50\}$ and the membership functions for the two sensors in discrete form as

$$\underset{\sim}{D}_1 = \left\{\frac{0}{0} + \frac{0.2}{10} + \frac{0.35}{20} + \frac{0.65}{30} + \frac{0.85}{40} + \frac{1}{50}\right\}$$

$$\underset{\sim}{D}_2 = \left\{\frac{0}{0} + \frac{0.35}{10} + \frac{0.25}{20} + \frac{0.8}{30} + \frac{0.95}{40} + \frac{1}{50}\right\}$$

find the following membership functions:

(a) $\mu_{\underset{\sim}{D}_1 \cup \underset{\sim}{D}_2}(x)$; (b) $\mu_{\underset{\sim}{D}_1 \cap \underset{\sim}{D}_2}(x)$; (c) $\mu_{\overline{\underset{\sim}{D}_1}}(x)$;
(d) $\mu_{\overline{\underset{\sim}{D}_2}}(x)$; (e) $\mu_{\underset{\sim}{D}_1 \cup \overline{\underset{\sim}{D}_1}}(x)$; (f) $\mu_{\underset{\sim}{D}_1 \cap \overline{\underset{\sim}{D}_1}}(x)$;
(g) $\mu_{\underset{\sim}{D}_2 \cup \overline{\underset{\sim}{D}_2}}$; (h) $\mu_{\underset{\sim}{D}_2 \cap \overline{\underset{\sim}{D}_2}}(x)$; (i) $\mu_{\underset{\sim}{D}_1 | \underset{\sim}{D}_2}(x)$;
(j) $\mu_{\underset{\sim}{D}_2 | \underset{\sim}{D}_1}(x)$

Solution: For the given fuzzy sets we have

(a) $\mu_{\underset{\sim}{D}_1 \cup \underset{\sim}{D}_2}(x)$
$= \max\{\mu_{\underset{\sim}{D}_1}(x), \mu_{\underset{\sim}{D}_2}(x)\}$
$= \left\{\frac{0}{0} + \frac{0.35}{10} + \frac{0.35}{20} + \frac{0.8}{30} + \frac{0.95}{40} + \frac{1}{50}\right\}$

(b) $\mu_{\underset{\sim}{D}_1 \cap \underset{\sim}{D}_2}(x)$
$= \min\{\mu_{\underset{\sim}{D}_1}(x), \mu_{\underset{\sim}{D}_2}(x)\}$
$= \left\{\frac{0}{0} + \frac{0.2}{10} + \frac{0.25}{20} + \frac{0.65}{30} + \frac{0.85}{40} + \frac{1}{50}\right\}$

(c) $\mu_{\overline{\underset{\sim}{D}_1}}(x)$
$= 1 - \mu_{\underset{\sim}{D}_1}(x)$
$= \left\{\frac{1}{0} + \frac{0.8}{10} + \frac{0.65}{20} + \frac{0.35}{30} + \frac{0.15}{40} + \frac{0}{50}\right\}$

(d) $\mu_{\overline{D_2}}(x)$

$= 1-\mu_{D_2}(x)$

$= \left\{\frac{1}{0}+\frac{0.65}{10}+\frac{0.75}{20}+\frac{0.2}{30}+\frac{0.05}{40}+\frac{0}{50}\right\}$

(e) $\mu_{D_1\cup\overline{D_1}}(x)$

$= \max\{\mu_{D_1}(x), \mu_{\overline{D_1}}(x)\}$

$= \left\{\frac{1}{0}+\frac{0.8}{10}+\frac{0.65}{20}+\frac{0.65}{30}+\frac{0.85}{40}+\frac{1}{50}\right\}$

(f) $\mu_{D_1\cap\overline{D_1}}(x)$

$= \min\{\mu_{D_1}(x), \mu_{\overline{D_1}}(x)\}$

$= \left\{\frac{0}{0}+\frac{0.2}{10}+\frac{0.35}{20}+\frac{0.35}{30}+\frac{0.15}{40}+\frac{0}{50}\right\}$

(g) $\mu_{D_2\cup\overline{D_2}}(x)$

$= \max\{\mu_{D_2}(x), \mu_{\overline{D_2}}(x)\}$

$= \left\{\frac{1}{0}+\frac{0.65}{10}+\frac{0.75}{20}+\frac{0.8}{30}+\frac{0.95}{40}+\frac{1}{50}\right\}$

(h) $\mu_{D_2\cap\overline{D_2}}(x)$

$= \min\{\mu_{D_2}(x), \mu_{\overline{D_2}}(x)\}$

$= \left\{\frac{0}{0}+\frac{0.35}{10}+\frac{0.25}{20}+\frac{0.2}{30}+\frac{0.05}{40}+\frac{0}{50}\right\}$

(i) $\mu_{D_1|D_2}(x)$

$= \mu_{D_1\cap\overline{D_2}}(x) = \min\{\mu_{D_1}(x), \mu_{\overline{D_2}}(x)\}$

$= \left\{\frac{0}{0}+\frac{0.2}{10}+\frac{0.35}{20}+\frac{0.2}{30}+\frac{0.05}{40}+\frac{0}{50}\right\}$

(j) $\mu_{D_2|D_1}(x)$

$= \mu_{D_2\cap\overline{D_1}}(x) = \min\{\mu_{D_2}(x), \mu_{\overline{D_1}}(x)\}$

$= \left\{\frac{0}{0}+\frac{0.35}{10}+\frac{0.25}{20}+\frac{0.35}{30}+\frac{0.15}{40}+\frac{0}{50}\right\}$

5. Design a computer software to perform image processing to locate objects within a scene. The two fuzzy sets representing a plane and a train image are:

$$\text{Plane} = \left\{\frac{0.2}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.3}{\text{boat}}+\frac{0.8}{\text{plane}}+\frac{0.1}{\text{house}}\right\}$$

$$\text{Train} = \left\{\frac{1}{\text{train}}+\frac{0.2}{\text{bike}}+\frac{0.4}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.2}{\text{house}}\right\}$$

Find the following:

(a) Plane ∪ Train; (b) Plane ∩ Train;

(c) $\overline{\text{Plane}}$; (d) $\overline{\text{Train}}$;

(e) Plane|Train; (f) $\overline{\text{Plane} \cup \text{Train}}$;

(g) $\overline{\text{Plane} \cap \text{Train}}$; (h) Plane ∪ $\overline{\text{Plane}}$;

(i) Plane ∩ $\overline{\text{Plane}}$; (j) Train ∪ $\overline{\text{Train}}$;

(k) Train ∪ $\overline{\text{Train}}$

Solution: For the given fuzzy sets we have the following:

(a) Plane ∪ Train

$= \max\{\mu_{\text{Plane}}(x), \mu_{\text{Train}}(x)\}$

$= \left\{\frac{1.0}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.4}{\text{boat}}+\frac{0.8}{\text{plane}}+\frac{0.2}{\text{house}}\right\}$

(b) Plane ∩ Train

$= \min\{\mu_{\text{Plane}}(x), \mu_{\text{Train}}(x)\}$

$= \left\{\frac{0.2}{\text{train}}+\frac{0.2}{\text{bike}}+\frac{0.3}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.1}{\text{house}}\right\}$

(c) $\overline{\text{Plane}} = 1-\mu_{\text{Plane}}(x)$

$= \left\{\frac{0.8}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.7}{\text{boat}}+\frac{0.2}{\text{plane}}+\frac{0.9}{\text{house}}\right\}$

(d) $\overline{\text{Train}} = 1-\mu_{\text{Train}}(x)$

$= \left\{\frac{0}{\text{train}}+\frac{0.8}{\text{bike}}+\frac{0.6}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.8}{\text{house}}\right\}$

(e) Plane|Train

$= \text{Plane} \cap \overline{\text{Train}}$

$= \min\{\mu_{\text{Plane}}(x), \mu_{\overline{\text{Train}}}(x)\}$

$= \left\{\frac{0}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.3}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.1}{\text{house}}\right\}$

(f) $\overline{\text{Plane} \cup \text{Train}}$

$= 1 - \max\{\mu_{\text{Plane}}(x), \mu_{\text{Train}}(x)\}$

$= \left\{\frac{0}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.6}{\text{boat}}+\frac{0.2}{\text{plane}}+\frac{0.8}{\text{house}}\right\}$

(g) $\overline{\text{Plane} \cap \text{Train}}$

$= 1 - \min\{\mu_{\text{Plane}}(x), \mu_{\text{Train}}(x)\}$

$= \left\{\frac{0.8}{\text{train}}+\frac{0.8}{\text{bike}}+\frac{0.7}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.9}{\text{house}}\right\}$

(h) Plane ∪ $\overline{\text{Plane}}$

$= \max\{\mu_{\text{Plane}}(x), \mu_{\overline{\text{Plane}}}(x)\}$

$= \left\{\frac{0.8}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.7}{\text{boat}}+\frac{0.8}{\text{plane}}+\frac{0.9}{\text{house}}\right\}$

(i) Plane ∩ $\overline{\text{Plane}}$

$= \min\{\mu_{\text{Plane}}(x), \mu_{\overline{\text{Plane}}}(x)\}$

$= \left\{\frac{0.2}{\text{train}}+\frac{0.5}{\text{bike}}+\frac{0.3}{\text{boat}}+\frac{0.2}{\text{plane}}+\frac{0.1}{\text{house}}\right\}$

(j) Train ∪ $\overline{\text{Train}}$

$= \max\{\mu_{\text{Train}}(x), \mu_{\overline{\text{Train}}}(x)\}$

$= \left\{\frac{1.0}{\text{train}}+\frac{0.8}{\text{bike}}+\frac{0.6}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.8}{\text{house}}\right\}$

(k) Train ∩ $\overline{\text{Train}}$

$= \min\{\mu_{\text{Train}}(x), \mu_{\overline{\text{Train}}}(x)\}$

$= \left\{\frac{0}{\text{train}}+\frac{0.2}{\text{bike}}+\frac{0.4}{\text{boat}}+\frac{0.5}{\text{plane}}+\frac{0.2}{\text{house}}\right\}$.

6. For aircraft simulator data the determination of certain changes in its operating conditions is made on the basis of hard break points in the mach region. We define two fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$ representing the condition of "near" a mach number of 0.65 and "in the region" of a mach number of 0.65, respectively, as follows

$\underset{\sim}{A}$ = near mach 0.65

$= \left\{\frac{0}{0.64}+\frac{0.75}{0.645}+\frac{1}{0.65}+\frac{0.5}{0.655}+\frac{0}{0.66}\right\}$

$\underset{\sim}{B}$ = in the region of mach 0.65

$= \left\{\frac{0}{0.64}+\frac{0.25}{0.645}+\frac{0.75}{0.65}+\frac{1}{0.655}+\frac{0.5}{0.66}\right\}$

For these two sets find the following:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B}$; (b) $\underset{\sim}{A} \cap \underset{\sim}{B}$; (c) $\overline{\underset{\sim}{A}}$;

(d) $\overline{\underset{\sim}{B}}$; (e) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$; (f) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}}$

Solution: For the two given fuzzy sets we have the following:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B}$

$= \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$= \left\{\frac{0}{0.64}+\frac{0.75}{0.645}+\frac{1}{0.65}+\frac{1}{0.655}+\frac{0.5}{0.66}\right\}$

(b) $\underset{\sim}{A} \cap \underset{\sim}{B}$

$= \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$= \left\{\frac{0}{0.64}+\frac{0.25}{0.645}+\frac{0.75}{0.65}+\frac{0.5}{0.655}+\frac{0}{0.66}\right\}$

(c) $\overline{\underset{\sim}{A}} = 1-\mu_{\underset{\sim}{A}}(x)$

$= \left\{\frac{1}{0.64}+\frac{0.25}{0.645}+\frac{0}{0.65}+\frac{0.5}{0.655}+\frac{1}{0.66}\right\}$

(d) $\overline{\underset{\sim}{B}} = 1-\mu_{\underset{\sim}{B}}(x)$

$= \left\{\frac{1}{0.64}+\frac{0.75}{0.645}+\frac{0.25}{0.65}+\frac{0}{0.655}+\frac{0.5}{0.66}\right\}$

(e) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$

$= 1 - \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$= \left\{\frac{1}{0.64}+\frac{0.25}{0.645}+\frac{0}{0.65}+\frac{0}{0.655}+\frac{0.5}{0.66}\right\}$

(f) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}}$

$= 1 - \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$= \left\{\frac{1}{0.64}+\frac{0.75}{0.645}+\frac{0.25}{0.65}+\frac{0.5}{0.655}+\frac{1}{0.66}\right\}$

7. For the two given fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{0.1}{0}+\frac{0.2}{1}+\frac{0.4}{2}+\frac{0.6}{3}+\frac{1}{4}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{1}{0}+\frac{0.5}{1}+\frac{0.7}{2}+\frac{0.3}{3}+\frac{0}{4}\right\}$$

find the following:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B}$; (b) $\underset{\sim}{A} \cap \underset{\sim}{B}$; (c) $\overline{\underset{\sim}{A}}$;
(d) $\overline{\underset{\sim}{B}}$; (e) $\underset{\sim}{A} \cup \overline{\underset{\sim}{A}}$; (f) $\underset{\sim}{A} \cap \overline{\underset{\sim}{A}}$;
(g) $\underset{\sim}{B} \cup \overline{\underset{\sim}{B}}$; (h) $\underset{\sim}{B} \cap \overline{\underset{\sim}{B}}$; (i) $\underset{\sim}{A} \cap \overline{\underset{\sim}{B}}$;
(j) $\underset{\sim}{A} \cup \overline{\underset{\sim}{B}}$; (k) $\underset{\sim}{B} \cap \overline{\underset{\sim}{A}}$; (l) $\underset{\sim}{B} \cup \overline{\underset{\sim}{A}}$;
(m) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$; (n) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}}$

Solution: For the given sets we have:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B} = \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{1}{0} + \frac{0.5}{1} + \frac{0.7}{2} + \frac{0.6}{3} + \frac{1}{4}\right\}$$

(b) $\underset{\sim}{A} \cap \underset{\sim}{B} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0.1}{0} + \frac{0.2}{1} + \frac{0.4}{2} + \frac{0.3}{3} + \frac{0}{4}\right\}$$

(c) $\overline{\underset{\sim}{A}} = 1 - \mu_{\underset{\sim}{A}}(x)$

$$= \left\{\frac{0.9}{0} + \frac{0.8}{1} + \frac{0.6}{2} + \frac{0.4}{3} + \frac{0}{4}\right\}$$

(d) $\overline{\underset{\sim}{B}} = 1 - \mu_{\underset{\sim}{B}}(x)$

$$= \left\{\frac{0}{0} + \frac{0.5}{1} + \frac{0.3}{2} + \frac{0.7}{3} + \frac{1}{4}\right\}$$

(e) $\underset{\sim}{A} \cup \overline{\underset{\sim}{A}} = \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)\}$

$$= \left\{\frac{0.9}{0} + \frac{0.8}{1} + \frac{0.6}{2} + \frac{0.6}{3} + \frac{1}{4}\right\}$$

(f) $\underset{\sim}{A} \cap \overline{\underset{\sim}{A}} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)\}$

$$= \left\{\frac{0.1}{0} + \frac{0.2}{1} + \frac{0.4}{2} + \frac{0.4}{3} + \frac{0}{4}\right\}$$

(g) $\underset{\sim}{B} \cup \overline{\underset{\sim}{B}} = \max\{\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{1}{0} + \frac{0.5}{1} + \frac{0.7}{2} + \frac{0.7}{3} + \frac{1}{4}\right\}$$

(h) $\underset{\sim}{B} \cap \overline{\underset{\sim}{B}} = \min\{\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0}{0} + \frac{0.5}{1} + \frac{0.3}{2} + \frac{0.3}{3} + \frac{0}{4}\right\}$$

(i) $\underset{\sim}{A} \cap \overline{\underset{\sim}{B}} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0}{0} + \frac{0.2}{1} + \frac{0.3}{2} + \frac{0.6}{3} + \frac{1}{4}\right\}$$

(j) $\underset{\sim}{A} \cup \overline{\underset{\sim}{B}} = \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0.1}{0} + \frac{0.5}{1} + \frac{0.4}{2} + \frac{0.7}{3} + \frac{1}{4}\right\}$$

(k) $\underset{\sim}{B} \cap \overline{\underset{\sim}{A}} = \min\{\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)\}$

$$= \left\{\frac{0.9}{0} + \frac{0.5}{1} + \frac{0.6}{2} + \frac{0.3}{3} + \frac{0}{4}\right\}$$

(l) $\underset{\sim}{B} \cup \overline{\underset{\sim}{A}} = \max\{\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)\}$

$$= \left\{\frac{1}{0} + \frac{0.8}{1} + \frac{0.7}{2} + \frac{0.4}{3} + \frac{0}{4}\right\}$$

(m) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}} = 1 - \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0}{0} + \frac{0.5}{1} + \frac{0.3}{2} + \frac{0.4}{3} + \frac{0}{4}\right\}$$

(n) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}} = \min\{\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0}{0} + \frac{0.5}{1} + \frac{0.3}{2} + \frac{0.4}{3} + \frac{0}{4}\right\}$$

8. Let U be the universe of military aircraft of interest' as defined below:

$$U = \{a10,\ b52,\ c130,\ f2,\ f9\}$$

Let $\underset{\sim}{A}$ be the fuzzy set of bomber class aircraft:

$$\underset{\sim}{A} = \left\{\frac{0.3}{a10} + \frac{0.4}{b52} + \frac{0.2}{c130} + \frac{0.1}{f2} + \frac{1}{f9}\right\}$$

Let $\underset{\sim}{B}$ be the fuzzy set of fighter class aircraft:

$$\underset{\sim}{B} = \left\{\frac{0.1}{a10} + \frac{0.2}{b52} + \frac{0.8}{c130} + \frac{0.7}{f2} + \frac{0}{f9}\right\}$$

Find the following:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B}$; (b) $\underset{\sim}{A} \cap \underset{\sim}{B}$; (c) $\overline{\underset{\sim}{A}}$; (d) $\overline{\underset{\sim}{B}}$;
(e) $\underset{\sim}{A} | \underset{\sim}{B}$; (f) $\underset{\sim}{B} | \underset{\sim}{A}$; (g) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$;
(h) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}}$; (i) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}}$; (j) $\overline{\underset{\sim}{B}} \cup \underset{\sim}{A}$

Solution: We have

(a) $\underset{\sim}{A} \cup \underset{\sim}{B} = \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0.3}{a10} + \frac{0.4}{b52} + \frac{0.8}{c130} + \frac{0.7}{f2} + \frac{1}{f9}\right\}$$

(b) $\underset{\sim}{A} \cap \underset{\sim}{B} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0.1}{a10} + \frac{0.2}{b52} + \frac{0.2}{c130} + \frac{0.1}{f2} + \frac{0}{f9}\right\}$$

(c) $\overline{\underset{\sim}{A}} = 1 - \mu_{\underset{\sim}{A}}(x)$

$$= \left\{\frac{0.7}{a10} + \frac{0.6}{b52} + \frac{0.8}{c130} + \frac{0.9}{f2} + \frac{0}{f9}\right\}$$

(d) $\overline{\underset{\sim}{B}} = 1 - \mu_{\underset{\sim}{B}}(x)$

$$= \left\{\frac{0.9}{a10} + \frac{0.8}{b52} + \frac{0.2}{c130} + \frac{0.3}{f2} + \frac{1}{f9}\right\}$$

(e) $\underset{\sim}{A} | \underset{\sim}{B} = \underset{\sim}{A} \cap \overline{\underset{\sim}{B}} = \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0.3}{a10} + \frac{0.4}{b52} + \frac{0.2}{c130} + \frac{0.1}{f2} + \frac{1}{f9}\right\}$$

(f) $\underset{\sim}{B} | \underset{\sim}{A} = \underset{\sim}{B} \cap \overline{\underset{\sim}{A}} = \min\{\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)\}$

$$= \left\{\frac{0.1}{a10} + \frac{0.2}{b52} + \frac{0.8}{c130} + \frac{0.7}{f2} + \frac{0}{f9}\right\}$$

(g) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}} = 1 - \max\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0.7}{a10} + \frac{0.6}{b52} + \frac{0.2}{c130} + \frac{0.3}{f2} + \frac{0}{f9}\right\}$$

(h) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}} = 1 - \min\{\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)\}$

$$= \left\{\frac{0.9}{a10} + \frac{0.8}{b52} + \frac{0.8}{c130} + \frac{0.9}{f2} + \frac{1}{f9}\right\}$$

(i) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}} = \max\{\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)\}$

$$= \left\{\frac{0.9}{a10} + \frac{0.8}{b52} + \frac{0.8}{c130} + \frac{0.9}{f2} + \frac{1}{f9}\right\}$$

(j) $\overline{\underset{\sim}{B}} \cup \underset{\sim}{A} = \max\{\mu_{\overline{\underset{\sim}{B}}}(x), \mu_{\underset{\sim}{A}}(x)\}$

$$= \left\{\frac{0.9}{a10} + \frac{0.8}{b52} + \frac{0.2}{c130} + \frac{0.3}{f2} + \frac{1}{f9}\right\}$$

9. Consider two fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{0.2}{1} + \frac{0.3}{2} + \frac{0.4}{3} + \frac{0.5}{4}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{0.1}{1} + \frac{0.2}{2} + \frac{0.2}{3} + \frac{1}{4}\right\}$$

Find the algebraic sum, algebraic product, bounded sum and bounded difference of the given fuzzy sets.

Solution: We have

(a) Algebraic sum

$$\mu_{\underset{\sim}{A}+\underset{\sim}{B}}(x) = [\mu_{\underset{\sim}{A}}(x) + \mu_{\underset{\sim}{B}}(x)] - [\mu_{\underset{\sim}{A}}(x) \cdot \mu_{\underset{\sim}{B}}(x)]$$

$$= \left\{\frac{0.3}{1} + \frac{0.5}{2} + \frac{0.6}{3} + \frac{0.5}{4}\right\} - \left\{\frac{0.02}{1} + \frac{0.06}{2} + \frac{0.08}{3} + \frac{0.5}{4}\right\}$$

$$= \left\{\frac{0.28}{1} + \frac{0.44}{2} + \frac{0.52}{3} + \frac{0}{4}\right\}$$

(b) Algebraic product

$$\mu_{\underset{\sim}{A} \cdot \underset{\sim}{B}}(x) = \mu_{\underset{\sim}{A}}(x) \cdot \mu_{\underset{\sim}{B}}(x)$$

$$= \left\{\frac{0.02}{1} + \frac{0.06}{2} + \frac{0.08}{3} + \frac{0.5}{4}\right\}$$

(c) Bounded sum

$$\mu_{\underset{\sim}{A} \oplus \underset{\sim}{B}}(x) = \min[1, \mu_{\underset{\sim}{A}}(x) + \mu_{\underset{\sim}{B}}(x)]$$

$$= \min\left\{1, \left\{\frac{0.3}{1} + \frac{0.5}{2} + \frac{0.6}{3} + \frac{0.5}{4}\right\}\right\}$$

$$= \left\{\frac{0.3}{1} + \frac{0.5}{2} + \frac{0.6}{3} + \frac{0.5}{4}\right\}$$

(d) Bounded difference

$$\mu_{\underset{\sim}{A} \ominus \underset{\sim}{B}}(x) = \max[0, \mu_{\underset{\sim}{A}}(x) - \mu_{\underset{\sim}{B}}(x)]$$

$$= \max\left\{0, \left\{\frac{0.1}{1} + \frac{0.1}{2} + \frac{0.2}{3} + \frac{0.5}{4}\right\}\right\}$$

$$= \left\{\frac{0.1}{1} + \frac{0.1}{2} + \frac{0.2}{3} + \frac{0.5}{4}\right\}$$

10. The discretized membership functions for a transistor and a resistor are given below:

$$\mu_{\underline{T}} = \left\{\frac{0}{0} + \frac{0.2}{1} + \frac{0.7}{2} + \frac{0.8}{3} + \frac{0.9}{4} + \frac{1}{5}\right\}$$

$$\mu_{\underline{R}} = \left\{\frac{0}{0} + \frac{0.1}{1} + \frac{0.3}{2} + \frac{0.2}{3} + \frac{0.4}{4} + \frac{0.5}{5}\right\}$$

Find the following: (a) Algebraic sum; (b) algebraic product; (c) bounded sum; (d) bounded difference.

Solution: We have

(a) Algebraic sum

$$\mu_{\underline{T}+\underline{R}}(x) = [\mu_{\underline{T}}(x)+\mu_{\underline{R}}(x)] - [\mu_{\underline{T}}(x)\cdot\mu_{\underline{R}}(x)]$$

$$= \left\{\frac{0}{0} + \frac{0.3}{1} + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.3}{4} + \frac{1.5}{5}\right\} - \left\{\frac{0}{0} + \frac{0.02}{1} + \frac{0.21}{2} + \frac{0.16}{3} + \frac{0.36}{4} + \frac{0.5}{5}\right\}$$

$$= \left\{\frac{0}{0} + \frac{0.28}{1} + \frac{0.79}{2} + \frac{0.84}{3} + \frac{0.94}{4} + \frac{1}{5}\right\}$$

(b) Algebraic product

$$\mu_{\underline{T}\cdot\underline{R}}(x) = \mu_{\underline{T}}(x)\cdot\mu_{\underline{R}}(x) = \left\{\frac{0}{0} + \frac{0.02}{1} + \frac{0.21}{2} + \frac{0.16}{3} + \frac{0.36}{4} + \frac{0.5}{5}\right\}$$

(c) Bounded sum

$$\mu_{\underline{T}\oplus\underline{R}}(x) = \min\{1, \mu_{\underline{T}}(x)+\mu_{\underline{R}}(x)\} = \min\left\{1, \left\{\frac{0}{0} + \frac{0.3}{1} + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.3}{4} + \frac{1.5}{5}\right\}\right\} = \left\{\frac{0}{0} + \frac{0.3}{1} + \frac{1.0}{2} + \frac{1.0}{3} + \frac{1.0}{4} + \frac{1.0}{5}\right\}$$

(d) Bounded difference

$$\mu_{\underline{T}\ominus\underline{R}}(x) = \max\{0, \mu_{\underline{T}}(x)-\mu_{\underline{R}}(x)\} = \max\left\{0, \left\{\frac{0}{0} + \frac{0.1}{1} + \frac{0.4}{2} + \frac{0.6}{3} + \frac{0.5}{4} + \frac{0.5}{5}\right\}\right\} = \left\{\frac{0}{0} + \frac{0.1}{1} + \frac{0.4}{2} + \frac{0.6}{3} + \frac{0.5}{4} + \frac{0.5}{5}\right\}$$

7.6 Review Questions

1. Define classical sets and fuzzy sets.
2. State the importance of fuzzy sets.
3. What are the methods of representation of a classical set?
4. Discuss the operations of crisp sets.
5. List the properties of classical sets.
6. What is meant by characteristic function?
7. Write the function theoretic form representation of crisp set operations.
8. Justify the following statement: "Partial membership is allowed in fuzzy sets."
9. Discuss in detail the operations and properties of fuzzy sets.
10. Represent the fuzzy sets operations using Venn diagram.
11. What is the cardinality of a fuzzy set? Whether a power set can be formed for a fuzzy set?
12. Apart from basic operations, state few other operations involved in fuzzy sets.

13. Compare and contrast classical logic and fuzzy logic.
14. Why the excluded middle law does not get satisfied in fuzzy logic?
15. Describe the importance of fuzzy sets and its application in engineering sector.

7.7 Exercise Problems

1. Find the cardinality of the given set:

$$A = \{1, 3, 5, 7, 9\}$$

2. Consider set $X = \{2, 4, 6, 8, 10\}$. Find its power set, cardinality and cardinality of power set.
3. Show the following fuzzy sets satisfy DeMorgan's law:

(a) $\mu_A(x) = \frac{1}{1+5x}$

(b) $\mu_B(x) = \left(\frac{1}{1+5x}\right)^{1/2}$

4. Consider two fuzzy sets

$$\underline{A} = \left\{\frac{1}{2.0} + \frac{0.65}{4.0} + \frac{0.5}{6.0} + \frac{0.35}{8.0} + \frac{0}{10.0}\right\}$$

$$\underline{B} = \left\{\frac{0}{2.0} + \frac{0.35}{4.0} + \frac{0.5}{6.0} + \frac{0.65}{8.0} + \frac{1}{10.0}\right\}$$

Find the following:

(a) $\underline{A} \cup \underline{B}$; (b) $\underline{A} \cap \underline{B}$; (c) $\overline{\underline{A}}$; (d) $\overline{\underline{B}}$;
(e) $\overline{\underline{A}} \cap \overline{\underline{B}}$; (f) $\overline{\underline{A}} \cup \overline{\underline{B}}$; (g) $\overline{\underline{A} \cup \underline{B}}$;
(h) $\overline{\underline{A} \cap \underline{B}}$; (i) $\underline{A} \cup \overline{\underline{A}}$; (j) $\underline{A} \cap \overline{\underline{A}}$;
(k) $\underline{B} \cup \overline{\underline{B}}$; (l) $\underline{B} \cap \overline{\underline{B}}$

5. We want to compare two liquid level controllers for their control levels and flow speed. The following values of flow speed and liquid control levels were recorded with a standard liquid flow monitor:

Flow speed	Control level 1	Control level 2
0	0	0
20	0.5	0.45
40	0.35	0.55
60	0.75	0.65
80	0.95	0.9
100	1	1

Given the universe of discourse is $X = \{0, 20, 40, 60, 80, 100\}$ and the membership functions

$$\underline{L}_1 = \left\{\frac{0}{0} + \frac{0.5}{20} + \frac{0.35}{40} + \frac{0.75}{60} + \frac{0.95}{80} + \frac{1}{100}\right\}$$

$$\underline{L}_2 = \left\{\frac{0}{0} + \frac{0.45}{20} + \frac{0.55}{40} + \frac{0.65}{60} + \frac{0.9}{80} + \frac{1}{100}\right\}$$

find the following memberships using standard set operations:

(a) $\mu_{\underline{L}_1 \cup \underline{L}_2}(x)$; (b) $\mu_{\underline{L}_1 \cap \underline{L}_2}(x)$; (c) $\mu_{\overline{\underline{L}_1}}(x)$;
(d) $\mu_{\overline{\underline{L}_2}}(x)$; (e) $\mu_{\overline{\underline{L}_1 \cup \underline{L}_2}}(x)$; (f) $\mu_{\overline{\underline{L}_1 \cap \underline{L}_2}}(x)$;
(g) $\mu_{\underline{L}_1 \cap \overline{\underline{L}_2}}$; (h) $\mu_{\underline{L}_1 \cup \overline{\underline{L}_2}}(x)$; (i) $\mu_{\underline{L}_1 \cup \overline{\underline{L}_1}}(x)$;
(j) $\mu_{\underline{L}_2 \cup \overline{\underline{L}_2}}(x)$

6. Consider two membership functions as follows:

For fuzzy set $\underline{A}$: $\mu_A(x) = \frac{|(60-x)|}{8} + 1$

For fuzzy set $\underline{B}$: $\mu_B(x) = \frac{|(40-x)|}{8} + 1$

Find the following:

(a) $\underline{A} \cup \underline{B}$; (b) $\underline{A} \cap \underline{B}$; (c) $\overline{\underline{A}}$; (d) $\overline{\underline{B}}$;
(e) $\overline{\underline{A} \cup \underline{B}}$; (f) $\overline{\underline{A} \cap \underline{B}}$

7. Let X be the universe of satellites of interest, as defined below:

$$X = \{a12,\ x15,\ b16,\ f4,\ f900,\ v111\}$$

Let $\underset{\sim}{A}$ be the fuzzy set of INSAT-V satellite:

$$\underset{\sim}{A} = \left\{\frac{0.2}{a12} + \frac{0.3}{x15} + \frac{1}{b16} + \frac{0.1}{f4} + \frac{0.5}{v111}\right\}$$

Let $\underset{\sim}{B}$ be the fuzzy set of INSAT-B satellite:

$$\underset{\sim}{B} = \left\{\frac{0.1}{a12} + \frac{0.25}{x15} + \frac{0.9}{b16} + \frac{0.7}{f4} + \frac{0.3}{f900} + \frac{0.2}{v111}\right\}$$

Find the following sets of combinations for these two sets:

(a) $\underset{\sim}{A} \cup \underset{\sim}{B}$; (b) $\underset{\sim}{A} \cap \underset{\sim}{B}$; (c) $\overline{\underset{\sim}{A}}$; (d) $\overline{\underset{\sim}{B}}$;

(e) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$; (f) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}}$; (g) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}}$;

(h) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}}$; (i) $\underset{\sim}{A} | \underset{\sim}{B}$; (j) $\underset{\sim}{B} | \underset{\sim}{A}$; (k) $\underset{\sim}{A} \cup \overline{\underset{\sim}{A}}$;

(l) $\underset{\sim}{A} \cap \overline{\underset{\sim}{A}}$; (m) $\underset{\sim}{B} \cup \overline{\underset{\sim}{B}}$; (n) $\underset{\sim}{B} \cap \overline{\underset{\sim}{B}}$

8. The discretized membership functions (in nondimensional units) for a UJT (uni-junction transistor) and BJT (bipolar junction transistor) are given below:

$$\mu_{\underset{\sim}{T1}} = \left\{\frac{0}{0} + \frac{0.2}{1} + \frac{0.3}{2} + \frac{0.6}{3} + \frac{0.9}{4} + \frac{1}{5}\right\}$$

$$\mu_{\underset{\sim}{T2}} = \left\{\frac{0}{0} + \frac{0.1}{1} + \frac{0.2}{2} + \frac{0.3}{3} + \frac{0.4}{4} + \frac{0.7}{5}\right\}$$

For the two fuzzy sets, perform the following calculations:

(a) $\mu_{\underset{\sim}{T1}} \vee \mu_{\underset{\sim}{T2}}$; (b) $\mu_{\underset{\sim}{T1}} \wedge \mu_{\underset{\sim}{T2}}$; (c) $\overline{\mu_{\underset{\sim}{T1}}}$;

(d) $\overline{\mu_{\underset{\sim}{T2}}}$; (e) $\overline{\mu_{\underset{\sim}{T1}} \wedge \mu_{\underset{\sim}{T2}}} = \overline{\mu_{\underset{\sim}{T1}}} \vee \overline{\mu_{\underset{\sim}{T2}}}$

9. Consider a local area network (LAN) of interconnected workstations that communicate using Ethernet protocols at a maximum rate of 12 Mbit/s. The two fuzzy sets given below represent the loading of the LAN:

$$\mu_{\underset{\sim}{S}}(x) = \left\{\frac{1.0}{0} + \frac{1.0}{1} + \frac{0.8}{2} + \frac{0.2}{5} + \frac{0.1}{7} + \frac{0.0}{9} + \frac{0.0}{10}\right\}$$

$$\mu_{\underset{\sim}{C}}(x) = \left\{\frac{0.0}{0} + \frac{0.0}{1} + \frac{0.0}{2} + \frac{0.5}{5} + \frac{0.7}{7} + \frac{0.8}{9} + \frac{1.0}{10}\right\}$$

where $\underset{\sim}{S}$ represents silent and $\underset{\sim}{C}$ represents congestion. Perform algebraic sum, algebraic product, bounded sum and bounded difference over the two fuzzy sets.

10. Consider the following two fuzzy sets:

$$\underset{\sim}{X} = \left\{\frac{0.1}{0} + \frac{0.2}{1} + \frac{0.3}{2} + \frac{0.4}{3} + \frac{0.5}{4}\right\}$$

$$\underset{\sim}{Y} = \left\{\frac{0.5}{0} + \frac{0.4}{1} + \frac{0.3}{2} + \frac{0.2}{3} + \frac{0.1}{4}\right\}$$

Perform the following operations over the given fuzzy sets:

(a) $\underset{\sim}{X} \cup \underset{\sim}{Y}$; (b) $\underset{\sim}{X} \cap \underset{\sim}{Y}$; (c) $\overline{\underset{\sim}{X}}$; (d) $\overline{\underset{\sim}{Y}}$;

(e) $\overline{\underset{\sim}{X} \cup \underset{\sim}{Y}}$; (f) $\overline{\underset{\sim}{X} \cap \underset{\sim}{Y}}$; (g) $\underset{\sim}{X} \cup \overline{\underset{\sim}{X}}$;

(h) $\underset{\sim}{X} \cap \overline{\underset{\sim}{X}}$; (i) $\underset{\sim}{X} \cup \overline{\underset{\sim}{Y}}$; (j) $\underset{\sim}{Y} \cup \overline{\underset{\sim}{X}}$;

(k) algebraic sum; (l) algebraic product;

(m) bounded sum; (n) bounded difference

Classical Relations and Fuzzy Relations

8

Learning Objectives

- Definition of classical relations and fuzzy relations.
- Formulation of Cartesian product of a relation.
- Operations and properties of classical relations and fuzzy relations.
- Composition of relations – max-min and max-product composition.
- Description on classical and fuzzy equivalence and tolerance relations.
- A short note on noninteractive fuzzy sets.

8.1 Introduction

Relationships between objects are the basic concepts involved in decision making and other dynamic system applications. The relations are also associated with graph theory, which has a great impact on designs and data manipulations. Relations represent mappings between sets and connectives in logic. A classical binary relation represents the presence or absence of a connection or interaction or association between the elements of two sets. Fuzzy binary relations are a generalization of crisp binary relations, and they allow various degrees of relationship (association) between elements. In other words, fuzzy relations impart degrees of strength to such connections and associations. In a fuzzy binary relation, the degree of association is represented by membership grades in the same way as the degree of set membership is represented in a fuzzy set. This chapter discusses the basic concepts and operations on fuzzy relations, and the composition between relations is studied via the max-min and max-product compositions. The properties and the cardinality of fuzzy relations are also discussed. Other topics discussed include the tolerance and equivalence relations on both crisp and fuzzy relations.

8.2 Cartesian Product of Relation

An ordered r-tuple is an ordered sequence of r-elements expressed in the form $(a_1, a_2, a_3, \ldots, a_r)$. An unordered r-tuple is a collection of r-elements without any restrictions in order. For $r = 2$, the r-tuple is called an ordered pair. For crisp sets $A_1, A_2, \ldots, A_r$, the set of all r-tuples $(a_1, a_2, a_3, \ldots, a_r)$, where $a_1 \in A_1, a_2 \in A_2, \ldots, a_r \in A_r$, is called the Cartesian product of $A_1, A_2, \ldots, A_r$ and is denoted by $A_1 \times A_2 \times \cdots \times A_r$. The Cartesian product of two or more sets is not the same as the arithmetic product of two or more sets. If all the a_r's are identical and equal to A, then the Cartesian product $A_1 \times A_2 \times \cdots \times A_r$ is denoted as A^r.

8.3 Classical Relation

An r-ary relation over $A_1, A_2, \ldots, A_r$ is a subset of the Cartesian product $A_1 \times A_2 \times \cdots \times A_r$. When $r = 2$, the relation is a subset of the Cartesian product $A_1 \times A_2$. This is called a binary relation from A_1 to A_2. When three, four or five sets are involved in the subset of full Cartesian product then the relations are called ternary, quaternary and quinary, respectively. Generally, the discussions are centered on binary relations.

Consider two universes X and Y; their Cartesian product $X \times Y$ is given by

$$X \times Y = \{(x, y) | x \in X, y \in Y\}$$

Here the Cartesian product forms an ordered pair of every $x \in X$ with every $y \in Y$. Every element in X is completely related to every element in Y. The characteristic function, denoted by χ, gives the strength of the relationship between ordered pair of elements in each universe. If it takes unity as its value, then complete relationship is found; if the value is zero, then there is no relationship, i.e.,

$$\chi_{X \times Y}(x, y) = \begin{cases} 1, & (x, y) \in X \times Y \\ 0, & (x, y) \notin X \times Y \end{cases}$$

When the universes or sets are finite, then the relation is represented by a matrix called relation matrix. An r-dimensional relation matrix represents an r-ary relation. Thus, binary relations are represented by two-dimensional matrices.

Consider the elements defined in the universes X and Y as follows:

$$X = \{2, 4, 6\}; \quad Y = \{p, q, r\}$$

The Cartesian product of these two sets leads to

$$X \times Y = \{(p, 2), (p, 4), (p, 6), (q, 2), (q, 4), (q, 6), (r, 2), (r, 4), (r, 6)\}$$

From this set one may select a subset such that

$$R = \{(p, 2), (q, 4), (r, 4), (r, 6)\}$$

Subset R can be represented using a coordinate diagram as shown in Figure 8-1.
The relation could equivalently be represented using a matrix as follows:

R	p	q	r
2	1	0	0
4	0	1	1
6	0	0	1

The relation between sets X and Y may also be expressed by mapping representations as shown in Figure 8-2.

A binary relation in which each element from first set X is not mapped to more than one element in second set Y is called a function and is expressed as

$$R : X \rightarrow Y$$

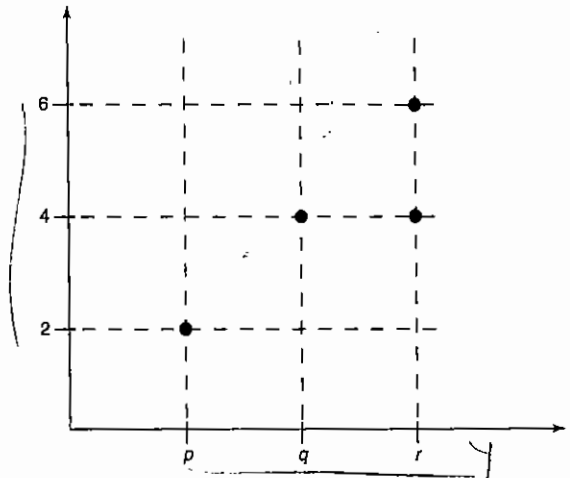


Figure 8-1 Coordinate diagram of a relation.

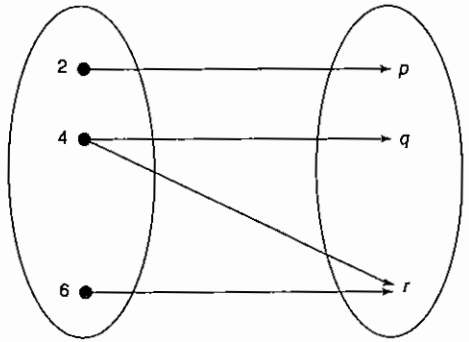


Figure 8-2 Mapping representation of a relation.

Figures 8-3 (A) and (B) show the illustration of $R : X \rightarrow Y$. Figure 8-3 shows mapping of an unconstrained relation. A more general crisp relation, R, exists when matches between elements in two universes are constrained. The characteristic function is used to assign values of relationship in the mapping of the Cartesian space $X \times Y$ to the binary values (0, 1) and is given by

$$\chi_R(x, y) = \begin{cases} 1, & (x, y) \in R \\ 0, & (x, y) \notin R \end{cases}$$

The constrained Cartesian product for sets when $r = 2$ (i.e., $A \times A = A^2$) is called identity relation, and the unconstrained Cartesian product for sets when $r = 2$ is called universal relation. Consider set $A = \{2, 4, 6\}$.

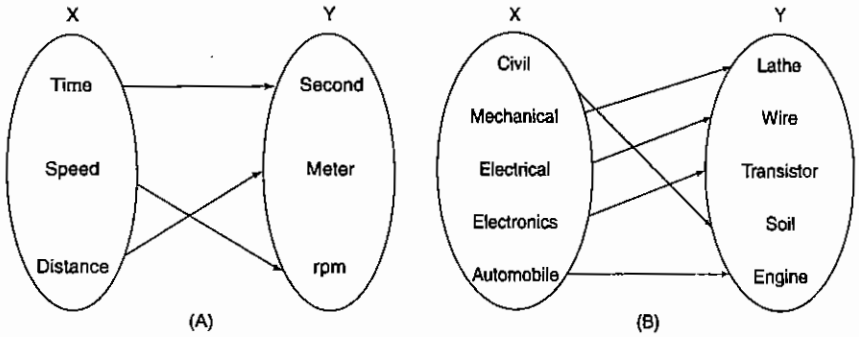


Figure 8-3 Illustrations of $R: X \to Y$.

Then universal relation (U_A) and identity relation (I_A) are given as follows:

$$U_A = \{(2,2),(2,4),(2,6),(4,2),(4,4),(4,6),(2,6),(4,6),(6,6)\}$$
$$I_A = \{(2,2),(4,4),(6,6)\}$$

8.3.1 Cardinality of Classical Relation

Consider n elements of universe X being related to m elements of universe Y. When the cardinality of $X = n_X$ and the cardinality of $Y = n_Y$, then the cardinality of relation R between the two universes is

$$n_{X\times Y} = n_X \times n_Y$$

The cardinality of the power set $P(X \times Y)$ describing the relation is given by

$$n_{P(X\times Y)} = 2^{(n_X n_Y)}$$

8.3.2 Operations on Classical Relations

Let R and S be two separate relations on the Cartesian universe $X \times Y$. The null relation and the complete relation are defined by the relation matrices ϕ_R and E_R. An example of a 3×3 form of the ϕ_R and E_R matrices is given below:

$$\phi_R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{and} \quad E_R = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Function-theoretic operations for the two crisp relations (R, S) are defined as follows:

1. Union

$$R \cup S \to \chi_{R\cup S}(x,y) : \chi_{R\cup S}(x,y) = \max[\chi_R(x,y), \chi_S(x,y)]$$

2. Intersection

$$R \cap S \to \chi_{R\cap S}(x,y) : \chi_{R\cap S}(x,y) = \min[\chi_R(x,y), \chi_S(x,y)]$$

3. Complement

$$\overline{R} \to \chi_{\overline{R}}(x,y) : \chi_{\overline{R}}(x,y) = 1 - \chi_{\overline{R}}(x,y)$$

4. Containment

$$R \subset S \to \chi_R(x,y) : \chi_R(x,y) \le \chi_S(x,y)$$

5. Identity

$$\phi \to \phi_R \quad \text{and} \quad X \to E_R$$

8.3.3 Properties of Crisp Relations

The properties of classical set operations such as commutativity, associativity, distributivity, involution and idempotency also hold good for classical relations. Similarly De Morgan's law and excluded middle laws hold good for crisp relations as they do for crisp sets. The null relation ϕ_R is analogous to null set ϕ and complete relation E_R is analogous to whole set X.

8.3.4 Composition of Classical Relations

The operation executed on two compatible binary relations to get a single binary relation is called composition.

Let R be a relation that maps elements from universe X to universe Y and S be a relation that maps elements from universe Y to universe Z. The two binary relations R and S are compatible if

$$R \subseteq X \times Y \quad \text{and} \quad S \subseteq Y \times Z$$

In other words, the second set in R must be the same as the first set in S. On the basis of this explanation, a relation T can be formed that relates the same elements of universe X contained in R with the same elements of universe Z contained in S. This type of relation can be obtained by performing the composition operation over the two given relations. The composition between the two relations is denoted by $R \circ S$. Consider the universal sets given by

$$X = \{a_1, a_2, a_3\}; \quad Y = \{b_1, b_2, b_3\}; \quad Z = \{c_1, c_2, c_3\}$$

Let the relations R and S be formed as

$$R = X \times Y = \{(a_1,b_1),(a_1,b_2),(a_2,b_2),(a_3,b_3)\}$$
$$S = Y \times Z = \{(b_1,c_1),(b_2,c_3),(b_3,c_2)\}$$

Relations R and S are illustrated in Figure 8-4. From Figure 8-4, it can be inferred that

$$T = R \circ S = \{(a_1,c_1),(a_2,c_3),(a_3,c_2),(a_1,c_3)\}$$

The representation of relations R and S in matrix form is given as

$$R = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \end{array} \begin{array}{c} \begin{array}{ccc} b_1 & b_2 & b_3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}; \quad S = \begin{array}{c} \\ b_1 \\ b_2 \\ b_3 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

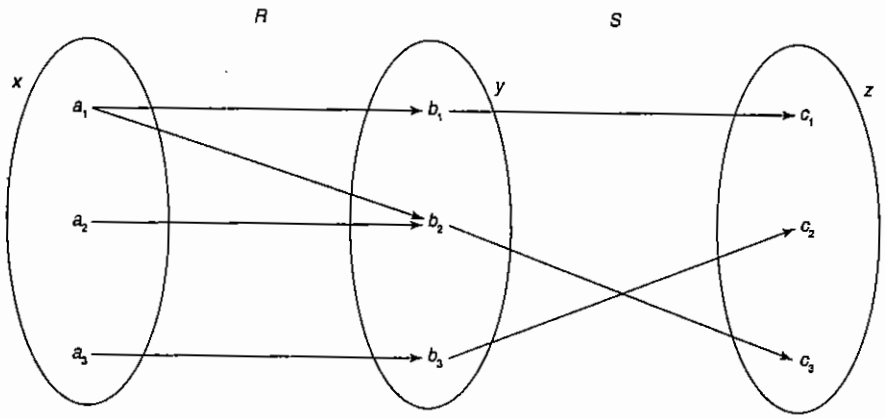


Figure 8-4 Illustration of relations R and S.

Composition $T = R \circ S$ is represented in matrix form as

$$T = \begin{array}{c} \\ a_1 \\ a_2 \\ a_3 \end{array} \begin{array}{c} \begin{array}{ccc} c_1 & c_2 & c_3 \end{array} \\ \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \end{array}$$

This matrix also leads to

$$T = R \circ S = \{(a_1, c_1), (a_2, c_3), (a_3, c_2), (a_1, c_3)\}$$

as expected. The composition operations are of two types:

1. Max-min composition
2. Max-product composition.

The max-min composition is defined by the function theoretic expression as

$$T = R \circ S$$
$$\chi_T(x, z) = \bigvee_{y \in Y} [\chi_R(x, y) \wedge \chi_S(y, z)]$$

The max-product composition is defined by the function theoretic expression as

$$T = R \circ S$$
$$\chi_T(x, z) = \bigvee_{y \in Y} [\chi_R(x, y) \cdot \chi_S(y, z)]$$

The max-product composition is sometimes also referred to as max-dot composition. Some properties of the composition operation are described in Table 8-1.

Table 8-1 Few properties of composition operation

Associative	$(R \circ S) \circ M = R \circ (S \circ M)$
Commutative	$R \circ S \neq S \circ R$
Inverse	$(R \circ S)^{-1} = S^{-1} \circ R^{-1}$

8.4 Fuzzy Relations

Fuzzy relations relate elements of one universe (say X) to those of another universe (say Y) through the Cartesian product of the two universes. These can also be referred to as fuzzy sets defined on universal sets, which are Cartesian products. A fuzzy relation is based on the concept that everything is related to some extent or unrelated.

A fuzzy relation is a fuzzy set defined on the Cartesian product of classical sets $\{X_1, X_2, \ldots, X_n\}$ where tuples $(x_1, x_2, \ldots, x_n)$ may have varying degrees of membership $\mu_R(x_1, x_2, \ldots, x_n)$ within the relation. That is,

$$R(X_1, X_2, \ldots, X_n) = \int_{X_1 \times X_2 \times \cdots \times X_n} \mu_R(x_1, x_2, \ldots, x_n)|(x_1, x_2, \ldots, x_n), \quad x_i \in X_i$$

A fuzzy relation between two sets X and Y is called binary fuzzy relation and is denoted by $R(X, Y)$. A binary relation $R(X, Y)$ is referred to as bipartite graph when $X \neq Y$. The binary relation on a single set X is called directed graph or digraph. This relation occurs when $X = Y$ and is denoted as $R(X, X)$ or $R(X^2)$.

Let

$$\underset{\sim}{X} = \{x_1, x_2, \ldots, x_n\} \quad \text{and} \quad \underset{\sim}{Y} = \{y_1, y_2, \ldots, y_m\}$$

Fuzzy relation $\underset{\sim}{R}(\underset{\sim}{X}, \underset{\sim}{Y})$ can be expressed by an $n \times m$ matrix as follows:

$$\underset{\sim}{R}(\underset{\sim}{X}, \underset{\sim}{Y}) = \begin{bmatrix} \mu_R(x_1, y_1) & \mu_R(x_1, y_2) & \cdots & \mu_R(x_1, y_m) \\ \mu_R(x_2, y_1) & \mu_R(x_2, y_2) & \cdots & \mu_R(x_2, y_m) \\ \vdots & \vdots & \cdots & \vdots \\ \mu_R(x_n, y_1) & \mu_R(x_n, y_2) & \cdots & \mu_R(x_n, y_m) \end{bmatrix}$$

The matrix representing a fuzzy relation is called fuzzy matrix. A fuzzy relation $\underset{\sim}{R}$ is a mapping from Cartesian space $X \times Y$ to the interval [0, 1] where the mapping strength is expressed by the membership function of the relation for ordered pairs from the two universes $[\mu_R(x, y)]$.

A fuzzy graph is a graphical representation of a binary fuzzy relation. Each element in $\underset{\sim}{X}$ and $\underset{\sim}{Y}$ corresponds to a node in the fuzzy graph. The connection links are established between the nodes by the elements of $\underset{\sim}{X} \times \underset{\sim}{Y}$ with nonzero membership grades in $R(X, Y)$. The links may also be present in the form of arcs. These links are labeled with the membership values as $\mu_R(x_i, y_j)$. When $X \neq Y$, the link connecting the two nodes is an undirected binary graph called bipartite graph. Here, each of the sets X and Y can be represented by a set of nodes such that the nodes corresponding to one set are clearly differentiated from the nodes representing the other set. When $X = Y$, a node is connected to itself and directed links are used; in such a case, the fuzzy graph is called directed graph. Here, only one set of nodes corresponding to set X is used.

The domain of a binary fuzzy relation $R(X, Y)$ is the fuzzy set, *dom* $R(X, Y)$, having the membership function as

$$\mu_{\text{domain } R}(x) = \max_{y \in Y} \mu_R(x, y) \quad \forall x \in X$$

The range of a binary fuzzy relation $R(X, Y)$ is the fuzzy set, *ran* $R(X, Y)$, having the membership function as

$$\mu_{\text{range } R}(y) = \max_{x \in X} \mu_R(x, y) \quad \forall y \in Y$$

Consider a universe $X = \{x_1, x_2, x_3, x_4\}$ and the binary fuzzy relation on X as

$$\underset{\sim}{R}(X, X) = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} \begin{array}{c} \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{bmatrix} 0.2 & 0 & 0.5 & 0 \\ 0 & 0.3 & 0.7 & 0.8 \\ 0.1 & 0 & 0.4 & 0 \\ 0 & 0.6 & 0 & 1 \end{bmatrix} \end{array}$$

The bipartite graph and simple fuzzy graph of $\underset{\sim}{R}(X, X)$ is shown in Figures 8-5(A) and (B), respectively. Let

$$\underset{\sim}{X} = \{x_1, x_2, x_3, x_4\} \quad \text{and} \quad \underset{\sim}{Y} = \{y_1, y_2, y_3, y_4\}$$

Let $\underset{\sim}{R}$ be a relation from $\underset{\sim}{X}$ to $\underset{\sim}{Y}$ given by

$$\underset{\sim}{R} = \frac{0.2}{(x_1, y_3)} + \frac{0.4}{(x_1, y_2)} + \frac{0.1}{(x_2, y_2)} + \frac{0.6}{(x_2, y_3)} + \frac{1.0}{(x_3, y_3)} + \frac{0.5}{(x_3, y_1)}$$

The corresponding fuzzy matrix for relation $\underset{\sim}{R}$ is

$$\underset{\sim}{R} = \begin{array}{c} \\ x_1 \\ x_2 \\ x_3 \end{array} \begin{array}{c} \begin{array}{ccc} y_1 & y_2 & y_3 \end{array} \\ \begin{bmatrix} 0 & 0.4 & 0.2 \\ 0 & 0.1 & 0.6 \\ 0.5 & 0 & 1.0 \end{bmatrix} \end{array}$$

The graph of the above relation $\underset{\sim}{R} = \underset{\sim}{X} \times \underset{\sim}{Y}$ is shown in Figure 8-6.

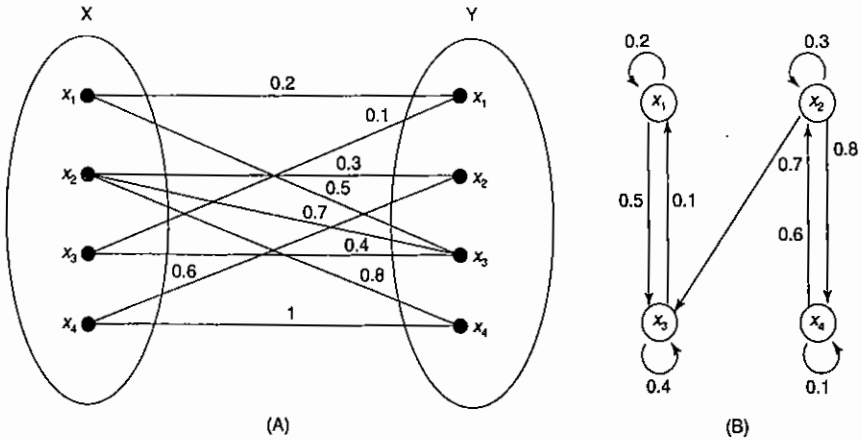


Figure 8-5 Graphical representation of fuzzy relations: (A) Bipartite graph; (B) simple fuzzy graph.

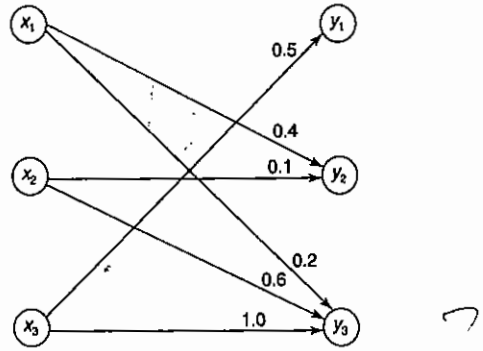


Figure 8-6 Graph of fuzzy relation.

8.4.1 Cardinality of Fuzzy Relations

The cardinality of fuzzy sets on any universe is infinity; hence the cardinality of a fuzzy relation between two or more universes is also infinity. This is mainly a result of the occurrence of partial membership in fuzzy sets and fuzzy relations.

8.4.2 Operations on Fuzzy Relations

The basic operations on fuzzy sets also apply on fuzzy relations. Let $\underset{\sim}{R}$ and $\underset{\sim}{S}$ be fuzzy relations on the Cartesian space $X \times Y$. The operations that can be performed on these fuzzy relations are described below:

1. Union

$$\mu_{\underset{\sim}{R} \cup \underset{\sim}{S}}(x, y) = \max\left[\mu_{\underset{\sim}{R}}(x, y), \mu_{\underset{\sim}{S}}(x, y)\right]$$

2. Intersection

$$\mu_{\underset{\sim}{R} \cap \underset{\sim}{S}}(x, y) = \min\left[\mu_{\underset{\sim}{R}}(x, y), \mu_{\underset{\sim}{S}}(x, y)\right]$$

3. Complement

$$\mu_{\overline{\underset{\sim}{R}}}(x, y) = 1 - \mu_{\underset{\sim}{R}}(x, y)$$

4. Containment

$$\underset{\sim}{R} \subset \underset{\sim}{S} \Rightarrow \mu_{\underset{\sim}{R}}(x, y) \leq \mu_{\underset{\sim}{S}}(x, y)$$

5. Inverse: The inverse of a fuzzy relation R on $X \times Y$ is denoted by R^{-1}. It is a relation on $Y \times X$ defined by $R^{-1}(y, x) = R(x, y)$ for all pairs $(y, x) \in Y \times X$.

6. Projection: For a fuzzy relation $R(X, Y)$, let $[R \downarrow Y]$ denote the projection of R onto Y. Then $[R \downarrow Y]$ is a fuzzy relation in Y whose membership function is defined by

$$\mu_{[R \downarrow Y]}(x, y) = \max_{x} \mu_R(x, y)$$

The projection concept can be extended to an n-ary relation $R(x_1, x_2, \ldots, x_n)$.

8.4.3 Properties of Fuzzy Relations

Like classical relations, the properties of commutativity, associativity, distributivity, idempotency and identity also hold good for fuzzy relations. DeMorgan's laws hold good for fuzzy relations as they do for classical relations. The null relation $\phi_{\tilde{R}}$ and complete relation E_R are analogous to the null set ϕ and the whole set E, respectively, in set theoretic form. The excluded middle laws are not satisfied in fuzzy relations as for fuzzy sets. This is because a fuzzy relation $\tilde{R}$ is also a fuzzy set, and there exists an overlap between a relation and its complement. Hence

$$\tilde{R} \cup \overline{\tilde{R}} \neq \tilde{E} \quad \text{(whole set)}$$

$$\tilde{R} \cap \overline{\tilde{R}} \neq \tilde{\phi} \quad \text{(null set)}$$

8.4.4 Fuzzy Composition

Before understanding the fuzzy composition techniques, let us learn about the fuzzy Cartesian product. Let $\tilde{A}$ be a fuzzy set on universe X and $\tilde{B}$ be a fuzzy set on universe Y. The Cartesian product over $\tilde{A}$ and $\tilde{B}$ results in fuzzy relation $\tilde{R}$ and is contained within the entire (complete) Cartesian space, i.e.,

$$\tilde{A} \times \tilde{B} = \tilde{R}$$

where

$$\tilde{R} \subset X \times Y$$

The membership function of fuzzy relation is given by

$$\mu_{\tilde{R}}(x, y) = \mu_{\tilde{A} \times \tilde{B}}(x, y) = \min\left[\mu_{\tilde{A}}(x), \mu_{\tilde{B}}(y)\right]$$

The Cartesian product is not an operation similar to arithmetic product. Cartesian product $\tilde{R} = \tilde{A} \times \tilde{B}$ is obtained in the same way as the cross-product of two vectors. For example, for a fuzzy set $\tilde{A}$ that has three elements (hence column vector of size 3×1) and a fuzzy set $\tilde{B}$ that has four elements (hence row vector of size 1×4), the resulting fuzzy relation $\tilde{R}$ will be represented by a matrix of size 3×4, i.e., $\tilde{R}$ will have three rows and four columns.

Now let's discuss the composition of fuzzy relations. There are two types of fuzzy composition techniques:

1. Fuzzy max-min composition
2. Fuzzy max-product composition

There also exists fuzzy min-max composition method, but the most commonly used technique is fuzzy max-min composition. Let $\tilde{R}$ be fuzzy relation on Cartesian space $X \times Y$, and $\tilde{S}$ be fuzzy relation on Cartesian space $Y \times Z$.

The max-min composition of $R(X, Y)$ and $S(Y, Z)$, denoted by $R(X, Y) \circ S(Y, Z)$ is defined by $T(X, Z)$ as

$$\mu_T(x, z) = \mu_{\tilde{R} \circ \tilde{S}}(x, z) = \max_{y \in Y} \{\min\left[\mu_{\tilde{R}}(x, y), \mu_{\tilde{S}}(y, z)\right]\}$$

$$= \bigvee_{y \in Y} [\mu_{\tilde{R}}(x, y) \wedge \mu_{\tilde{S}}(y, z)] \quad \forall x \in X, \quad z \in Z$$

The min-max composition of $R(X, Y)$ and $S(Y, Z)$, denoted as $R(X, Y) \circ S(Y, Z)$, is defined by $T(X, Z)$ as

$$\mu_T(x, z) = \mu_{\tilde{R} \circ \tilde{S}}(x, z) = \min_{y \in Y} \{\max[\mu_{\tilde{R}}(x, y), \mu_{\tilde{S}}(y, z)]\} = \bigwedge_{y \in Y} [\mu_{\tilde{R}}(x, y) \vee \mu_{\tilde{S}}(y, z)] \quad \forall x \in X, \, z \in Z$$

From the above definitions it can be noted that

$$\overline{R(X, Y) \circ S(Y, Z)} = \overline{R(X, Y)} \circ \overline{S(Y, Z)}$$

The max-min composition is most widely used, hence the problems discussed in this chapter are limited to max-min composition. The max-product composition of $R(X, Y)$ and $S(Y, Z)$, denoted as $R(X, Y) \cdot S(Y, Z)$, is defined by $T(X, Z)$ as

$$\mu_T(x, z) = \mu_{\tilde{R} \cdot \tilde{S}}(x, z) = \max_{y \in Y} [\mu_{\tilde{R}}(x, y) \cdot \mu_{\tilde{S}}(y, z)]$$

$$= \bigvee_{y \in Y} [\mu_{\tilde{R}}(x, y) \cdot \mu_{\tilde{S}}(y, z)]$$

The properties of fuzzy composition can be given as follows:

$$\tilde{R} \circ \tilde{S} \neq \tilde{S} \circ \tilde{R}$$

$$(\tilde{R} \circ \tilde{S})^{-1} = \tilde{S}^{-1} \circ \tilde{R}^{-1}$$

$$(\tilde{R} \circ \tilde{S}) \circ \tilde{M} = \tilde{R} \circ (\tilde{S} \circ \tilde{M})$$

8.5 Tolerance and Equivalence Relations

Relations possess various useful properties. Some of them are discussed in this section. Relations play a major role in graph theory. The three characteristic properties of relations discussed are: reflexivity, symmetry and transitivity. The antonyms of these properties are: irreflexivity, asymmetry and nontransitivity.

1. A relation is said to be reflexive if every vertex (node) in the graph originates a single loop as shown in Figure 8-7.
2. A relation is said to be symmetric if for every edge pointing from vertex i to vertex j, there is an edge pointing in the opposite direction, i.e., from vertex j to vertex i where $i, j = 1, 2, 3, \ldots$. Figure 8-8 represents a symmetric relation.
3. A relation is said to be transitive if for every pair of edges in the graph – one pointing from vertex i to vertex j and the other pointing from vertex j to vertex k, there is an edge pointing from vertex i to vertex k.

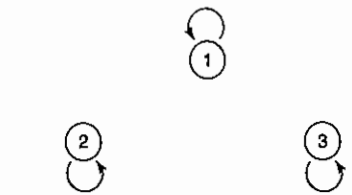


Figure 8-7 Three-vertex node – reflexive property.

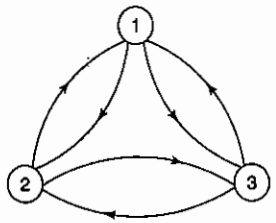


Figure 8-8 Three-vertex node – symmetry property.

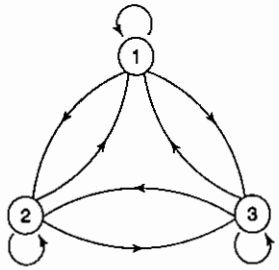


Figure 8-9 Three-vertex graph – transitive property.

Figure 8-9 represents a transitive relation. Here an arrow points from node 1 to node 2 and another arrow extends from node 2 to node 3. There is also an arrow from node 1 to node 3.

8.5.1 Classical Equivalence Relation

Let relation R on universe X be a relation from X to X. Relation R is an equivalence relation if the following three properties are satisfied:

1. Reflexivity
2. Symmetry
3. Transitivity

The function theoretic forms of representation of these properties are as follows:

1. Reflexivity

$$\chi_R(x_i, x_i) = 1 \text{ or } (x_i, x_i) \in R$$

2. Symmetry

$$\chi_R(x_i, x_j) = \chi_R(x_j, x_i)$$
$$\text{i.e., } (x_i, x_j) \in R \Rightarrow (x_j, x_i) \in R$$

3. Transitivity

$$\chi_R(x_i, x_j) \text{ and } \chi_R(x_j, x_k) = 1, \text{ so } \chi_R(x_i, x_k) = 1$$
$$\text{i.e., } (x_i, x_j) \in R (x_j, x_k) \in R, \text{ so } (x_i, x_k) \in R$$

The best example of an equivalence relation is the relation of similarity among triangles.

8.5.2 Classical Tolerance Relation

A tolerance relation R_1 on universe X is one where the only the properties of reflexivity and symmetry are satisfied. The tolerance relation can also be called proximity relation. An equivalence relation can be formed from tolerance relation R_1 by $(n-1)$ compositions within itself, where n is the cardinality of the set that defines R_1, here it is X, i.e.

$$\underbrace{R_1^{n-1}}_{\text{Tolerance relation}} = R_1 \circ R_1 \circ \cdots \circ R_1 = \underbrace{R}_{\text{Equivalence relation}}$$

8.5.3 Fuzzy Equivalence Relation

Let $\underset{\sim}{R}$ be a fuzzy relation on universe X, which maps elements from X to X. Relation $\underset{\sim}{R}$ will be a fuzzy equivalence relation if all the three properties – reflexive, symmetry and transitivity – are satisfied. The membership function theoretic forms for these properties are represented as follows:

1. Reflexivity

$$\mu_{\underset{\sim}{R}}(x_i, x_i) = 1 \quad \forall x \in X$$

If this is not the case for few $x \in X$, then $R(X, X)$ is said to be irreflexive.

2. Symmetry

$$\mu_{\underset{\sim}{R}}(x_i, x_j) = \mu_{\underset{\sim}{R}}(x_j, x_i) \text{ for all } x_i, x_j \in X$$

If this is not satisfied for few $x_i, x_j \in X$, then $R(X, X)$ is called asymmetric.

3. Transitivity

$$\mu_{\underset{\sim}{R}}(x_i, x_j) = \lambda_1 \quad \text{and} \quad \mu_{\underset{\sim}{R}}(x_j, x_k) = \lambda_2$$
$$\Rightarrow \mu_{\underset{\sim}{R}}(x_i, x_k) = \lambda$$

where

$$\lambda = \min[\lambda_1, \lambda_2]$$
$$\text{i.e., } \mu_{\underset{\sim}{R}}(x_i, x_j) \geq \max_{x_j \in X} \min[\mu_{\underset{\sim}{R}}(x_i, x_j), \mu_{\underset{\sim}{R}}(x_j, x_k)] \quad \forall (x_i, x_k) \in X^2$$

This can also be called max-min transitive. If this is not satisfied for some members of X, then $R(X, X)$ is nontransitive. If the given transitivity inequality is not satisfied for all the members $(x_i, x_k) \in X^2$, then the relation is called as antitransitive.

The max-product transitive can also be defined. It is given by

$$\mu_{\underset{\sim}{R}}(x_i, x_k) \geq \max_{x_j \in X} \cdot [\mu_{\underset{\sim}{R}}(x_i, x_j) \bullet \mu_{\underset{\sim}{R}}(x_j, x_k)] \quad \forall (x_i, x_k) \in X^2$$

The equivalence relation discussed can also be called similarity relation.

8.5.4 Fuzzy Tolerance Relation

A binary fuzzy relation that possesses the properties of reflexivity and symmetry is called fuzzy tolerance relation or resemblance relation. The equivalence relations are a special case of the tolerance relation. The fuzzy tolerance relation can be reformed into fuzzy equivalence relation in the same way as a crisp tolerance relation is reformed into crisp equivalence relation, i.e.,

$$\underbrace{\underset{\sim}{R}_1^{n-1}}_{\substack{\text{Fuzzy}\\ \text{tolerance}\\ \text{relation}}} = \underset{\sim}{R}_1 \circ \underset{\sim}{R}_1 \circ \cdots \circ \underset{\sim}{R}_1 = \underbrace{\underset{\sim}{R}}_{\substack{\text{Fuzzy}\\ \text{equivalence}\\ \text{relation}}}$$

where "n" is the cardinality of the set that defines R_1.

8.6 Noninteractive Fuzzy Sets

The independent events in probability theory are analogous to noninteractive fuzzy sets in fuzzy theory. A noninteractive fuzzy set is defined as follows. We are defining fuzzy set $\underset{\sim}{A}$ on the Cartesian space $X = X_1 \times X_2$. Set $\underset{\sim}{A}$ is separable into two noninteractive fuzzy sets called orthogonal projections, if and only if

$$\underset{\sim}{A} = \text{OPr}_{X_1}(\underset{\sim}{A}) \times \text{OPr}_{X_2}(\underset{\sim}{A})$$

where

$$\mu_{\text{OPr}_{X_1}(\underset{\sim}{A})}(x_1) = \max_{x_2 \in X_2} \mu_{\underset{\sim}{A}}(x_1, x_2) \quad \forall x_1 \in X_1$$
$$\mu_{\text{OPr}_{X_2}(\underset{\sim}{A})}(x_2) = \max_{x_1 \in X_1} \mu_{\underset{\sim}{A}}(x_1, x_2) \quad \forall x_2 \in X_2$$

The equations represent membership functions for the orthographic projections of $\underset{\sim}{A}$ on universes X_1 and X_2, respectively.

8.7 Summary

This chapter discussed the properties and operations of crisp and fuzzy relations. The relation concept is most powerful, and is used for nonlinear simulation, classification and control. The description on composition of relations gives a view of extending fuzziness into functions. Tolerance and equivalence relations are helpful for solving similar classification problems. The noninteractivity between fuzzy sets is analogous to the assumption of independence in probability modeling.

8.8 Solved Problems

1. The elements in two sets A and B are given as

$$A = \{2, 4\} \quad \text{and} \quad B = \{a, b, c\}$$

Find the various Cartesian products of these two sets.

Solution: The various Cartesian products of these two given sets are

$$A \times B = \{(2, a), (2, b), (2, c), (4, a), (4, b), (4, c)\}$$
$$B \times A = \{(a, 2), (a, 4), (b, 2), (b, 4), (c, 2), (c, 4)\}$$
$$A \times A = A^2 = \{(2, 2), (2, 4), (4, 2), (4, 4)\}$$
$$B \times B = B^2 = \{(a, a), (a, b), (a, c), (b, a), (b, b), (b, c), (c, a), (c, b), (c, c)\}$$

2. Consider the following two fuzzy sets:

$$\underset{\sim}{A} = \left\{\frac{0.3}{x_1} + \frac{0.7}{x_2} + \frac{1}{x_3}\right\}$$
$$\text{and} \quad \underset{\sim}{B} = \left\{\frac{0.4}{y_1} + \frac{0.9}{y_2}\right\}$$

Perform the Cartesian product over these given fuzzy sets.

Solution: The fuzzy Cartesian product performed over fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$ results in fuzzy relation $\underset{\sim}{R}$ given by $\underset{\sim}{R} = \underset{\sim}{A} \times \underset{\sim}{B}$. Hence

$$\underset{\sim}{R} = \begin{bmatrix} 0.3 & 0.3 \\ 0.4 & 0.7 \\ 0.4 & 0.9 \end{bmatrix}$$

The calculation for $\underset{\sim}{R}$ is as follows:

$$\mu_{\underset{\sim}{R}}(x_1, y_1) = \min[\mu_{\underset{\sim}{A}}(x_1), \mu_{\underset{\sim}{B}}(y_1)] = \min(0.3, 0.4) = 0.3$$
$$\mu_{\underset{\sim}{R}}(x_1, y_2) = \min[\mu_{\underset{\sim}{A}}(x_1), \mu_{\underset{\sim}{B}}(y_2)] = \min(0.3, 0.9) = 0.3$$
$$\mu_{\underset{\sim}{R}}(x_2, y_1) = \min[\mu_{\underset{\sim}{A}}(x_2), \mu_{\underset{\sim}{B}}(y_2)] = \min(0.7, 0.4) = 0.4$$
$$\mu_{\underset{\sim}{R}}(x_2, y_2) = \min[\mu_{\underset{\sim}{A}}(x_2), \mu_{\underset{\sim}{B}}(y_2)] = \min(0.7, 0.9) = 0.7$$
$$\mu_{\underset{\sim}{R}}(x_3, y_1) = \min[\mu_{\underset{\sim}{A}}(x_3), \mu_{\underset{\sim}{B}}(y_1)] = \min(1, 0.4) = 0.4$$
$$\mu_{\underset{\sim}{R}}(x_3, y_2) = \min[\mu_{\underset{\sim}{A}}(x_3), \mu_{\underset{\sim}{B}}(y_2)] = \min(1, 0.9) = 0.9$$

Thus, the Cartesian product between fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$ are obtained.

3. Two fuzzy relations are given by

$$\underset{\sim}{R} = \begin{matrix} \\ x_1 \\ x_2 \end{matrix} \begin{matrix} \begin{matrix} y_1 & y_2 \end{matrix} \\ \begin{bmatrix} 0.6 & 0.3 \\ 0.2 & 0.9 \end{bmatrix} \end{matrix}$$

$$\text{and} \quad \underset{\sim}{S} = \begin{matrix} \\ y_1 \\ y_2 \end{matrix} \begin{matrix} \begin{matrix} z_1 & z_2 & z_3 \end{matrix} \\ \begin{bmatrix} 1 & 0.5 & 0.3 \\ 0.8 & 0.4 & 0.7 \end{bmatrix} \end{matrix}$$

Obtain fuzzy relation $\underset{\sim}{T}$ as a composition between the fuzzy relations.

Solution: The composition between two given fuzzy relations is performed in two ways as

(a) Max-min composition
(b) Max-product composition

(a) Max-min composition

$$\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S} = \begin{matrix} \\ x_1 \\ x_2 \end{matrix} \begin{matrix} \begin{matrix} z_1 & z_2 & z_3 \end{matrix} \\ \begin{bmatrix} 0.6 & 0.5 & 0.3 \\ 0.8 & 0.4 & 0.7 \end{bmatrix} \end{matrix}$$

The calculations for obtaining $\underset{\sim}{T}$ are as follows:

$$\mu_{\underset{\sim}{T}}(x_1, z_1) = \max\{\min[\mu_{\underset{\sim}{R}}(x_1, y_1), \mu_{\underset{\sim}{S}}(y_1, z_1)], \min[\mu_{\underset{\sim}{R}}(x_1, y_2), \mu_{\underset{\sim}{S}}(y_2, z_1)]\} = \max[\min(0.6, 1), \min(0.3, 0.8)] = \max(0.6, 0.3) = 0.6$$
$$\mu_{\underset{\sim}{T}}(x_1, z_2) = \max[\min(0.6, 0.5), \min(0.3, 0.4)] = \max(0.5, 0.3) = 0.5$$
$$\mu_{\underset{\sim}{T}}(x_1, z_3) = \max[\min(0.6, 0.3), \min(0.3, 0.7)] = \max(0.3, 0.3) = 0.3$$
$$\mu_{\underset{\sim}{T}}(x_2, z_1) = \max[\min(0.2, 1), \min(0.9, 0.8)] = \max(0.2, 0.8) = 0.8$$
$$\mu_{\underset{\sim}{T}}(x_2, z_2) = \max[\min(0.2, 0.5), \min(0.9, 0.4)] = \max(0.2, 0.4) = 0.4$$
$$\mu_{\underset{\sim}{T}}(x_2, z_3) = \max[\min(0.2, 0.3), \min(0.9, 0.7)] = \max(0.2, 0.7) = 0.7$$

(b) Max-product composition

$$\underset{\sim}{T} = \underset{\sim}{R} \bullet \underset{\sim}{S}$$

Calculations for $\underset{\sim}{T}$ are as follows:

$$\mu_{\underset{\sim}{T}}(x_1, z_1) = \max\{[\mu_{\underset{\sim}{R}}(x_1, y_1) \bullet \mu_{\underset{\sim}{S}}(y_1, z_1)], [\mu_{\underset{\sim}{R}}(x_1, y_2) \bullet \mu_{\underset{\sim}{S}}(y_2, z_1)]\} = \max(0.6, 0.24) = 0.6$$

$$\mu_T(x_1, z_2) = \max[(0.6 \times 0.5), (0.3 \times 0.4)]$$
$$= \max(0.3, 0.12) = 0.3$$
$$\mu_T(x_1, z_3) = \max[(0.6 \times 0.3), (0.3 \times 0.7)]$$
$$= \max(0.18, 0.21) = 0.21$$
$$\mu_T(x_2, z_1) = \max[(0.2 \times 1), (0.9 \times 0.8)]$$
$$= \max(0.2, 0.72) = 0.72$$
$$\mu_T(x_2, z_2) = \max[(0.2 \times 0.5), (0.9 \times 0.4)]$$
$$= \max(0.1, 0.36) = 0.36$$
$$\mu_T(x_2, z_3) = \max[(0.2 \times 0.3), (0.9 \times 0.7)]$$
$$= \max(0.06, 0.63) = 0.63$$

The fuzzy relation T by max-product composition is given as

$$T = \begin{matrix} \\ x_1 \\ x_2 \end{matrix} \begin{matrix} z_1 & z_2 & z_3 \end{matrix} \begin{bmatrix} 0.6 & 0.3 & 0.21 \\ 0.72 & 0.36 & 0.63 \end{bmatrix}$$

4. For a speed control of DC motor, the membership functions of series resistance, armature current and speed are given as follows:

$$R_{\sim} = \left\{\frac{0.4}{30} + \frac{0.6}{60} + \frac{1.0}{100} + \frac{0.1}{120}\right\}$$

$$I_{a\sim} = \left\{\frac{0.2}{20} + \frac{0.3}{40} + \frac{0.6}{60} + \frac{0.8}{80} + \frac{1.0}{100} + \frac{0.2}{120}\right\}$$

$$N = \left\{\frac{0.35}{500} + \frac{0.67}{1000} + \frac{0.97}{1500} + \frac{0.25}{1800}\right\}$$

Compute relation T for relating series resistance to motor speed, i.e., $R_{\sim}$ to N. Perform max-min composition only.

Solution: For relating series resistance to motor-speed, i.e., $R_{\sim}$ to N, we have to perform the following operations – two fuzzy cross-products and one fuzzy composition (max-min):

$$R = R_{\sim} \times I_{a\sim}$$
$$S = I_{a\sim} \times N$$
$$T = R \circ S$$

Relation R is obtained as the Cartesian product of $R_{\sim}$ and $I_{a\sim}$, i.e.,

$$R = R_{\sim} \times I_{a\sim}$$

	20	40	60	80	100	120
30	0.2	0.3	0.4	0.4	0.4	0.2
60	0.2	0.3	0.6	0.6	0.6	0.2
100	0.2	0.3	0.6	0.8	1.0	0.2
120	0.1	0.1	0.1	0.1	0.1	0.1

Relation S is obtained as the Cartesian product of $I_{a\sim}$ and N, i.e.,

$$S = I_{a\sim} \times N =$$

	500	1000	1500	1800
20	0.2	0.2	0.2	0.2
40	0.3	0.3	0.3	0.25
60	0.35	0.6	0.6	0.25
80	0.35	0.67	0.8	0.25
100	0.35	0.67	0.97	0.25
120	0.2	0.2	0.2	0.2

Relation T is obtained as the composition between relations R and S, i.e.,

$$T = R \circ S =$$

	500	1000	1500	1800
30	0.35	0.4	0.4	0.25
60	0.35	0.6	0.6	0.25
100	0.35	0.67	0.97	0.25
120	0.1	0.1	0.1	0.1

5. Consider two fuzzy sets given by

$$A = \left\{\frac{1}{\text{low}} + \frac{0.2}{\text{medium}} + \frac{0.5}{\text{high}}\right\}$$

$$B = \left\{\frac{0.9}{\text{positive}} + \frac{0.4}{\text{zero}} + \frac{0.9}{\text{negative}}\right\}$$

(a) Find the fuzzy relation for the Cartesian product of A and B, i.e., $R = A \times B$.

(b) Introduce a fuzzy set C given by

$$C = \left\{\frac{0.1}{\text{low}} + \frac{0.2}{\text{medium}} + \frac{0.7}{\text{high}}\right\}$$

Find the relation between C and B using Cartesian product, i.e., find $S = C \times B$.

(c) Find $C \circ R$ using max-min composition.

(d) Find $C \circ S$ using max-min composition.

Solution:

(a) The Cartesian product between A and B is obtained as

$$R = A \times B = \min[\mu_A(x), \mu_B(y)]$$

	positive	zero	negative
low	0.9	0.4	0.9
= medium	0.2	0.2	0.2
high	0.5	0.4	0.5

(b) The new fuzzy set is

$$C = \left\{\frac{0.1}{\text{low}} + \frac{0.2}{\text{medium}} + \frac{0.7}{\text{high}}\right\}$$

The Cartesian product between C and B is obtained as

$$S = C \times B = \min[\mu_C(x), \mu_B(y)]$$

	positive	zero	negative
low	0.1	0.1	0.1
= medium	0.2	0.2	0.2
high	0.7	0.4	0.7

(c)

$$C \circ R = [0.1 \quad 0.2 \quad 0.7]_{1\times3} \begin{bmatrix} 0.9 & 0.4 & 0.9 \\ 0.2 & 0.2 & 0.2 \\ 0.5 & 0.4 & 0.5 \end{bmatrix}_{3\times3}$$

$$= [0.5 \quad 0.4 \quad 0.5]$$

For instance,

$$\mu_{C\circ R}(x_1, y_1) = \max[\min(0.1, 0.9), \min(0.2, 0.2), \min(0.7, 0.5)]$$
$$= \max(0.1, 0.2, 0.5) = 0.5$$

(d)

$$C \circ S = [0.1 \quad 0.2 \quad 0.7]_{1\times3} \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.2 & 0.2 & 0.2 \\ 0.7 & 0.4 & 0.7 \end{bmatrix}_{3\times3}$$

$$= [0.7 \quad 0.4 \quad 0.7]$$

Hence max-min composition was used to find the relations.

6. Consider a universe of aircraft speed near the speed of sound as $X = \{0.72, 0.725, 0.75, 0.775, 0.78\}$ and a fuzzy set on this universe for the speed "near mach 0.75" = M where

$$M = \left\{\frac{0}{0.72} + \frac{0.8}{0.725} + \frac{1}{0.75} + \frac{0.8}{0.775} + \frac{0}{0.78}\right\}$$

Define a universe of altitudes as $Y = \{21, 22, 23, 24, 25, 26, 27\}$ in k-feet and a fuzzy set on this universe for the altitude fuzzy set "approximately 24,000 feet" = N where

$$N = \left\{\frac{0}{21k} + \frac{0.2}{22k} + \frac{0.7}{23k} + \frac{1}{24k} + \frac{0.7}{25k} + \frac{0.2}{26k} + \frac{0}{27k}\right\}$$

(a) Construct a relation $R = M \times N$

(b) For another aircraft speed, say M_1, in the region of mach 0.75 where

$$M_1 = \left\{\frac{0}{0.72} + \frac{0.8}{0.725} + \frac{1}{0.75} + \frac{0.6}{0.775} + \frac{0}{0.78}\right\}$$

find relation $S = M_1 \circ R$ using max-min composition.

Solution: The two given fuzzy sets are

$$M = \left\{\frac{0}{0.72} + \frac{0.8}{0.725} + \frac{1}{0.75} + \frac{0.8}{0.775} + \frac{0}{0.78}\right\}$$

$$N = \left\{\frac{0}{21k} + \frac{0.2}{22k} + \frac{0.7}{23k} + \frac{1}{24k} + \frac{0.7}{25k} + \frac{0.2}{26k} + \frac{0}{27k}\right\}$$

(a) Relation $\underset{\sim}{R} = \underset{\sim}{M} \times \underset{\sim}{N}$ is obtained by using Cartesian product

$$\underset{\sim}{R} = \min[\mu_{\underset{\sim}{M}}(x), \mu_{\underset{\sim}{N}}(y)]$$

$$= \begin{array}{c} \\ 0.72 \\ 0.725 \\ 0.75 \\ 0.775 \\ 0.78 \end{array} \begin{array}{c} \begin{array}{ccccccc} 21k & 22k & 23k & 24k & 25k & 26k & 27k \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.7 & 0.8 & 0.7 & 0.2 & 0 \\ 0 & 0.2 & 0.7 & 1 & 0.7 & 0.2 & 0 \\ 0 & 0.2 & 0.7 & 0.8 & 0.7 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

(b) Relation $\underset{\sim}{S} = \underset{\sim}{M}_1 \circ \underset{\sim}{R}$ is found by using max-min composition

$$\underset{\sim}{S} = \max\{\min[\mu_{\underset{\sim}{M}}(x), \mu_{\underset{\sim}{R}}(x,y)]\}$$

$$= [0 \quad 0.8 \quad 1 \quad 0.6 \quad 0]_{1\times 5}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.2 & 0.7 & 0.8 & 0.7 & 0.2 & 0 \\ 0 & 0.2 & 0.7 & 1 & 0.7 & 0.2 & 0 \\ 0 & 0.2 & 0.7 & 0.8 & 0.7 & 0.2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}_{5\times 7}$$

$$\underset{\sim}{S} = [0 \quad 0.2 \quad 0.7 \quad 1 \quad 0.7 \quad 0.2 \quad 0]_{1\times 7}$$

7. Consider two relations

$$\underset{\sim}{R} = \begin{array}{c} \\ 9 \\ 18 \\ 27 \\ 36 \end{array} \begin{array}{c} \begin{array}{ccccc} -100 & -50 & 0 & 50 & 100 \end{array} \\ \begin{bmatrix} 0.2 & 0.5 & 0.7 & 1 & 0.9 \\ 0.3 & 0.5 & 0.7 & 1 & 0.8 \\ 0.4 & 0.6 & 0.8 & 0.9 & 0.4 \\ 0.9 & 1 & 0.8 & 0.6 & 0.4 \end{bmatrix} \end{array}$$

and

$$\underset{\sim}{S} = \begin{array}{c} \\ -100 \\ -50 \\ 0 \\ 50 \\ 100 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 4 & 8 & 16 & 20 \end{array} \\ \begin{bmatrix} 1 & 0.8 & 0.6 & 0.3 & 0.1 \\ 0.7 & 1 & 0.7 & 0.5 & 0.4 \\ 0.5 & 0.6 & 1 & 0.8 & 0.8 \\ 0.3 & 0.4 & 0.6 & 1 & 0.9 \\ 0.9 & 0.3 & 0.5 & 0.7 & 1 \end{bmatrix} \end{array}$$

If $\underset{\sim}{R}$ is a relationship between frequency and temperature and $\underset{\sim}{S}$ represents a relation between temperature and reliability index of a circuit, obtain the relation between frequency and reliability index using (a) max-min composition and (b) max-product composition.

Solution:

(a) Max-min composition is performed as follows.

$$\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S} = \max\{\min[\mu_{\underset{\sim}{R}}(x,y), \mu_{\underset{\sim}{S}}(x,y)]\}.$$

$$= \begin{array}{c} \\ 9 \\ 18 \\ 27 \\ 36 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 4 & 8 & 16 & 20 \end{array} \\ \begin{bmatrix} 0.9 & 0.6 & 0.7 & 1 & 0.9 \\ 0.8 & 0.6 & 0.7 & 1 & 0.9 \\ 0.6 & 0.6 & 0.8 & 0.9 & 0.9 \\ 0.9 & 1 & 0.8 & 0.8 & 0.8 \end{bmatrix} \end{array}$$

(b) Max-product composition is performed as follows.

$$\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S} = \max\{\min[\mu_{\underset{\sim}{R}}(x,y) \times \mu_{\underset{\sim}{S}}(x,y)]\}$$

$$= \begin{array}{c} \\ 9 \\ 18 \\ 27 \\ 36 \end{array} \begin{array}{c} \begin{array}{ccccc} 2 & 4 & 8 & 16 & 20 \end{array} \\ \begin{bmatrix} 0.81 & 0.5 & 0.7 & 1.0 & 0.9 \\ 0.72 & 0.5 & 0.7 & 1.0 & 0.9 \\ 0.4 & 0.6 & 0.8 & 0.9 & 0.81 \\ 0.9 & 1.0 & 0.8 & 0.64 & 0.64 \end{bmatrix} \end{array}$$

Thus the relation between frequency and reliability index has been found using composition techniques.

8. Three fuzzy sets are given as follows:

$$\underset{\sim}{P} = \left\{\frac{0.1}{2} + \frac{0.3}{4} + \frac{0.7}{6} + \frac{0.4}{8} + \frac{0.2}{10}\right\}$$

$$\underset{\sim}{Q} = \left\{\frac{0.1}{0.1} + \frac{0.3}{0.2} + \frac{0.3}{0.3} + \frac{0.4}{0.4} + \frac{0.5}{0.5} + \frac{0.2}{0.6}\right\}$$

$$\underset{\sim}{T} = \left\{\frac{0.1}{0} + \frac{0.7}{0.5} + \frac{0.3}{1}\right\}$$

The following operations are performed over the fuzzy sets:

(a) $\underset{\sim}{R} = \underset{\sim}{P} \times \underset{\sim}{Q} = \min[\mu_{\underset{\sim}{P}}(x), \mu_{\underset{\sim}{Q}}(y)]$

$$= \begin{array}{c} \\ 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{array} \begin{array}{c} \begin{array}{cccccc} 0.1 & 0.2 & 0.3 & 0.4 & 0.5 & 0.6 \end{array} \\ \begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.3 & 0.3 & 0.3 & 0.2 \\ 0.1 & 0.3 & 0.3 & 0.4 & 0.5 & 0.2 \\ 0.1 & 0.3 & 0.3 & 0.4 & 0.4 & 0.2 \\ 0.1 & 0.2 & 0.2 & 0.2 & 0.2 & 0.2 \end{bmatrix} \end{array}$$

(b) $\underset{\sim}{S} = \underset{\sim}{Q} \times \underset{\sim}{T} = \min[\mu_{\underset{\sim}{Q}}(x), \mu_{\underset{\sim}{T}}(y)]$

$$= \begin{array}{c} \\ 0.1 \\ 0.2 \\ 0.3 \\ 0.4 \\ 0.5 \\ 0.6 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 0.5 & 1 \end{array} \\ \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.3 \\ 0.1 & 0.3 & 0.3 \\ 0.1 & 0.4 & 0.3 \\ 0.1 & 0.5 & 0.3 \\ 0.1 & 0.2 & 0.2 \end{bmatrix} \end{array}$$

(c) $\underset{\sim}{M} = \underset{\sim}{R} \circ \underset{\sim}{S} = \max[\min[\mu_{\underset{\sim}{R}}(x,y), \mu_{\underset{\sim}{S}}(x,y)]]$

$$= \begin{array}{c} \\ 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 0.5 & 1 \end{array} \\ \begin{bmatrix} 0.1 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.3 \\ 0.1 & 0.5 & 0.3 \\ 0.1 & 0.4 & 0.3 \\ 0.1 & 0.2 & 0.2 \end{bmatrix} \end{array}$$

(d) $\underset{\sim}{M} = \underset{\sim}{R} \circ \underset{\sim}{S} = \max[\mu_{\underset{\sim}{R}}(x,y) \times \mu_{\underset{\sim}{S}}(x,y)]$

$$= \begin{array}{c} \\ 2 \\ 4 \\ 6 \\ 8 \\ 10 \end{array} \begin{array}{c} \begin{array}{ccc} 0 & 0.5 & 1 \end{array} \\ \begin{bmatrix} 0.01 & 0.05 & 0.03 \\ 0.03 & 0.05 & 0.09 \\ 0.05 & 0.25 & 0.15 \\ 0.04 & 0.20 & 0.12 \\ 0.02 & 0.0 & 0.06 \end{bmatrix} \end{array}$$

Thus the operations were performed over the given fuzzy sets.

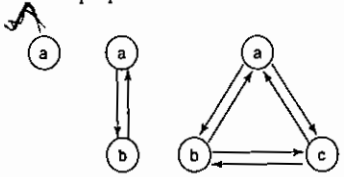
9. Which of the following are equivalence relations?

No.	Set	Relation on the set
(i)	People	is the brother of
(ii)	People	has the same parents as
(iii)	Points on a map	is connected by a road to
(iv)	Lines in plane geometry	is perpendicular to
(v)	Positive integers	for some integer k, equals 10^k times

Draw graphs of the equivalence relations.

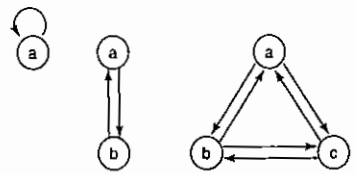
Solution:

(a) The set is people. The relation of the set "is the brother of." The relation (figure below) is not equivalence relation because people considered cannot be brothers to themselves. So, reflexive property is not satisfied. But symmetry and transitive properties are satisfied.



The figure illustrates that the relation is not an equivalence relation.

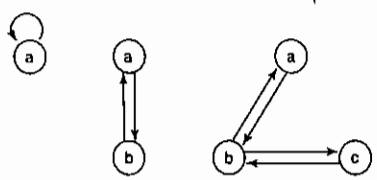
(b) The set is people. The relation is "has the same parents as." In this case (figure below), all the three properties are satisfied, hence it is an equivalence relation.



Thus the relation is an equivalence relation.

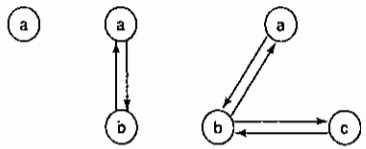
(c) The set is "points on a map." The relation is "is connected by a road to." This relation (figure on next page) is not an equivalence relation because the transitive property is not satisfied. The road may connect 1st point and 2nd point; 2nd point

and 3rd point; but it may not connect 1st and 3rd points. Thus, transitive property is not satisfied.



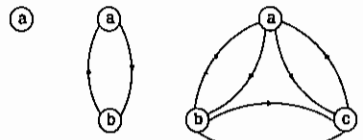
The figure illustrates that the relation is not an equivalence relation.

(d) The set is "lines in plane geometry." The relation "is perpendicular to." The relation (figure below) defined here is not an equivalence relation because both reflexive and transitive properties are not satisfied. A line cannot be perpendicular to itself, hence reflexivity is not satisfied. Also transitivity property is not satisfied because 1st line and 2nd line may be perpendicular to each other, 2nd line and 3rd line may also be perpendicular to each other, but 1st line and 3rd line will not be perpendicular to each other. However, symmetry property is satisfied.



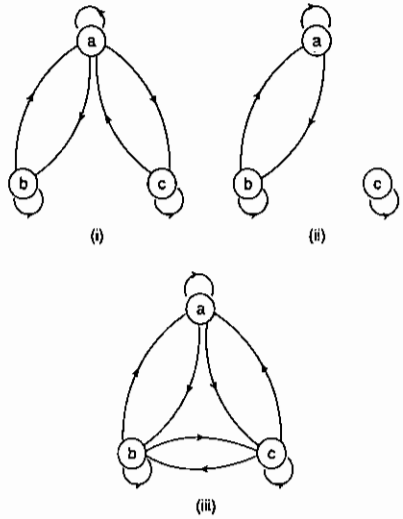
The figure illustrates that the relation is not an equivalence relation.

(e) The set is "positive integers". The relation is "for some integer k, equals 10^k times." In this case (figure below), reflexivity is not satisfied because a positive integer, for some integer k, equals 10^k times is not possible. Symmetry and transitivity properties are satisfied. Thus, the relation is not an equivalence relation.



The figure illustrates that the relation is not an equivalence relation.

10. The following figure shows three relations on the universe $X = \{a, b, c\}$. Are these relations equivalence relations?



Solution:

(a) The relation in (i) is not equivalence relation because transitive property is not satisfied.

(b) The relation in (ii) is not equivalence relation because transitive property is not satisfied.

(c) The relation in (iii) is equivalence relation because reflexive, symmetry and transitive properties are satisfied.

8.9 Review Questions

1. Define classical relations and fuzzy relations.
2. State the Cartesian product of a relation.
3. How are the relations represented in various forms?
4. What is one-ore mapping of a relation?

5. Compare constrained relation and non-constrained relation.
6. Give the cardinality of classical relation.
7. Mention the operations performed on classical relations.
8. List the various properties of crisp relations.
9. What is the necessity of composition of a relation?
10. What are the various types of composition techniques?
11. Define fuzzy matrix and fuzzy graph.
12. Give the cardinality of fuzzy relation.
13. Explain the operations and properties over a fuzzy relation.
14. Discuss fuzzy composition techniques.
15. What are tolerance and equivalence relations?
16. Describe in detail classical equivalence relation.
17. Write short note on fuzzy equivalence relation.
18. How are a crisp tolerance relation and a fuzzy tolerance relation converted to crisp equivalence relation and fuzzy equivalence relation respectively?
19. Explain with suitable diagrams and examples fuzzy equivalence relation.
20. What is meant by non-interactive fuzzy sets?

8.10 Exercise Problems

1. The elements in two sets X and Y are given as $X = \{1, 2, 3\}, Y = \{p, q, r\}$. Find the various Cartesian products of these two sets.
2. For the fuzzy sets given

$$\underset{\sim}{A} = \left\{\frac{0.5}{x_1} + \frac{0.2}{x_2} + \frac{0.9}{x_3}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{1}{y_1} + \frac{0.5}{y_2} + \frac{1}{y_3}\right\}$$

find relation $\underset{\sim}{R}$ by performing Cartesian product over the given fuzzy sets.

3. The fuzzy relations are given as

$$\underset{\sim}{R} = \begin{matrix} \\ x_1 \\ x_2 \end{matrix}\begin{matrix} \begin{matrix} y_1 & y_2 & y_3 \end{matrix} \\ \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \end{bmatrix} \end{matrix}; \quad \underset{\sim}{S} = \begin{matrix} \\ y_1 \\ y_2 \\ y_3 \end{matrix}\begin{matrix} \begin{matrix} z_1 & z_2 \end{matrix} \\ \begin{bmatrix} 0.8 & 0.1 \\ 0.6 & 0.9 \\ 0.4 & 1.0 \end{bmatrix} \end{matrix}$$

Perform composition over the two given fuzzy relations and obtain a fuzzy relation $\underset{\sim}{T}$.

4. Three fuzzy sets are defined as follows:

$$\underset{\sim}{A} = \left\{\frac{0.1}{30} + \frac{0.2}{60} + \frac{0.3}{90} + \frac{0.4}{120}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{1}{1} + \frac{0.2}{2} + \frac{0.5}{3} + \frac{0.7}{4} + \frac{0.3}{5} + \frac{0}{6}\right\}$$

$$\underset{\sim}{C} = \left\{\frac{0.33}{100} + \frac{0.65}{200} + \frac{0.92}{300} + \frac{0.21}{400}\right\}$$

Find the following:

(a) $\underset{\sim}{R} = \underset{\sim}{A} \times \underset{\sim}{B}$

(b) $\underset{\sim}{S} = \underset{\sim}{B} \times \underset{\sim}{C}$

(c) $\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S}$ using max-min composition

(d) $\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S}$ using max-product composition

5. For two fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{0.2}{\text{LS}} + \frac{0.5}{\text{MS}} + \frac{0.7}{\text{HS}}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{0.1}{\text{PE}} + \frac{0.55}{\text{ZE}} + \frac{0.85}{\text{NE}}\right\}$$

(a) Find $\underset{\sim}{R} = \underset{\sim}{A} \times \underset{\sim}{B}$

(b) Introducing a fuzzy set $\underset{\sim}{C}$ given by

$$\underset{\sim}{C} = \left\{\frac{0.25}{\text{LS}} + \frac{0.5}{\text{MS}} + \frac{0.75}{\text{HS}}\right\}$$

Find $\underset{\sim}{S} = \underset{\sim}{B} \times \underset{\sim}{C}$.

(c) Find $\underset{\sim}{C} \circ \underset{\sim}{S}$ using max-min composition.

(d) Find $\underset{\sim}{C} \circ \underset{\sim}{R}$ using max-min composition.

(e) Find $\underset{\sim}{C} \circ \underset{\sim}{R}$ using max-product composition.

6. Three elements for a medicinal research are defined as

$$\underset{\sim}{D} = \left\{\frac{0.3}{0} + \frac{0.7}{1} + \frac{1}{2}\right\}$$

$$\underset{\sim}{I} = \left\{\frac{0.5}{20} + \frac{0.75}{30} + \frac{0.6}{40}\right\}$$

$$\underset{\sim}{V} = \left\{\frac{0.7}{20} + \frac{0.8}{30} + \frac{0.5}{40}\right\}$$

Based on these membership functions, find the following:

(a) $\underset{\sim}{R} = \underset{\sim}{D} \times \underset{\sim}{I}$

(b) Max-min composition of $\underset{\sim}{V} \circ \underset{\sim}{R}$.

(c) Max-product composition of $\underset{\sim}{V} \circ \underset{\sim}{R}$.

7. An athletic race was conducted. The following membership functions are defined based the speed of athletes:

$$\underset{\sim}{\text{Low}} = \left\{\frac{0}{100} + \frac{0.1}{200} + \frac{0.3}{300}\right\}$$

$$\underset{\sim}{\text{Medium}} = \left\{\frac{0.5}{100} + \frac{0.57}{200} + \frac{0.6}{300}\right\}$$

$$\underset{\sim}{\text{High}} = \left\{\frac{0.8}{100} + \frac{0.9}{200} + \frac{1.0}{300}\right\}$$

Find the following:

(a) $\underset{\sim}{R} = \underset{\sim}{\text{Low}} \times \underset{\sim}{\text{Medium}}$

(b) $\underset{\sim}{S} = \underset{\sim}{\text{Medium}} \times \underset{\sim}{\text{High}}$

(c) $\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S}$ using max-min composition.

(d) $\underset{\sim}{T} = \underset{\sim}{R} \circ \underset{\sim}{S}$ using max-product composition.

8. Two relations are defined as

$$\underset{\sim}{R} = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cccc} B_1 & B_2 & B_3 & B_4 \end{array} \\ \begin{bmatrix} 1 & 0.2 & 0.3 & 0 \\ 0.2 & 0.4 & 0.5 & 0.6 \\ 0.3 & 0.4 & 0.6 & 0.9 \\ 0 & 0.2 & 0.9 & 1 \end{bmatrix} \end{array}$$

$$\underset{\sim}{S} = \begin{array}{c} \\ A_1 \\ A_2 \\ A_3 \\ A_4 \end{array} \begin{array}{c} \begin{array}{cc} V_1 & V_2 \end{array} \\ \begin{bmatrix} 1 & 0.5 \\ 1 & 0.5 \\ 0.5 & 1 \\ 0.5 & 1 \end{bmatrix} \end{array}$$

Find relation, $\underset{\sim}{T} = \underset{\sim}{R}^{T} \circ \underset{\sim}{S}$ using

(a) Max-min composition

(b) Max-product composition

9. Two relations are given as

$$\underset{\sim}{A} = \begin{bmatrix} 0.8 & 1 & 0.5 & 0.1 & 0 & 0 \\ 0.1 & 0.2 & 0.3 & 0.2 & 1 & 0 \\ 0.1 & 0.6 & 0.2 & 0.7 & 1 & 0 \\ 0.1 & 0.4 & 0.5 & 0.8 & 1 & 0.9 \end{bmatrix}$$

$$\underset{\sim}{B} = \begin{bmatrix} 0.1 & 0.2 & 0.5 & 0.9 & 0 \\ 0.1 & 0.2 & 0.5 & 0.9 & 0 \\ 0.1 & 0.2 & 0.5 & 0.9 & 0 \\ 0.3 & 0.4 & 0.7 & 0.6 & 1 \\ 0.3 & 0.4 & 0.7 & 0.6 & 1 \\ 0.3 & 0.4 & 0.7 & 0.1 & 1 \end{bmatrix}$$

Find the relation $\underset{\sim}{A} \circ \underset{\sim}{B}$, using

(a) Max-min composition

(b) Max-product composition

10. Which of the following are equivalence relations?

No.	Set	Relation on the set
(i)	People	is the sister of
(ii)	People	has the same grandparents as
(iii)	Lines in plane geometry	is parallel to
(iv)	Positive integer	For some integer k, equals e^{-k} times
(v)	Points on a map	Is connected by a rail to

Draw graphs of the equivalence relations with appropriate labels on the vertices.

9 Membership Functions

Learning Objectives

- Scope of membership functions.
- About fuzzification process.
- How membership functions are used to define the fuzziness existing in the fuzzy set.
- Different types of fuzzification processes.
- Determination of fuzzy membership functions using neural networks and genetic algorithms.
- Classifications of fuzzy sets.

9.1 Introduction

Membership function defines the fuzziness in a fuzzy set irrespective of the elements in the set, which are discrete or continuous. The membership functions are generally represented in graphical form. There exist certain limitations for the shapes used to represent graphical form of membership function. The rules that describe fuzziness graphically are also fuzzy. But standard shapes of the membership functions are maintained over the years. Fuzzy membership functions are determined in practical problem by the opinion of experts. Membership function can be thought of as a technique to solve empirical problems on the basis of experience rather than knowledge. Available histograms and other probability information can also help in constructing the membership function. There are several ways to characterize fuzziness; in a similar way, there are several ways to graphically construct a membership function that describes fuzziness. In this chapter few possibilities of describing membership functions are dealt with. Also few methodologies have been discussed to build these membership functions.

9.2 Features of the Membership Functions

The membership function defines all the information contained in a fuzzy set; hence it is important to discuss the various features of the membership functions. A fuzzy set $\underset{\sim}{A}$ in the universe of discourse X can be defined as a set of ordered pairs:

$$\underset{\sim}{A} = \{(x, \mu_{\underset{\sim}{A}}(x)) \mid x \in X\}$$

where $\mu_{\underset{\sim}{A}}(\cdot)$ is called membership function of $\underset{\sim}{A}$. The membership function $\mu_{\underset{\sim}{A}}(\cdot)$ maps X to the membership space M, i.e., $\mu_{\underset{\sim}{A}} : X \to M$. The membership value ranges in the interval [0, 1], i.e., the range of the membership function is a subset of the non-negative real numbers whose supremum is finite.

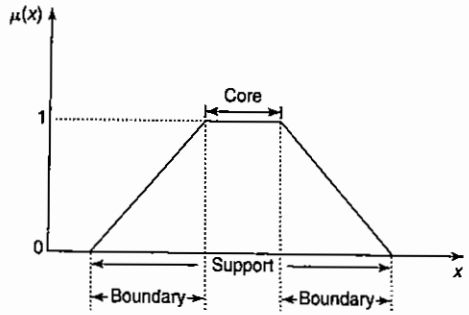


Figure 9-1 Features of membership functions.

Figure 9-1 shows the basic features of the membership functions. The three main basic features involved in characterizing membership function are the following.

1. *Core*: The core of a membership function for some fuzzy set $\underset{\sim}{A}$ is defined as that region of universe that is characterized by complete membership in the set $\underset{\sim}{A}$. The core has elements x of the universe such that

$$\mu_{\underset{\sim}{A}}(x) = 1$$

The core of a fuzzy set may be an empty set.

2. *Support*: The support of a membership function for a fuzzy set $\underset{\sim}{A}$ is defined as that region of universe that is characterized by a nonzero membership in the set $\underset{\sim}{A}$. The support comprises elements x of the universe such that

$$\mu_{\underset{\sim}{A}}(x) > 0$$

A fuzzy set whose support is a single element in X with $\mu_{\underset{\sim}{A}}(x) = 1$ is referred to as a fuzzy singleton.

3. *Boundary*: The support of a membership function for a fuzzy set $\underset{\sim}{A}$ is defined as that region of universe containing elements that have a nonzero but not complete membership. The boundary comprises those elements of x of the universe such that

$$0 < \mu_{\underset{\sim}{A}}(x) < 1$$

The boundary elements are those which possess partial membership in the fuzzy set $\underset{\sim}{A}$.

The core, support and boundary are the three main features of a fuzzy set membership function. There are various other types of fuzzy sets, of which a few are discussed below.

A fuzzy set whose membership function has at least one element x in the universe whose membership value is unity is called *normal fuzzy set*. The element for which the membership is equal to 1 is called prototypical element. A fuzzy set wherein no membership function has its value equal to 1 is called *subnormal fuzzy set*. The normal and subnormal fuzzy sets are shown in Figures 9-2(A) and (B), respectively.

A *convex fuzzy set* has a membership function whose membership values are strictly monotonically increasing or strictly monotonically decreasing or strictly monotonically increasing than strictly monotonically decreasing with increasing values for elements in the universe. A fuzzy set possessing characteristics opposite to that of convex fuzzy set is called *nonconvex fuzzy set*, i.e., the membership values of the membership function

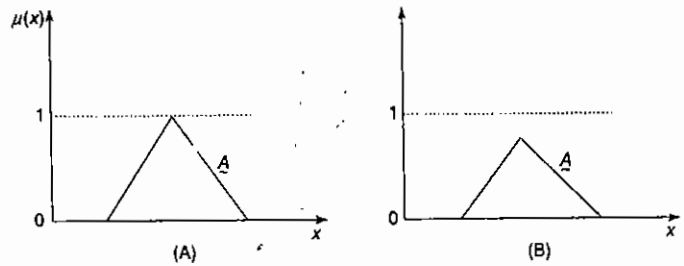


Figure 9-2 (A) Normal fuzzy set and (B) subnormal fuzzy set.

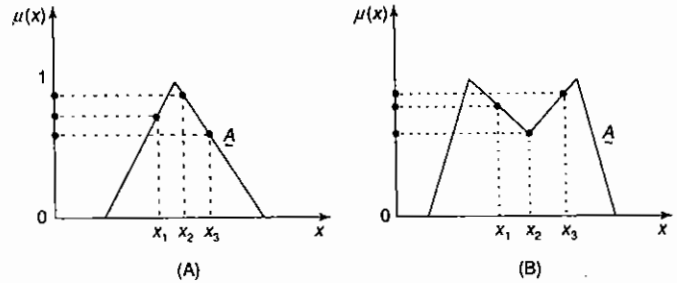


Figure 9-3 (A) Convex normal fuzzy set and (B) nonconvex normal fuzzy set.

are not strictly monotonically increasing or decreasing or strictly monotonically increasing than decreasing. The convex and nonconvex normal fuzzy sets are shown in Figures 9-3(A) and (B), respectively.

From Figure 9-3(A), the convex normal fuzzy set can be defined in the following way. For elements x_1, x_2 and x_3 in a fuzzy set $\underset{\sim}{A}$, if the following relation between x_1, x_2 and x_3 holds, i.e.,

$$\mu_{\underset{\sim}{A}}(x_2) \geq \min [\mu_{\underset{\sim}{A}}(x_1), \mu_{\underset{\sim}{A}}(x_3)]$$

then $\underset{\sim}{A}$ is said to be a convex fuzzy set. The membership of the element x_2 should be greater than or equal to the membership of elements x_1 and x_3. For a nonconvex fuzzy set, the constraint is not satisfied.

$$\mu_{\underset{\sim}{A}}(x_2) \geq \min [\mu_{\underset{\sim}{A}}(x_1), \mu_{\underset{\sim}{A}}(x_3)]$$

The intersection of two convex fuzzy sets is also a convex fuzzy set. The element in the universe for which a particular fuzzy set $\underset{\sim}{A}$ has its value equal to 0.5 is called crossover point of a membership function. The membership value of a crossover point of a fuzzy set is equal to 0.5, i.e., $\mu_{\underset{\sim}{A}}(x) = 0.5$. It is shown in Figure 9-4. There can be more than one crossover point in a fuzzy set.

The maximum value of the membership function in a fuzzy set $\underset{\sim}{A}$ is called as the height of the fuzzy set. For a normal fuzzy set, the height is equal to 1, because the maximum value of the membership function allowed is 1. Thus, if the height of a fuzzy set is less than 1, then the fuzzy set is called subnormal fuzzy set. When the fuzzy set $\underset{\sim}{A}$ is a convex single-point normal fuzzy set defined on the real time, then $\underset{\sim}{A}$ is termed as a fuzzy number.

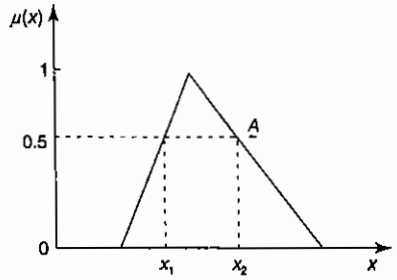


Figure 9-4 Crossover point of a fuzzy set.

9.3 Fuzzification

Fuzzification is the process of transforming a crisp set to a fuzzy set or a fuzzy set to a fuzzier set, i.e., crisp quantities are converted to fuzzy quantities. This operation translates accurate crisp input values into linguistic variables. In real-life world, the quantities that we consider may be thought of as crisp, accurate and deterministic, but actually they are not so. They possess uncertainty within themselves. The uncertainty may arise due to vagueness, imprecision or uncertainty; in this case the variable is probably fuzzy and can be represented by a membership function. For example, when one is told that the temperature is 9 °C, the person translates the crisp input value into linguistic variable such as cold or warm according to ones knowledge and then makes a decision about the need to wear jacket or not. If one fails to fuzzify then it is not possible to continue the decision process or error decision may be reached.

For a fuzzy set $\underline{A} = \{\mu_i/x_i | x_i \in X\}$, a common fuzzification algorithm is performed by keeping μ_i constant and x_i being transformed to a fuzzy set $Q(x_i)$ depicting the expression about x_i. The fuzzy set $Q(x_i)$ is referred to as the *kernel of fuzzification*. The fuzzified set $\underline{A}$ can be expressed as

$$\underline{A} = \mu_1 Q(x_1) + \mu_2 Q(x_2) + \cdots + \mu_n Q(x_n)$$

where the symbol $\sim$ means fuzzified. This process of fuzzification is called support fuzzification (s-fuzzification). There is another method of fuzzification called *grade fuzzification* (g-fuzzification) where x_i is kept constant and μ_i is expressed as a fuzzy set. Thus, using these methods, fuzzification is carried out.

9.4 Methods of Membership Value Assignments

There are several ways to assign membership values to fuzzy variables in comparison with the probability density functions to random variables. The process of membership value assignment may be by intuition, logical reasoning, procedural method or algorithmic approach. The methods for assigning membership value are as follows:

1. Intuition;
2. inference;
3. rank ordering;
4. angular fuzzy sets;
5. neural networks;

6. genetic algorithm;
7. inductive reasoning.

These methods are discussed in detail in the following subsections. Apart from these methods, there are other methods such as soft partitioning, meta rules and fuzzy statistics, to name a few.

9.4.1 Intuition

Intuition method is based upon the common intelligence of human. It is the capacity of the human to develop membership functions on the basis of their own intelligence and understanding capability. There should be an in-depth knowledge of the application to which membership value assignment has to be made. Figure 9-5 shows various shapes of weights of people measured in kilogram in the universe. Each curve is a membership function corresponding to various fuzzy (linguistic) variables, such as vety light, light, normal, heavy and very heavy. The curves are based on context functions and the human developing them. For example, if the weights are referred to range of thin persons we get one set of curves, and if they are referred to range of normal weighing persons we get another set and so on. The main characteristics of these curves for their usage in fuzzy operations are based on their overlapping capacity.

9.4.2 Inference

The inference method uses knowledge to perform deductive reasoning. Deduction achieves conclusion by means of forward inference. There are various methods for performing deductive reasoning. Here the knowledge of geometrical shapes and geometry is used for defining membership values. The membership functions may be defined by various shapes: triangular, trapezoidal, bell-shaped, Gaussian and so on. The inference method here is discussed via triangular shape.

Consider a triangle, where X, Y and Z are the angles, such that $X \geq Y \geq Z \geq 0$, and let U be the universe of triangles, i.e.,

$$U = \{(X, Y, Z) | X \geq Y \geq Z \geq 0; X + Y + Z = 180\}$$

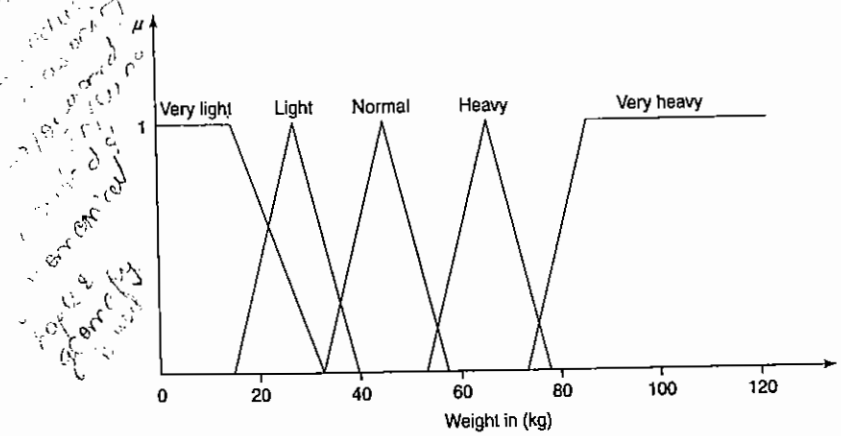


Figure 9-5 Membership functions for the fuzzy variable "weight."

There are various types of triangles available. Here a few are considered to explain inference methodology:

$\underline{I}$ = isosceles triangle (approximate)
$\underline{E}$ = equilateral triangle (approximate)
$\underline{R}$ = right-angle triangle (approximate)
$\underline{IR}$ = isosceles and right-angle triangle (approximate)
$\underline{T}$ = other triangles

By the method of inference, we can obtain the membership values for all the above-mentioned triangles, since we possess knowledge about geometry that helps us to make membership assignments.

The membership values of approximate isosceles triangle is obtained using the following definition, where $X \geq Y \geq Z \geq 0$ and $X + Y + Z = 180°$:

$$\mu_{\underline{I}}(X, Y, Z) = 1 - \frac{1}{60°} \min(X - Y, Y - Z)$$

If $X = Y$ or $Y = Z$; the membership value of approximate isosceles triangle is equal to 1. On the other hand, if $X = 120°$, $Y = 60°$ and $Z = 0°$, we get

$$\begin{aligned}\mu_{\underline{I}}(X, Y, Z) &= 1 - \frac{1}{60°} \min(120° - 60°, 60° - 0°) \\ &= 1 - \frac{1}{60°} \min(60°, 60°) \\ &= 1 - \frac{1}{60°} \times 60° \\ &= 1 - 1 = 0\end{aligned}$$

The membership value of approximate right-angle triangle is given by

$$\mu_{\underline{R}}(X, Y, Z) = 1 - \frac{1}{90°} |X - 90°|$$

If $X = 90°$, the membership value of a right-angle triangle is 1, and if $X = 180°$, the membership value $\mu_{\underline{R}}$ becomes 0:

$$X = 90° \Rightarrow \mu_{\underline{R}} = 1$$
$$X = 180° \Rightarrow \mu_{\underline{R}} = 0$$

The membership value of approximate isosceles right-angle triangle is obtained by taking the logical intersection of the approximate isosceles and approximate right-angle triangle membership functions, i.e.,

$$\underline{IR} = \underline{I} \cap \underline{R}$$

and it is given by

$$\begin{aligned}\mu_{\underline{IR}}(X, Y, Z) &= \min[\mu_{\underline{I}}(X, Y, Z), \mu_{\underline{R}}(X, Y, Z)] \\ &= 1 - \max\left[\frac{1}{60°} \min(X - Y, Y - Z), \frac{1}{90°} |X - 90°|\right]\end{aligned}$$

The membership function for a fuzzy equilateral triangle is given by

$$\mu_{\underline{E}}(X, Y, Z) = 1 - \frac{1}{180°} |X - Z|$$

The membership function of other triangles, denoted by $\underline{T}$, is the complement of the logical union of $\underline{I}$, $\underline{R}$ and $\underline{E}$, i.e.,

$$\underline{T} = \overline{\underline{I} \cup \underline{R} \cup \underline{E}}$$

By using De Morgan's law, we get

$$\underline{T} = \overline{\underline{I}} \cap \overline{\underline{R}} \cap \overline{\underline{E}}$$

The membership value can be obtained using the equation

$$\begin{aligned}\mu_{\underline{T}}(X, Y, Z) &= \min\{1 - \mu_{\underline{I}}(X, Y, Z), 1 - \mu_{\underline{E}}(X, Y, Z), 1 - \mu_{\underline{R}}(X, Y, Z)\} \\ &= \frac{1}{180°} \min\{3(X - Y), 3(Y - Z), 2|X - 90°|, X - Z\}\end{aligned}$$

The inference method as discussed for triangular shape can be extended for trapezoidal shape and so on, on the basis of knowledge of geometry.

9.4.3 Rank Ordering

The formation of government is based on the polling concept; to identify a best student, ranking may be performed; to buy a car, one can ask for several opinions and so on. All the above mentioned activities are carried out on the basis of the preferences made by an individual, a committee, a poll and other opinion methods. This methodology can be adapted to assign membership values to a fuzzy variable. Pairwise comparisons enable us to determine preferences and this results in determining the order of the membership. The in-depth details of rank ordering are included in Chapter 14.

9.4.4 Angular Fuzzy Sets

Coordinate description is the major difference between the angular fuzzy sets and standard fuzzy sets. Angular fuzzy sets are defined on a universe of angles, thus repeating the shapes every 2π cycles. The truth values of the linguistic variable are represented by angular fuzzy sets. The logical propositions are equated to the membership value "truth," as they are associated with the degree of truth. The certain proposition with membership value "1" is said to be true and that the proposition with membership value "0" is said to be false. The intermediate values between 0 and 1 correspond to a proposition being partially true or partially false.

The angular fuzzy sets are explained as follows: Consider the pH value of wastewater from a dyeing industry. These pH readings are assigned linguistic labels, such as high base, medium acid, etc., to understand the quality of the polluted water. The pH value should be taken care of because the waste from the dyeing industry should not be hazardous to the environment. As is known, the neutral solution has a pH value of 7. The linguistic variables are build in such a way that a "neutral (N)" solution corresponds to $\theta = 0$ rad, and "exact base (EB)" and "exact acid (EA)" corresponds to $\theta = \pi/2$ rad and $\theta = -\pi/2$ rad, respectively. The levels of pH between 7 and 14 can be termed as "very base" (VB), "medium base" (MB) and so on and are represented between 0 to $\pi/2$. Levels of pH between 0 and 7 can be termed as "very acid (VA)," "medium

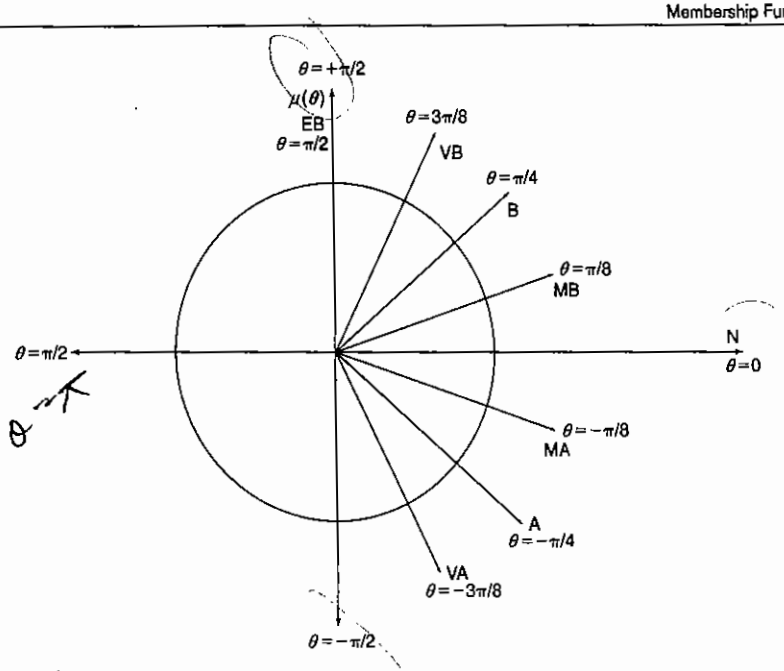


Figure 9-6 Model of angular fuzzy set.

acid (MA)" and so on and are represented between 0 rad and $-\pi/2$ rad. The model of the angular fuzzy set using these linguistic labels for pH is shown in Figure 9-6.

The values of the linguistic variables vary with "θ" and their membership values are on the $\mu(\theta)$ axis. The membership value corresponding to the linguistic term can be obtained from the equation

$$\mu_t(\theta) = t \cdot \tan(\theta)$$

where t is horizontal projection of radial vector. Angular fuzzy sets are best in cases with polar coordinates or in cases where the value of the variable is cyclic.

9.4.5 Neural Networks

The basic concepts of neural networks and various types of neural networks were discussed in detail in Chapters 2–6. The neural network can be used to obtain fuzzy membership values. Consider a case where fuzzy membership functions are to be created for fuzzy classes of an input data set. The input data set is collected and divided into training data set and testing data set. The training data set trains the neural network. Consider an input training data set as shown in Figure 9-7(A). The data set is found to contain several data points. The data points are first divided into different classes by conventional clustering techniques. In Figure 9-7(A), it can be noticed that the data points are divided into three classes, R_A, R_B and R_C. Consider data point 1 having input coordinate values $x_1 = 0.6$ and $x_2 = 0.8$. This data point lies in the region R_B; hence we assign complete membership of 1 to class R_B and of 0 to classes R_A and R_C. In a similar manner, the other data points are given membership values of 1 for the classes they initially belong. A neural network is created

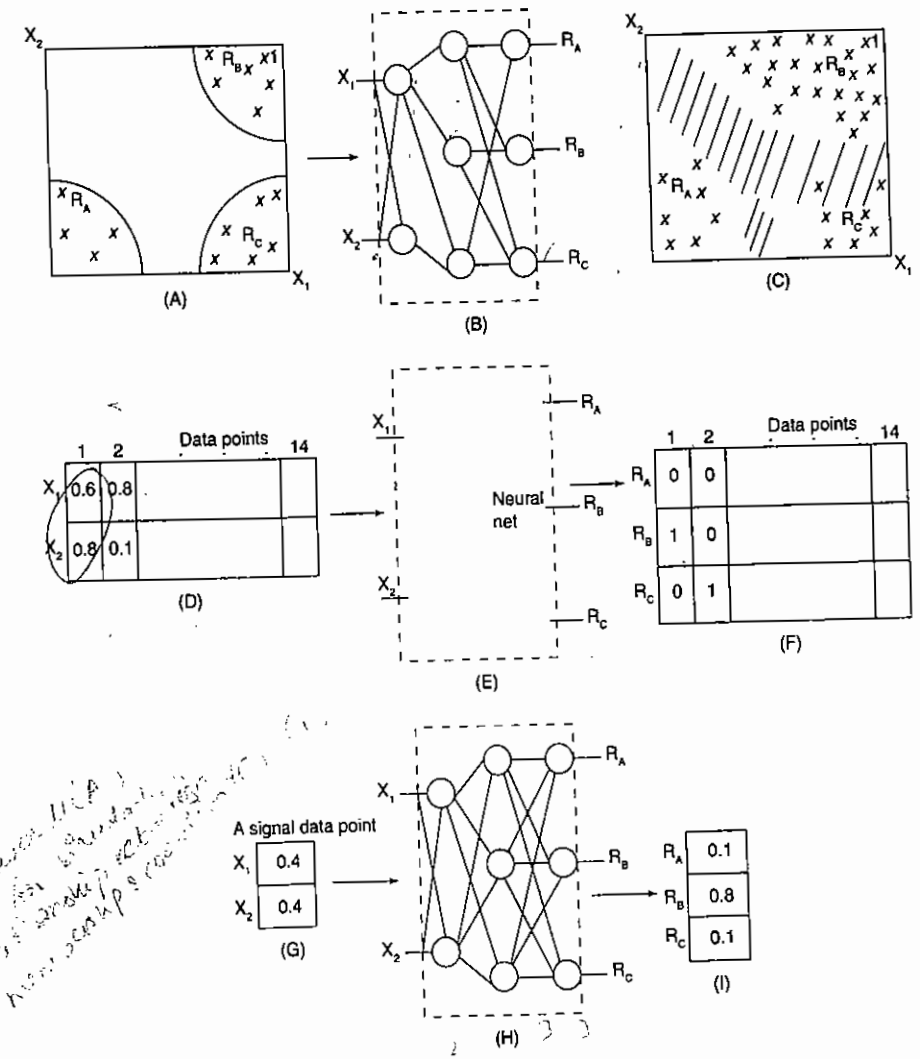


Figure 9-7 Fuzzy membership function evaluated from neural networks.

[Figures 9-7(B), (E), (H)] which uses the data point marked 1 and the corresponding membership values in different classes for training itself for simulating the relationship between coordinate locations and the membership values. The output of neural network is shown in Figure 9-7(C), which classifies data points into one of the three regions. The neural net then uses the next set of data values and membership values for further training process as can be seen in Figure 9-7(D). The process is continued until the neural network simulates the entire set of input–output values. The network performance is tested using the testing data set.

When the neural network is ready in its final version, it can be used to determine the membership values of any input data [Figure 9-7(G)] in the different regions (classes) [Figure 9-7(I)]. A complete mapping of the membership of various data points in various fuzzy classes can be derived to determine the overlap of the different classes. The overlap of the three fuzzy classes is shown in hatched portion of Figure 9-7(C). In this manner, neural network is used to determine the fuzzy membership functions.

9.4.6 Genetic Algorithms

Genetic algorithm is based on the Darwin's theory of evolution; the basic rule is "survival of the fittest." The genetic algorithm is used here to determine the fuzzy membership functions. This can be done using the following steps:

1. For a particular functional mapping system, the same membership functions and shapes are assumed for various fuzzy variables to be defined.
2. These chosen membership functions are then coded into bit strings.
3. Then these bit strings are concatenated together.
4. The fitness function to be used here is noted. In genetic algorithm, fitness function plays a major role similar to that played by activation function in neural network.
5. The fitness function is used to evaluate the fitness of each set of membership functions.
6. These membership functions define the functional mapping of the system.

The process of generating and evaluating strings is carried out until we get a convergence to the solution within a generation, i.e., we obtain the membership functions with best fitness value. Thus, fuzzy membership functions can be obtained from genetic algorithm.

9.4.7 Induction Reasoning

Induction is used to deduce causes by means of backward inference. The characteristics of inductive reasoning can be used to generate membership functions. Induction employs entropy minimization principle, which clusters the parameters corresponding to the output classes. To perform inductive reasoning method, a well-defined database for the input–output relationship should exist. The inductive reasoning can be applied for complex systems where the data are abundant and static. For dynamic data sets, this method is not best suited, because the membership functions continually changes with time. There exist three laws of induction (Christeuseu, 1980):

1. Given a set of irreducible outcomes of an experiment, the induced probabilities are those probabilities consistent with all available information that maximize the entropy of the set.
2. The induced probability of a set of independent observations is proportional to the probability density of the induced probability of a single observation.
3. The induced rule is that rule consistent with all available information of that minimizes the entropy.

The third law stated above is widely used for the development of membership functions. The membership functions using inductive reasoning are generated as follows:

1. A fuzzy threshold is to be established between classes of data.
2. Using entropy minimization screening method, first determine the threshold line.
3. Then start the segmentation process.
4. The segmentation process results into two classes.
5. Again partitioning the first two classes one more time, we obtain three different classes.
6. The partitioning is repeated with threshold value calculations, which lead us to partition the data set into a number of classes or fuzzy sets.
7. Then on the basis of the shape, membership function is determined.

Thus the membership function is generated on the basis of the partitioning or analog screening concept. This draws a threshold line between two classes of sample data. The idea behind drawing the threshold line is to classify the samples when minimizing the entropy for optimum partitioning.

9.5 Summary

Membership functions and their features are discussed in this chapter. Also, the different methods of obtaining the membership functions are dealt with. The formation of the membership function is the core for the entire fuzzy system operation. The capability of human reasoning is very important for membership functions. The inference method is based on the geometrical shapes and geometry, whereas the angular fuzzy set is based on the angular features. Using neural networks and reasoning methods the memberships are tuned in a cyclic fashion and are based on rule structure. The improvements are carried out to achieve an optimum solution using genetic algorithms. Thus, the membership function can be formed using any one of the methods discussed.

9.6 Solved Problems

1. Using your own intuition and definitions of the universe of discourse, plot fuzzy membership functions for "weight of people."

Solution: The universe of discourse is weight of people. Let the weights be in kg, i.e., kilogram. Let the linguistic variables be the following:

Very thin (VT) : $W \leq 25$
Thin (T) : $25 < W \leq 45$
Average (AV) : $45 < W \leq 60$
Stout (S) : $60 < W \leq 75$
Very stout (VS) : $W > 75$

Now plotting the defined linguistic variables using triangular membership functions, we obtain Figure 1.

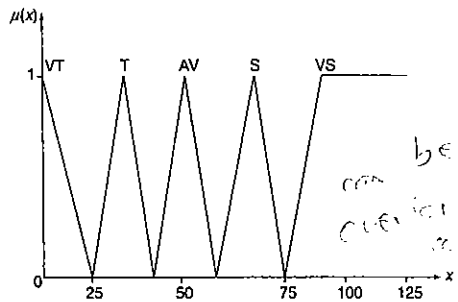


Figure 1 Membership function for weight of people.

2. Using your own intuition, plot the fuzzy membership function for the age of people.

Solution: The universe of discourse is age of people. Let "A" denote age of people in years. The linguistic variables are defined as follows:

Very young (VY) : $A < 12$
Young (Y) : $10 \leq A \leq 22$

Middle age (M) : $20 \le A \le 42$

Old (O) : $40 \le A \le 72$

Very old (VO) : $70 < A$

These variables are represented using triangular membership function in Figure 2.

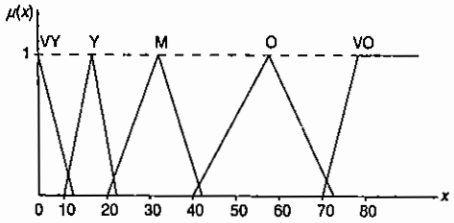


Figure 2 Membership function for age of people.

3. Compare "medium wave (MW)" and "short wave (SW)" receivers according to their frequency range. Plot the membership functions using intuition. The linguistic variables are defined based on the following:

   Medium wave receivers: frequency lesser than $\approx 10^6$ Hz

   Short wave receivers: frequency greater than $\approx 10^6$ Hz

Solution: Let the frequency range of receivers be universe of discourse. The linguistic variables are the following:

Medium wave receivers (MW): frequency lesser than $\approx 10^6$ Hz

Short wave receivers (SW): frequency greater than $\approx 10^6$ Hz

This is represented using Gaussian membership function in Figure 3.

4. Using the inference approach, find the membership values for the triangular shapes $\underline{I}, \underline{R}, \underline{E}, \underline{IR}$, and $\underline{T}$ for a triangle with angles 45°, 55° and 80°.

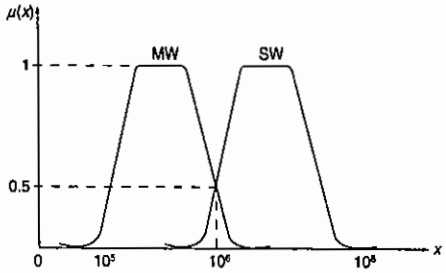


Figure 3 Membership function for frequency range of receivers.

Solution: Let the universe of discourse be

$$U = \{(X, Y, Z) : X = 80° \ge Y = 55° \ge Z = 45° \text{ and } X + Y + Z = 80° + 55° + 45° = 180°\}$$

- Membership value of isosceles triangle, $\underline{I}$:

$$\mu_{\underline{I}} = 1 - \frac{1}{60°}\min(X - Y, Y - Z)$$
$$= 1 - \frac{1}{60°}\min(80° - 55°, 55° - 45°)$$
$$= 1 - \frac{1}{60°}\min(25°, 10°)$$
$$= 1 - \frac{1}{60°} \times 10°$$
$$= 1 - 0.1667 = 0.833$$

- Membership value of right-angle triangle, $\underline{R}$:

$$\mu_{\underline{R}} = 1 - \frac{1}{90°}|X - 90°| = 1 - \frac{1}{90°}|80° - 90°|$$
$$= 1 - \frac{1}{90°} \times 10° = 0.889$$

- Membership value of equilateral triangle, $\underline{E}$:

$$\mu_{\underline{E}} = 1 - \frac{1}{180°}(X - Z) = 1 - \frac{1}{180°}(80° - 45°)$$
$$= 1 - \frac{1}{180°} \times 35° = 0.8056$$

- Membership value of isosceles and right-angle triangle, $\underline{IR}$:

$$\mu_{\underline{IR}} = \min[\mu_{\underline{I}}, \mu_{\underline{R}}] = \min[0.833, 0.889]$$
$$= 0.833$$

- Membership value of other triangles, $\underline{T}$:

$$\mu_{\underline{T}} = \min[1 - \mu_{\underline{I}}, 1 - \mu_{\underline{E}}, 1 - \mu_{\underline{R}}]$$
$$= \min[0.167, 0.1944, 0.111] = 0.111$$

Thus the membership function is calculated for the triangular shapes.

5. Using the inference approach, obtain the membership values for the triangular shapes $(\underline{I}, \underline{R}, \underline{T})$ for a triangle with angles 40°, 60° and 80°.

Solution: Let the universe of discourse be

$$U = \{(X, Y, Z) : X = 80° \ge Y = 60° \ge Z = 40° \text{ and } X + Y + Z = 80° + 60° + 40° = 180°\}$$

- Membership value of isosceles triangle, $\underline{I}$:

$$\mu_{\underline{I}} = 1 - \frac{1}{60°}\min(X - Y, Y - Z)$$
$$= 1 - \frac{1}{60°}\min(80° - 60°, 60° - 40°)$$
$$= 1 - \frac{1}{60°}\min(20°, 20°)$$
$$= 1 - \frac{1}{60°} \times 20° = 0.667$$

- Membership value of right-angle triangle, $\underline{R}$:

$$\mu_{\underline{R}} = 1 - \frac{1}{90°}|X - 90°| = 1 - \frac{1}{90°}|80° - 90°|$$
$$= 1 - \frac{1}{90°} \times 10° = 0.889$$

- Membership value of other triangles, $\underline{T}$:

$$\mu_{\underline{T}} = \min[1 - \mu_{\underline{I}}, 1 - \mu_{\underline{R}}]$$
$$= \min[1 - 0.667, 1 - 0.889]$$
$$= \min[0.333, 0.111] = 0.111$$

Thus the membership values for isosceles, right-angle triangle and other triangles are calculated.

6. The energy E of a particle spinning in a magnetic field B is given by the equation

$$E = \mu B \sin\theta$$

where μ is magnetic moment of spinning particle and θ is complement angle of magnetic moment

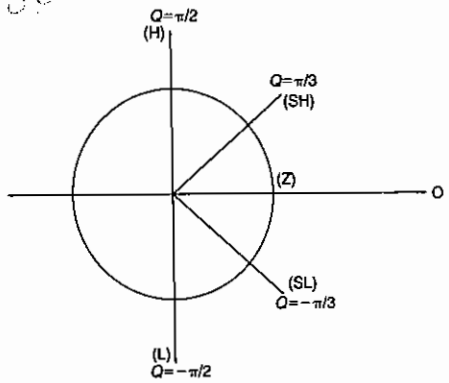


Figure 4 Angular fuzzy set.

with respect to the direction of the magnetic field.

Assume the magnetic field B and magnetic moment μ to be constant, and the linguistic terms for the complement angle of magnetic moment be given as

High moment (H): $\theta = \pi/2$

Slightly high moment (SH): $\theta = \pi/3$

No moment (Z): $\theta = 0$

Slightly low moment (SL): $\theta = -\pi/3$

Low moment (L): $\theta = -\pi/2$

Find the membership values using the angular fuzzy set approach for these linguistic labels and plot these values versus θ.

Solution: The angular fuzzy set is shown in Figure 4. Now calculate the angular fuzzy membership values as shown in the table below.

θ	$\tan\theta$	$z = \cos\theta$	$\mu = \lvert(z)\tan\theta\rvert$
$\pi/2$	∞	0	1
$\pi/3$	1.732	0.5	0.866
0	0	1	0
$-\pi/3$	−1.732	0.5	−0.866
$-\pi/2$	∞	0	1

The plot for the membership function shown in this table is given in Figure 5.

Table 1

	Number who preferred							
	Maruti 800	Scorpio	Matiz	Santro	Octavia	Total	Percentage	Rank order
Maruti 800	—	192	246	592	621	1651	16.5	5
Scorpio	403	—	621	540	391	1955	19.6	2
Matiz	235	336	—	797	492	1860	18.6	4
Santro	523	364	417	—	608	1912	19.1	3
Octavia	616	534	746	726	—	2622	26.2	1
Total						10000		

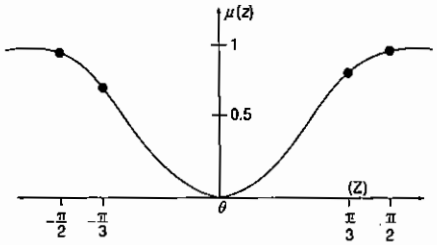


Figure 5 Plot of membership function.

7. Suppose 1000 people respond to a questionnaire about their pairwise preferences among five cars, X = {Maruti 800, Scorpio, Matiz, Santro, Octavia}. Define a fuzzy set $\underline{A}$ on the universe of cars "best car."

Solution: Table 1 shows the rank ordering for performance of cars is a summary of the opinion survey.

In Table 1, for example, out of 1000 people, 192 preferred Maruti 800 to the Scorpio, etc. The total number of responses is 10,000 (10 comparisons). On the basis of the preferences, the percentage is calculated. The ordering is then performed. It is found that Octavia is selected as the best car. Figure 6 shows the membership function for this example.

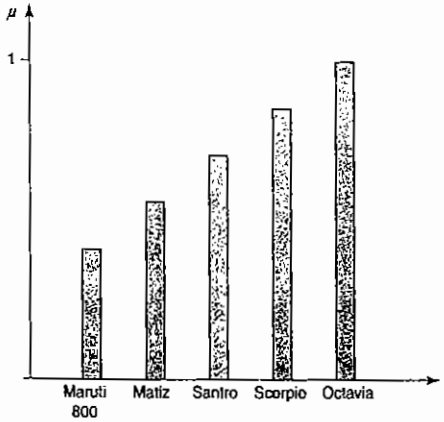


Figure 6 Membership function for best car.

9.7 Review Questions

1. Define membership function and state its importance in fuzzy logic.
2. Explain the features of membership functions.
3. Differentiate the following:
   - Convex and nonconvex fuzzy set.
   - Normal and subnormal fuzzy set.
4. What is meant by crossover point in a fuzzy set?
5. Define height of a fuzzy set.
6. Write short note on fuzzification.
7. List the various methods employed for the membership value assignment.
8. With suitable examples, explain how membership assignment is performed using intuition.
9. Define fuzzy number.
10. Explain in detail the inference method adopted for assigning membership values.
11. How is rank ordering used to define membership functions based on polling concept?
12. Discuss in detail the membership value assignments using angular fuzzy sets.
13. Describe how neural network is used to obtain fuzzy membership functions.
14. With suitable example, explain the method by which membership value assignments are performed using genetic algorithm.
15. Give details on membership value assignments using inductive reasoning.

9.8 Exercise Problems

1. Using intuition, assign the membership functions for (a) population of cars and (b) library usage.
2. Using your own intuition, develop fuzzy membership functions on the real line for the fuzzy number 5, using the following shapes:
   (a) Quadrilateral
   (b) Trapezoid
   (c) Gaussian function
   (d) Isosceles triangle
   (e) Symmetric triangle
3. Using intuition and your own definition of the universe of discourse, plot fuzzy membership functions to the following variables:
   (i) Liquid level in the tank
      (a) Very small
      (b) Small
      (c) Empty
      (d) Full
      (e) Very full
   (ii) Race of people
      (a) White
      (b) Moderate
      (c) Black
   (iii) Height of people
      (a) Very tall
      (b) Tall
      (c) Normal
      (d) Short
      (e) Very short
4. Using inference approach outlined in this chapter, find the membership values for each of the triangular shapes ($\underline{L}, \underline{R}, \underline{E}, \underline{IR}, \underline{T}$) for each of the following (all in degrees):
   (a) 20°, 40°, 120°
   (b) 90°, 45°, 45°
   (c) 35°, 75°, 70°
   (d) 10°, 60°, 110°
   (e) 50°, 75°, 55°

5. Using inference method, find the membership values of the triangular shapes for each of the following triangles:
   (a) 30°, 60°, 90°
   (b) 45°, 65°, 70°
   (c) 85°, 55°, 40°

6. The following data was determined by the pair-wise comparison of work preferences of 100 people: When it was compared with software (S), 72 persons polled preferred hardware (H), 65 of them preferred teaching (T), 55 of them preferred business (B) and 25 preferred textile (TX). On comparison with hardware (H), the preferences were 60 for S, 42 for T, 66 for B and 35 for TX. When compared with teaching, the preferences were 62 for S, 48 for H, 38 for B and 25 for TX. On comparison with business, the preferences were 52 for S, 47 for H, 35 for T, 20 for TX. When compared with textile, the preferences were 70 for S, 65 for H, 44 for T and 40 for B. Using rank ordering plot the membership function for the "most preferred work."

7. The following raw data determines a pair-wise comparison of a new scooter in a poll of 100 people. On comparison with Victor (V), 79 preferred Splender (S), 59 preferred Honda Activa (HA), 85 preferred Bajaj (B) and 62 preferred Infinity (I). When S was compared, the preferences were 21 for V, 22 for HA, 37 for B and 45 for I. When HA was compared, the preferences were 20 for V, 77 for S, 35 for B and 48 for I. Finally when infinity was compared, the preferences were 32 for V, 54 for S, 52 for HA and 50 for B. Using rank ordering, obtain the membership function of "most preferred bike."

8. Develop membership function for trapezoid similar to algorithm developed for triangle and the function should have two independent variables so that it can be passed. For the table shown, show the first iteration to compute the membership values for input variables X_1, X_2 and X_3 in the output regions R_A and R_B.

X_1	X_2	X_3	R_A	R_B
1.5	0.5	2.5	1.0	0.0

   (a) Use 3 × 3 × 1 neural network.
   (b) Use 3 × 3 × 2 neural network.

9. For data shown in the following table (Table A), show the first two iterations using a genetic algorithm to find the optimum membership function (right triangular function S) for the input variable X and output variable Y in the rule table.

Table A: Data

X	0	0.2	0.7	1.0
Y	1	0.64	0.55	0.35

Table B: Rules

X	L	S
Y	Z	S

Note: L – large; S – small; Z – zero.

10. The energy E of a particular spinning in a magnetic field B is given by the equation

$$E = \mu B \sin\theta$$

where μ is magnetic moment of spinning particle and θ complement angle of magnetic moment with respect to the direction of the magnetic field.

Assuming the magnetic field and magnetic moment to be constant, we propose linguistic terms for the complement angle of magnetic moment as follows:

High moment (H): $\theta = \pi/2$

Slightly high moment (SH): $\theta = \pi/8$

No moment (N): $\theta = 0$

Slightly low moment (SL): $\theta = -\pi/8$

Low moment (L): $\theta = -\pi/2$

Find the membership values using the angular fuzzy set approach for these linguistic labels for the complement angles and plot these value versus "θ."

10 Defuzzification

Learning Objectives

- Need for defuzzification process.
- How lambda-cuts for fuzzy sets and fuzzy relations can be carried out.
- Various types of defuzzification methods.
- To know how λ-cut relation of a fuzzy tolerance and fuzzy equivalence relation results in crisp tolerance and crisp equivalence relation respectively.
- An example provided to depict how the various defuzzification methods are used to obtain crisp outputs.

10.1 Introduction

In fuzzification process, we have made the conversion from crisp quantities to fuzzy quantities; however, in several applications and engineering area, it is necessary to "defuzzify" the fuzzy results we have generated through the fuzzy set analysis, i.e., it is necessary to convert fuzzy results into crisp results. Defuzzification is a mapping process from a space of fuzzy control actions defined over an output universe of discourse into a space of crisp (nonfuzzy) control actions. This is required because in many practical applications crisp control actions are needed to actuate the control. A defuzzification process produces a nonfuzzy control action that best represents the possibility distribution of an inferred fuzzy control action. The defuzzification process has the capability to reduce a fuzzy set into a crisp single-valued quantity or into a crisp set; to convert a fuzzy matrix into a crisp matrix; or to convert a fuzzy number into a crisp number. Mathematically, the defuzzification process may also be termed as "rounding it off." Fuzzy set with a collection of membership values or a vector of values on the unit interval may be reduced to a single scalar quantity using defuzzification process. Enormous defuzzification methods have been suggested in the literature; although no method has proved to be always more advantageous than the others. The selection of the method to be used depends on the experience of the designer. It may be done on the basis of the computational complexity involved, applicability to the situations considered and plausibility of the outputs obtained based on engineering point of view. In this chapter we will discuss the various defuzzification methods employed for converting fuzzy variables into crisp variables.

10.2 Lambda-Cuts for Fuzzy Sets (Alpha-Cuts)

Consider a fuzzy set $\underset{\sim}{A}$. The set $A_\lambda (0 < \lambda < 1)$, called the lambda (λ)-cut (or alpha [α]-cut) set, is a crisp set of the fuzzy set and is defined as follows:

$$A_\lambda = \{x \mid \mu_{\underset{\sim}{A}}(x) \geq \lambda\}; \quad \lambda \in [0, 1]$$

The set A_λ is called a weak lambda-cut set if it consists of all the elements of a fuzzy set whose membership functions have values greater than or equal to a specified value. On the other hand, the set A_λ is called a strong lambda-cut set if it consists of all the elements of a fuzzy set whose membership functions have values strictly greater than a specified value. A strong λ-cut set is given by

$$A_\lambda = \{x | \mu_{\underset{\sim}{A}}(x) > \lambda\}; \quad \lambda \in [0, 1]$$

All the λ-cut sets form a family of crisp sets. It is important to note the λ-cut set A_λ (or A_α, if α-cut set) does not have a tilde score, because it is a crisp set derived from parent fuzzy set $\underset{\sim}{A}$. Any particular fuzzy set $\underset{\sim}{A}$ can be transformed into an infinite number of λ-cut sets, because there are infinite number of values λ can take in the interval $[0, 1]$.

The properties of λ-cut sets are as follows:

1. $(\underset{\sim}{A} \cup \underset{\sim}{B})_\lambda = A_\lambda \cup B_\lambda$
2. $(\underset{\sim}{A} \cap \underset{\sim}{B})_\lambda = A_\lambda \cap B_\lambda$
3. $(\overline{\underset{\sim}{A}})_\lambda \neq (\overline{A_\lambda})$ except when $\lambda = 0.5$.
4. For any $\lambda \leq \beta$, where $0 \leq \beta \leq 1$, it is true that $A_\beta \subseteq A_\lambda$, where $A_0 = X$.

The fourth property is essentially used in graphics. Figure 10-1 shows a continuous-valued fuzzy set with two λ-cut values. In Figure 10-1, notice that for $\lambda = 0.2$ and $\beta = 0.5, A_{0.2}$ has a greater domain than $A_{0.5}$, i.e., for $\lambda \leq \beta$ $(0.2 \leq 0.5), A_{0.5} \subseteq A_{0.2}$. Figure 10-2 shows the features of the membership functions. The core of $\underset{\sim}{A}$ is the $\lambda = 1$-cut set A_1. The support of $\underset{\sim}{A}$ is the λ-cut set A_{0+}, where $\lambda = 0^+$, and it can be defined as

$$A_{0+} = \{x | \mu_{\underset{\sim}{A}}(x) > 0\}$$

The interval $[A_{0+}, A_1]$ forms the boundaries of the fuzzy set $\underset{\sim}{A}$, i.e., the regions with the membership values between 0 and 1, i.e., for $\lambda = 0$ to 1.

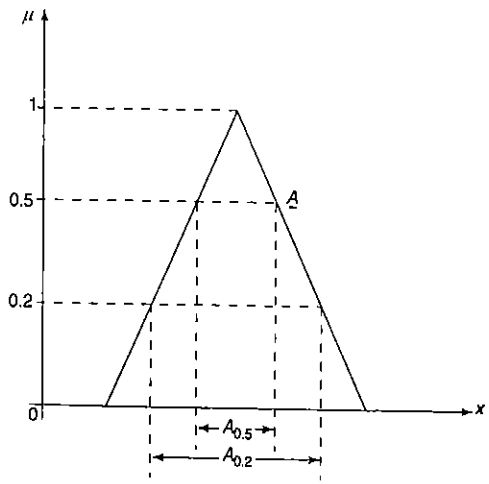


Figure 10-1 Two different λ-cut sets for a continuous-valued fuzzy set.

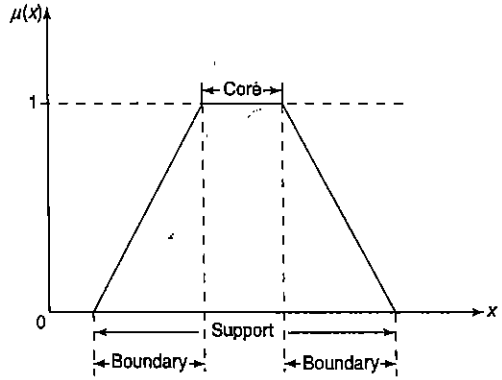


Figure 10-2 Features of the membership functions.

10.3 Lambda-Cuts for Fuzzy Relations

The λ-cut for fuzzy relations is similar to that for fuzzy sets. Let $\underset{\sim}{R}$ be a fuzzy relation where each row of the relational matrix is considered a fuzzy set. The jth row in a fuzzy relation matrix $\underset{\sim}{R}$ denotes a discrete membership function for a fuzzy set $\underset{\sim}{R}$. A fuzzy relation can be converted into a crisp relation in the following manner:

$$R_\lambda = \{(x, y) | \mu_{\underset{\sim}{R}}(x, y) \geq \lambda\}$$

where R_λ is a λ-cut relation of the fuzzy relation $\underset{\sim}{R}$. Since here $\underset{\sim}{R}$ is defined as a two-dimensional array, defined on the universes X and Y, therefore any pair $(x, y) \in R_\lambda$ belongs to $\underset{\sim}{R}$ with a relation greater than or equal to λ.

Similar to the properties of λ-cut fuzzy set, the λ-cuts on fuzzy relations also obey certain properties. They are listed as follows. For two fuzzy relations $\underset{\sim}{R}$ and $\underset{\sim}{S}$ the following properties should hold:

1. $(\underset{\sim}{R} \cup \underset{\sim}{S})_\lambda = R_\lambda \cup S_\lambda$.
2. $(\underset{\sim}{R} \cap \underset{\sim}{S})_\lambda = R_\lambda \cap S_\lambda$.
3. $(\overline{\underset{\sim}{R}})_\lambda \neq (\overline{R_\lambda})$ except when $\lambda = 0.5$.
4. For any $\lambda \leq \beta$, where $0 \leq \beta \leq 1$, it is true that $R_\beta \subseteq R_\lambda$.

10.4 Defuzzification Methods

Defuzzification is the process of conversion of a fuzzy quantity into a precise quantity. The output of a fuzzy process may be union of two or more fuzzy membership functions defined on the universe of discourse of the output variable.

Consider a fuzzy output comprising two parts: the first part, $\underset{\sim}{C}_1$, a triangular membership shape [as shown in Figure 10-3(A)], the second part, $\underset{\sim}{C}_2$, a trapezoidal shape [as shown in Figure 10-3(B)]. The union of these two membership functions, i.e., $\underset{\sim}{C} = \underset{\sim}{C}_1 \cup \underset{\sim}{C}_2$ involves the max-operator, which is going to be the outer envelope of the two shapes shown in Figures 10-3(A) and (B); the final shape of $\underset{\sim}{C}$ is shown in Figure 10-3(C).

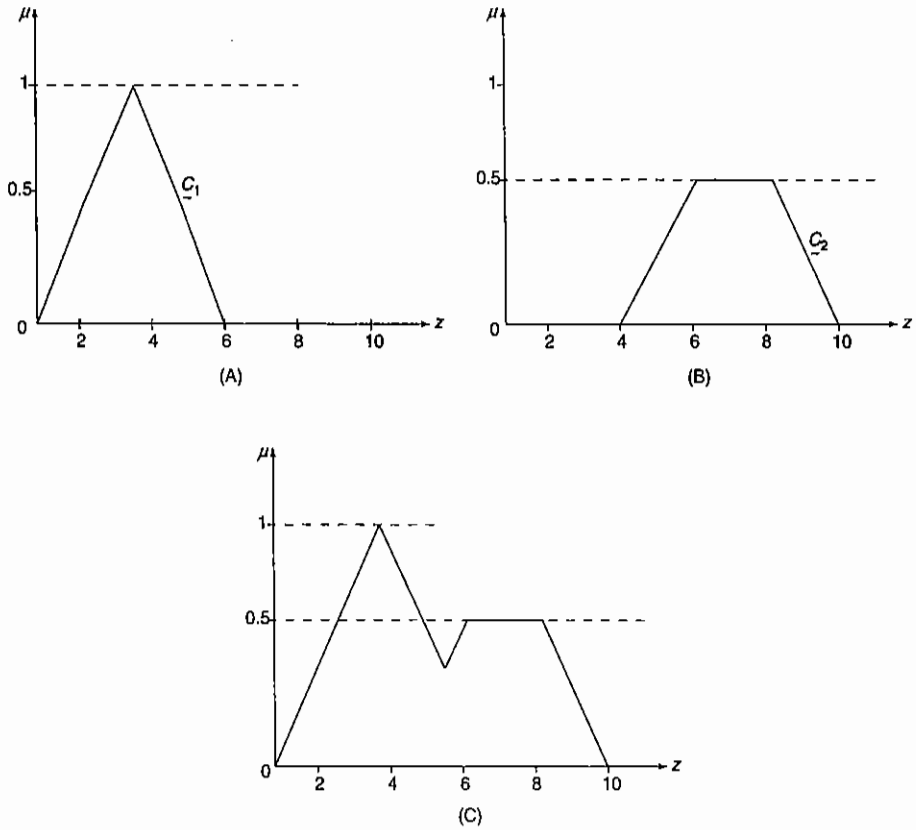


Figure 10-3 (A) First part of fuzzy output, (B) second part of fuzzy output, (C) union of parts (A) and (B).

A fuzzy output process may involve many output parts, and the membership function representing each part of the output can have any shape. The membership function of the fuzzy output need not always be normal. In general, we have

$$\underset{\sim}{C}_n = \bigcup_{i=1}^{n} \underset{\sim}{C}_i = \underset{\sim}{C}$$

Defuzzification methods include the following:

1. Max-membership principle.
2. Centroid method.
3. Weighted average method.
4. Mean-max membership.

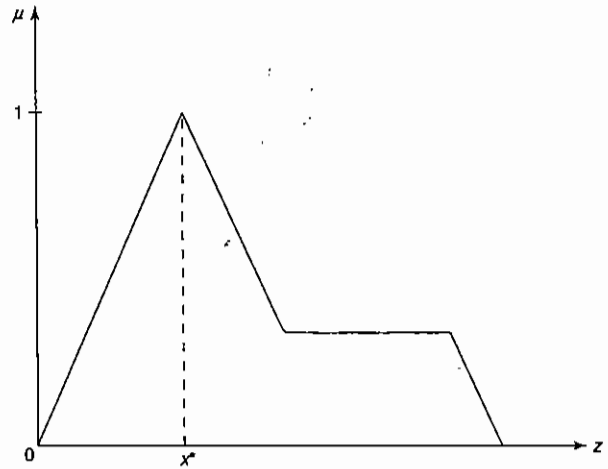


Figure 10-4 Max-membership defuzzification method.

5. Center of sums.
6. Center of largest area.
7. First of maxima, last of maxima.

Now we discuss the methods listed above.

10.4.1 Max-Membership Principle

This method is also known as height method and is limited to peak output functions. This method is given by the algebraic expression

$$\mu_{\underset{\sim}{C}}(x^*) \geq \mu_{\underset{\sim}{C}}(x) \text{ for all } x \in X$$

The method is illustrated in Figure 10-4.

10.4.2 Centroid Method

This method is also known as center of mass, center of area or center of gravity method. It is the most commonly used defuzzification method. The defuzzified output x^* is defined as

$$x^* = \frac{\int \mu_{\underset{\sim}{C}}(x) \cdot x dx}{\int \mu_{\underset{\sim}{C}}(x) dx}$$

where the symbol $\int$ denotes an algebraic integration. This method is illustrated in Figure 10-5.

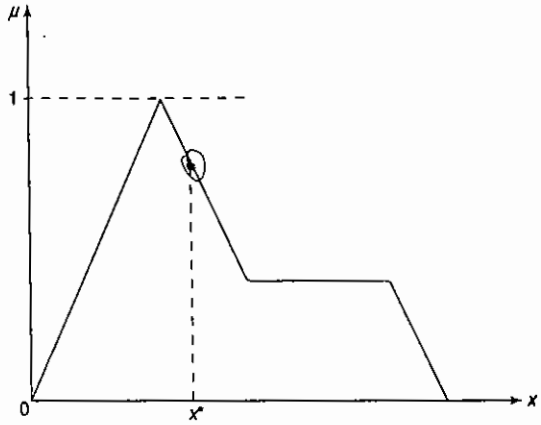


Figure 10-5 Centroid defuzzification method.

10.4.3 Weighted Average Method

This method is valid for symmetrical output membership functions only. Each membership function is weighted by its maximum membership value. The output in this case is given by

$$x^* = \frac{\sum \mu_C(\bar{x}_i) \cdot \bar{x}_i}{\sum \mu_C(\bar{x}_i)}$$

where $\sum$ denotes algebraic sum and $\bar{x}_i$ is the maximum of the ith membership function. The method is illustrated in Figure 10-6, where two fuzzy sets are considered. From Figure 10-6, we notice that the defuzzified output is given by

$$x^* = \frac{0.5a + 0.8b}{0.5 + 0.8}$$

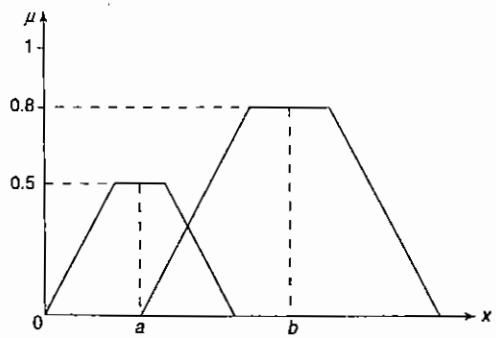


Figure 10-6 Weighted average defuzzification method (two symmetrical membership functions).

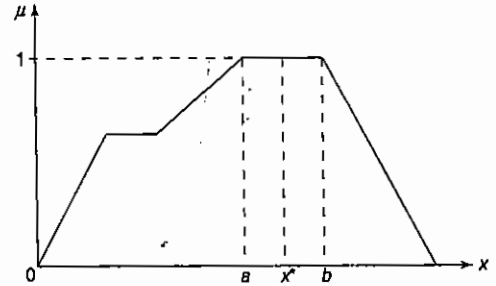


Figure 10-7 Mean-max membership defuzzification method.

As this method is limited to symmetrical membership functions, the values of a and b are the means of their respective shapes.

10.4.4 Mean-Max Membership

This method is also known as the middle of the maxima. This is closely related to max-membership method, except that the locations of the maximum membership can be nonunique. The output here is given by

$$x^* = \frac{\sum_{i=1}^{n} \bar{x}_i}{n}$$

This is illustrated in Figure 10-7. From Figure 10-7, we notice that the defuzzified output is given by

$$x^* = \frac{a + b}{2}$$

where a and b are as shown in the figure.

10.4.5 Center of Sums

This method employs the algebraic sum of the individual fuzzy subsets instead of their union. The calculations here are very fast, but the main drawback is that intersecting areas are added twice. The defuzzified value x^* is given by

$$x^* = \frac{\int_x x \sum_{i=1}^{n} \mu_{C_i}(x)dx}{\int_x \sum_{i=1}^{n} \mu_{C_i}(x)dx}$$

Figure 10-8 illustrates the center of sums method. In center of sums method, the weights are the areas of the respective membership functions, whereas in the weighted average method the weights are individual membership values.

10.4.6 Center of Largest Area

This method can be adopted when the output consists of at least two convex fuzzy subsets which are not overlapping. The output in this case is biased towards a side of one membership function. When output fuzzy

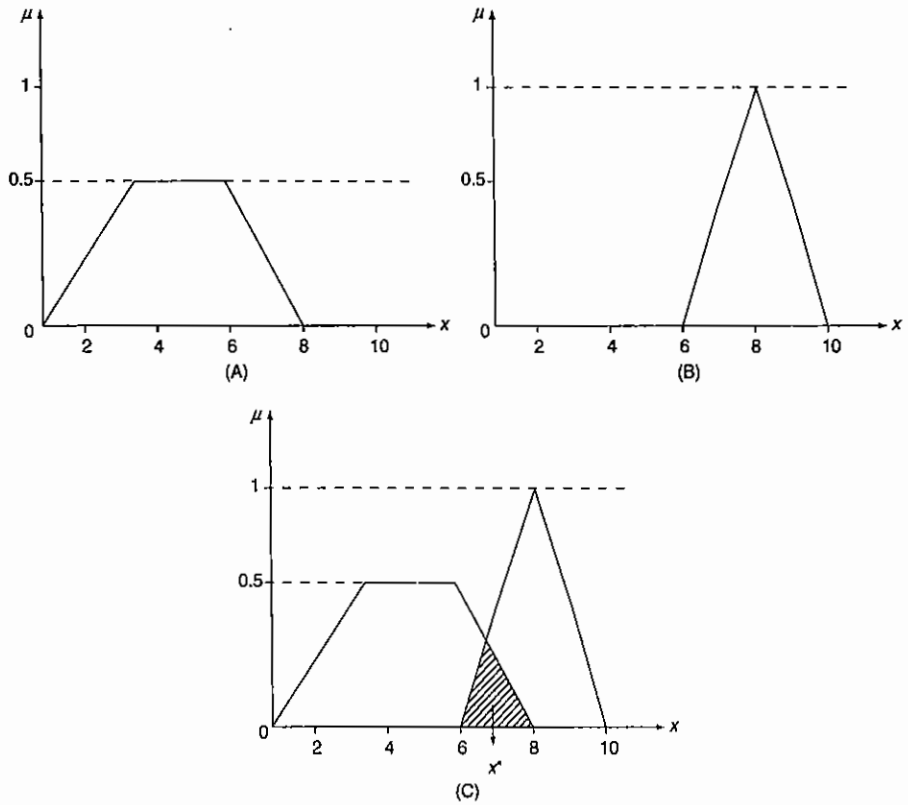


Figure 10-8 (A) First and (B) second membership functions, (C) defuzzification.

set has at least two convex regions, then the center of gravity of the convex fuzzy subregion having the largest area is used to obtain the defuzzified value x^*. This value is given by

$$x^* = \frac{\int \mu_{\mathcal{G}_i}(x) \cdot x dx}{\int \mu_{\mathcal{G}_i}(x) dx}$$

where $\mathcal{G}_i$ is the convex subregion that has the largest area making up $\mathcal{G}_i$. Figure 10-9 illustrates the center of largest area.

10.4.7 First of Maxima (Last of Maxima)

This method uses the overall output or union of all individual output fuzzy sets $\mathcal{G}_i$ for determining the smallest value of the domain with maximized membership in $\mathcal{G}_i$. The steps used for obtaining x^* are

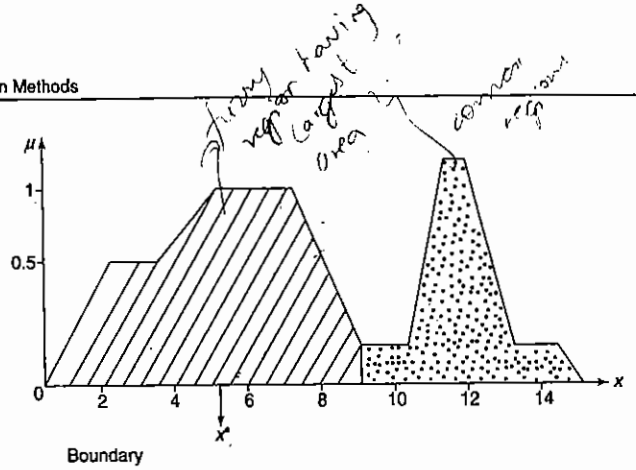


Figure 10-9 Center of largest area method.

as follows:

1. Initially, the maximum height in the union is found:

$$\text{hgt}(\mathcal{G}_i) = \sup_{x \in X} \mu_{\mathcal{G}_i}(x)$$

   where sup is supremum, i.e., the least upper bound.

2. Then the first of maxima is found:

$$x^* = \inf_{x \in X} \{x \in X | \mu_{\mathcal{G}_i}(x) = \text{hgt}(\mathcal{G}_i)\}$$

   where inf is the infimum, i.e., the greatest lower bound.

3. After this the last maxima is found:

$$x^* = \sup_{x \in X} \{x \in X | \mu_{\mathcal{G}_i}(x) = \text{hgt}(\mathcal{G}_i)\}$$

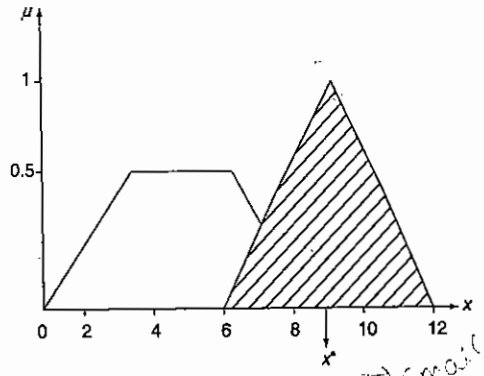


Figure 10-10 First of maxima (last of maxima) method.

where sup = supremum, i.e., the least upper bound; inf = infimum, i.e., the greatest lower bound. This is illustrated in Figure 10-10. From Figure 10-10, the first maxima is also the last maxima, and since it is a distinct max, it is also the mean-max.

10.5 Summary

In this chapter we have discussed the methods of converting fuzzy variables into crisp variables by a process called as defuzzification. Defuzzification process is essential because some engineering applications need exact values for performing the operation. For example, if speed of a motor has to be varied, we cannot instruct to raise it "slightly," "high," etc., using linguistic variables; rather, it should be specified as raise it by 200 rpm or so, a specific amount of raise should be mentioned. Defuzzification is a natural and essential technique. Lambda-cut for fuzzy sets and fuzzy relations were discussed. Apart from the lambda-cut method, seven defuzzification methods were presented. There are analyses going on to justify which of the defuzzification method is the best? The method of defuzzification should be assessed on the basis of the output in the context of data available.

10.6 Solved Problems

1. Consider two fuzzy sets $\underset{\sim}{A}$ and $\underset{\sim}{B}$, both defined on X, given as follows:

$\mu(x_i;X)$	x_1	x_2	x_3	x_4	x_5
$\underset{\sim}{A}$	0.2	0.3	0.4	0.7	0.1
$\underset{\sim}{B}$	0.4	0.5	0.6	0.8	0.9

Express the following λ-cut sets using Zadeh's notation:

(a) $(\overline{\underset{\sim}{A}})_{0.7}$; (b) $(\underset{\sim}{B})_{0.2}$; (c) $(\underset{\sim}{A} \cup \underset{\sim}{B})_{0.6}$;
(d) $(\underset{\sim}{A} \cap \underset{\sim}{B})_{0.5}$; (e) $(\underset{\sim}{A} \cup \overline{\underset{\sim}{A}})_{0.7}$; (f) $(\underset{\sim}{B} \cap \overline{\underset{\sim}{B}})_{0.3}$;
(g) $(\overline{\underset{\sim}{A} \cap \underset{\sim}{B}})_{0.6}$; (h) $(\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}})_{0.8}$

Solution: The two fuzzy sets given are

$$\underset{\sim}{A} = \left\{\frac{0.2}{x_1} + \frac{0.3}{x_2} + \frac{0.4}{x_3} + \frac{0.7}{x_4} + \frac{0.1}{x_5}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{0.4}{x_1} + \frac{0.5}{x_2} + \frac{0.6}{x_3} + \frac{0.8}{x_4} + \frac{0.9}{x_5}\right\}$$

We now find the λ-cut set:

$$A_\lambda = \{x \mid \mu_{\underset{\sim}{A}}(x) \geq \lambda\}$$

(a) $(\overline{\underset{\sim}{A}})_{0.7} = 1 - \mu_{\underset{\sim}{A}}(x)$

$$= \left\{\frac{0.8}{x_1} + \frac{0.7}{x_2} + \frac{0.6}{x_3} + \frac{0.3}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(\overline{A})_{0.7} = \{x_1, x_2, x_5\}$$

(b) $$\underset{\sim}{B} = \left\{\frac{0.4}{x_1} + \frac{0.5}{x_2} + \frac{0.6}{x_3} + \frac{0.8}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(B)_{0.2} = \{x_1, x_2, x_3, x_4, x_5\}$$

(c) $(\underset{\sim}{A} \cup \underset{\sim}{B}) = \max[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)]$

$$= \left\{\frac{0.4}{x_1} + \frac{0.5}{x_2} + \frac{0.6}{x_3} + \frac{0.8}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(A \cup B)_{0.6} = \{x_3, x_4, x_5\}$$

(d) $(\underset{\sim}{A} \cap \underset{\sim}{B}) = \min[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(x)]$

$$= \left\{\frac{0.2}{x_1} + \frac{0.3}{x_2} + \frac{0.4}{x_3} + \frac{0.7}{x_4} + \frac{0.1}{x_5}\right\}$$

$$(A \cap B)_{0.5} = \{x_4\}$$

(e) $(\underset{\sim}{A} \cup \overline{\underset{\sim}{A}}) = \max[\mu_{\underset{\sim}{A}}(x), \mu_{\overline{\underset{\sim}{A}}}(x)]$

$$= \left\{\frac{0.8}{x_1} + \frac{0.7}{x_2} + \frac{0.6}{x_3} + \frac{0.7}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(A \cup \overline{A})_{0.7} = \{x_1, x_2, x_4, x_5\}$$

(f) $(\underset{\sim}{B} \cap \overline{\underset{\sim}{B}}) = \min[\mu_{\underset{\sim}{B}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)]$

$$= \left\{\frac{0.4}{x_1} + \frac{0.5}{x_2} + \frac{0.4}{x_3} + \frac{0.2}{x_4} + \frac{0.1}{x_5}\right\}$$

$$(B \cap \overline{B})_{0.3} = \{x_1, x_2, x_3\}$$

(g) $\overline{(\underset{\sim}{A} \cap \underset{\sim}{B})} = 1 - \mu_{(\underset{\sim}{A} \cap \underset{\sim}{B})}$

$$= \left\{\frac{0.8}{x_1} + \frac{0.7}{x_2} + \frac{0.6}{x_3} + \frac{0.3}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(\overline{A \cap B})_{0.6} = \{x_1, x_2, x_3, x_5\}$$

(h) $(\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}}) = \max[\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(x)]$

$$= \left\{\frac{0.8}{x_1} + \frac{0.7}{x_2} + \frac{0.6}{x_3} + \frac{0.3}{x_4} + \frac{0.9}{x_5}\right\}$$

$$(\overline{A} \cup \overline{B})_{0.8} = \{x_1, x_5\}$$

2. Using Zadeh's notation, determine the λ-cut sets for the given fuzzy sets:

$$\underset{\sim}{S_1} = \left\{\frac{0}{0} + \frac{0.5}{20} + \frac{0.65}{40} + \frac{0.85}{60} + \frac{1.0}{80} + \frac{1.0}{100}\right\}$$

$$\underset{\sim}{S_2} = \left\{\frac{0}{0} + \frac{0.45}{20} + \frac{0.6}{40} + \frac{0.8}{60} + \frac{.95}{80} + \frac{1.0}{100}\right\}$$

Express the following for $\lambda = 0.5$:

(a) $(\underset{\sim}{S_1} \cup \underset{\sim}{S_2})$; (b) $(\underset{\sim}{S_1} \cap \underset{\sim}{S_2})$; (c) $\overline{\underset{\sim}{S_1}}$; (d) $\overline{\underset{\sim}{S_2}}$;
(e) $\overline{(\underset{\sim}{S_1} \cup \underset{\sim}{S_2})}$; (f) $\overline{(\underset{\sim}{S_1} \cap \underset{\sim}{S_2})}$

Solution: The two fuzzy sets given are

$$\underset{\sim}{S_1} = \left\{\frac{0}{0} + \frac{0.5}{20} + \frac{0.65}{40} + \frac{0.85}{60} + \frac{1.0}{80} + \frac{1.0}{100}\right\}$$

$$\underset{\sim}{S_2} = \left\{\frac{0}{0} + \frac{0.45}{20} + \frac{0.6}{40} + \frac{0.8}{60} + \frac{.95}{80} + \frac{1.0}{100}\right\}$$

The λ-cut set is obtained using

$$A_\lambda = \{x \mid \mu_{\underset{\sim}{A}}(x) \geq \lambda\}$$

Here $\lambda = 0.5$.

(a) $(\underset{\sim}{S_1} \cup \underset{\sim}{S_2}) = \max[\mu_{\underset{\sim}{S_1}}(x), \mu_{\underset{\sim}{S_2}}(x)]$

$$= \left\{\frac{0}{0} + \frac{0.5}{20} + \frac{0.65}{40} + \frac{0.85}{60} + \frac{1.0}{80} + \frac{1.0}{100}\right\}$$

$$(S_1 \cup S_2)_{0.5} = \{20, 40, 60, 80, 100\}$$

(b) $(\underset{\sim}{S_1} \cap \underset{\sim}{S_2}) = \min[\mu_{\underset{\sim}{S_1}}(x), \mu_{\underset{\sim}{S_2}}(x)]$

$$= \left\{\frac{0}{0} + \frac{0.45}{20} + \frac{0.6}{40} + \frac{0.8}{60} + \frac{.95}{80} + \frac{1.0}{100}\right\}$$

$$(S_1 \cap S_2)_{0.5} = \{40, 60, 80, 100\}$$

(c) $\overline{\underset{\sim}{S_1}} = 1 - \mu_{\underset{\sim}{S_1}}(x)$

$$= \left\{\frac{1}{0} + \frac{0.5}{20} + \frac{0.35}{40} + \frac{0.15}{60} + \frac{0}{80} + \frac{0}{100}\right\}$$

$$(\overline{S_1})_{0.5} = \{0, 20\}$$

(d) $\overline{\underset{\sim}{S_2}} = 1 - \mu_{\underset{\sim}{S_2}}(x)$

$$= \left\{\frac{1}{0} + \frac{0.55}{20} + \frac{0.4}{40} + \frac{0.2}{60} + \frac{0.05}{80} + \frac{0}{100}\right\}$$

$$(\overline{S_2})_{0.5} = \{0, 20\}$$

(e) $\overline{(\underset{\sim}{S_1} \cup \underset{\sim}{S_2})} = 1 - \mu_{\underset{\sim}{S_1} \cup \underset{\sim}{S_2}(x)}$

$$= \left\{\frac{1}{0} + \frac{0.5}{20} + \frac{0.35}{40} + \frac{0.15}{60} + \frac{0}{80} + \frac{0}{100}\right\}$$

$$\overline{(S_1 \cup S_2)}_{0.5} = \{0, 20\}$$

(f) $\overline{(\underset{\sim}{S_1} \cap \underset{\sim}{S_2})} = 1 - \mu_{\underset{\sim}{S_1} \cap \underset{\sim}{S_2}(x)}$

$$= \left\{\frac{1}{0} + \frac{0.55}{20} + \frac{0.4}{40} + \frac{0.2}{60} + \frac{0.05}{80} + \frac{0}{100}\right\}$$

$$\overline{(S_1 \cap S_2)}_{0.5} = \{0, 20\}$$

3. Consider the two fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{0}{0.2} + \frac{0.8}{0.4} + \frac{1}{0.6}\right\}$$

and $$\underset{\sim}{B} = \left\{\frac{0.9}{0.2} + \frac{0.7}{0.4} + \frac{0.3}{0.6}\right\}$$

Using Zadeh's notations, express the fuzzy sets into λ-cut sets for λ = 0.4 and λ = 0.7 for the following operations:

(a) $\overline{\underset{\sim}{A}}$; (b) $\overline{\underset{\sim}{B}}$; (c) $\underset{\sim}{A} \cup \underset{\sim}{B}$;

(d) $\underset{\sim}{A} \cap \underset{\sim}{B}$; (e) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}}$; (f) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}}$

Solution: The two fuzzy sets given are

$$\underset{\sim}{A} = \left\{\frac{0}{0.2} + \frac{0.8}{0.4} + \frac{1}{0.6}\right\}$$

and $$\underset{\sim}{B} = \left\{\frac{0.9}{0.2} + \frac{0.7}{0.4} + \frac{0.3}{0.6}\right\}$$

Case (i): λ = 0.4

(a) $\overline{\underset{\sim}{A}} = 1 - \mu_{\underset{\sim}{A}}(x) = \left\{\frac{1}{0.2} + \frac{0.2}{0.4} + \frac{0}{0.6}\right\}$

$(\overline{A})_{0.4} = \{0.2\}$

(b) $\overline{\underset{\sim}{B}} = 1 - \mu_{\underset{\sim}{B}}(y) = \left\{\frac{0.1}{0.2} + \frac{0.3}{0.4} + \frac{0.7}{0.6}\right\}$

$(\overline{B})_{0.4} = \{0.6\}$

(c) $\underset{\sim}{A} \cup \underset{\sim}{B} = \max[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(y)]$

$$= \left\{\frac{0.9}{0.2} + \frac{0.8}{0.4} + \frac{1}{0.6}\right\}$$

$(A \cup B)_{0.4} = \{0.2, 0.4, 0.6\}$

(d) $\underset{\sim}{A} \cap \underset{\sim}{B} = \min[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(y)]$

$$= \left\{\frac{0}{0.2} + \frac{0.7}{0.4} + \frac{0.3}{0.6}\right\}$$

$(A \cap B)_{0.4} = \{0.4\}$

(e) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}} = \max[\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(y)]$

$$= \left\{\frac{1}{0.2} + \frac{0.3}{0.4} + \frac{0.7}{0.6}\right\}$$

$(\overline{A} \cup \overline{B})_{0.4} = \{0.2, 0.6\}$

(f) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}} = \min[\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(y)]$

$$= \left\{\frac{0.1}{0.2} + \frac{0.2}{0.4} + \frac{0}{0.6}\right\}$$

$(\overline{A} \cap \overline{B})_{0.4} = \{\phi\}$

Case (ii): λ = 0.7

(a) $\overline{\underset{\sim}{A}} = 1 - \mu_{\underset{\sim}{A}}(x) = \left\{\frac{1}{0.2} + \frac{0.2}{0.4} + \frac{0}{0.6}\right\}$

$(\overline{A})_{0.7} = \{0.2\}$

(b) $\overline{\underset{\sim}{B}} = 1 - \mu_{\underset{\sim}{B}}(y) = \left\{\frac{0.1}{0.2} + \frac{0.3}{0.4} + \frac{0.7}{0.6}\right\}$

$(\overline{B})_{0.7} = \{0.6\}$

(c) $\underset{\sim}{A} \cup \underset{\sim}{B} = \max[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(y)]$

$$= \left\{\frac{0.9}{0.2} + \frac{0.8}{0.4} + \frac{1}{0.6}\right\}$$

$(A \cup B)_{0.7} = \{0.2, 0.4, 0.6\}$

(d) $\underset{\sim}{A} \cap \underset{\sim}{B} = \min[\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(y)]$

$$= \left\{\frac{0}{0.2} + \frac{0.7}{0.4} + \frac{0.3}{0.6}\right\}$$

$(A \cap B)_{0.7} = \{0.4\}$

(e) $\overline{\underset{\sim}{A}} \cup \overline{\underset{\sim}{B}} = \max[\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(y)]$

$$= \left\{\frac{1}{0.2} + \frac{0.3}{0.4} + \frac{0.7}{0.6}\right\}$$

$(\overline{A} \cup \overline{B})_{0.7} = \{0.2, 0.6\}$

(f) $\overline{\underset{\sim}{A}} \cap \overline{\underset{\sim}{B}} = \min[\mu_{\overline{\underset{\sim}{A}}}(x), \mu_{\overline{\underset{\sim}{B}}}(y)]$

$$= \left\{\frac{0.1}{0.2} + \frac{0.2}{0.4} + \frac{0}{0.6}\right\}$$

$(\overline{A} \cap \overline{B})_{0.7} = \{\phi\}$

4. Consider the discrete fuzzy set defined on the universe $X = \{a, b, c, d, e\}$ as

$$\underset{\sim}{A} = \left\{\frac{1}{a} + \frac{0.9}{b} + \frac{0.6}{c} + \frac{0.3}{d} + \frac{0}{e}\right\}$$

Using Zadeh's notation, find the λ-cut sets for λ = 1, 0.9, 0.6, 0.3, 0^+ and 0.

Solution: The fuzzy set given on the universe of discourse is

$$\underset{\sim}{A} = \left\{\frac{1}{a} + \frac{0.9}{b} + \frac{0.6}{c} + \frac{0.3}{d} + \frac{0}{e}\right\}$$

The λ-cut set is given as

$$A_\lambda = \{x \,|\, \mu_{\underset{\sim}{A}}(x) \geq \lambda\}$$

It should be noted that the sets present in λ-cut set will have unity membership and the sets not in λ-cut set have zero membership. Hence λ-cut sets for different values of λ can be expressed as follows.

(a) λ = 1, $A_1 = \left\{\frac{1}{a} + \frac{0}{b} + \frac{0}{c} + \frac{0}{d} + \frac{0}{e}\right\}$

(b) λ = 0.9, $A_{0.9} = \left\{\frac{1}{a} + \frac{1}{b} + \frac{0}{c} + \frac{0}{d} + \frac{0}{e}\right\}$

(c) λ = 0.6, $A_{0.6} = \left\{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{0}{d} + \frac{0}{e}\right\}$

(d) λ = 0.3, $A_{0.3} = \left\{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d} + \frac{0}{e}\right\}$

(e) $\lambda = 0^+$, $A_{0^+} = \left\{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d} + \frac{0}{e}\right\}$

(f) λ = 0, $A_0 = \left\{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d} + \frac{1}{e}\right\}$

5. Determine the crisp λ-cut relation when λ = 0.1, 0^+, 0.3 and 0.9 for the following relation $\underset{\sim}{R}$:

$$\underset{\sim}{R} = \begin{bmatrix} 0 & 0.2 & 0.4 \\ 0.3 & 0.7 & 0.1 \\ 0.8 & 0.9 & 1.0 \end{bmatrix}$$

Solution: For the given fuzzy relation, the λ-cut relation is given by

$$R_\lambda = \{(x, y) \,|\, \mu_{R(x,y)} \geq \lambda\}$$
$$= \{1 \,|\, \mu_{R(x,y)} \geq \lambda \,;\, 0 \,|\, \mu_{R(x,y)} < \lambda\}$$

(a) λ = 0.1,

$$R_{0.1} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(b) $\lambda = 0^+$,

$$R_{0^+} = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

(c) λ = 0.3,

$$R_{0.3} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

(d) λ = 0.9,

$$R_{0.9} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix}$$

6. For the fuzzy relation $\underset{\sim}{R}$,

$$\underset{\sim}{R} = \begin{bmatrix} 1 & 0.1 & 0 & 0.5 & 0.3 \\ 0.02 & 0.1 & 0.55 & 1 & 0.6 \\ 0.2 & 1 & 0.6 & 1 & 0 \\ 0.03 & 0.5 & 1 & 0.3 & 0 \end{bmatrix}$$

find the λ-cut relation for λ = 0^+, 0.1, 0.4 and 0.8.

Solution: For the given fuzzy relation, the λ-cut relation can be obtained by the following relation:

$$R_\lambda = \begin{cases} 1, & \mu_{R(x,y)} \geq \lambda \\ 0, & \mu_{R(x,y)} < \lambda \end{cases}$$

(a) $\lambda = 0^+$,

$$R_{0^+} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(b) $\lambda = 0.1$,

$$R_{0.1} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

(c) $\lambda = 0.4$,

$$R_{0.4} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

(d) $\lambda = 0.8$,

$$R_{0.8} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

7. For the fuzzy relation $\underset{\sim}{R}$,

$$\underset{\sim}{R} = \begin{bmatrix} 0.2 & 0.5 & 0.7 & 1 & 0.9 \\ 0.3 & 0.5 & 0.7 & 1 & 0.8 \\ 0.4 & 0.6 & 0.8 & 0.9 & 0.4 \\ 0.9 & 1 & 0.8 & 0.6 & 0.4 \end{bmatrix}$$

find the λ-cut relation for $\lambda = 0.2, 0.4, 0.7$ and 0.9

Solution: For the given fuzzy relation, the λ-cut relation is given by

$$R_\lambda = \begin{cases} 1, & \mu_{\underset{\sim}{R}(x,y)} \geq \lambda \\ 0, & \mu_{\underset{\sim}{R}(x,y)} < \lambda \end{cases}$$

(a) $\lambda = 0.2$,

$$R_{0.2} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(b) $\lambda = 0.4$,

$$R_{0.4} = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

(c) $\lambda = 0.7$,

$$R_{0.7} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

(d) $\lambda = 0.9$,

$$R_{0.9} = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

8. Show that any λ-cut relation of a fuzzy tolerance relation results in a crisp tolerance relation.

Solution: Consider the fuzzy relation

$$\underset{\sim}{R} = \begin{bmatrix} 1 & 0.8 & 0 & 0.1 & 0.2 \\ 0.8 & 1 & 0.4 & 0 & 0.9 \\ 0 & 0.4 & 1 & 0 & 0 \\ 0.1 & 0 & 0 & 1 & 0.5 \\ 0.2 & 0.9 & 0 & 0.5 & 1 \end{bmatrix}$$

It is a fuzzy tolerance relation because it does not satisfy transitive property, i.e.,

$$\mu_{\underset{\sim}{R}}(x_1, x_2) = 0.8, \quad \mu_{\underset{\sim}{R}}(x_2, x_5) = 0.9$$

From the relation $\underset{\sim}{R}$, we have

$$\mu_{\underset{\sim}{R}}(x_1, x_5) = 0.2 \qquad (1)$$

But on calculating we obtain

$$\mu_{\underset{\sim}{R}}(x_1, x_5) = \min\left[\mu_{\underset{\sim}{R}}(x_1, x_2), \mu_{\underset{\sim}{R}}(x_2, x_5)\right]$$
$$= \min[0.8, 0.9] = 0.8 \qquad (2)$$

As (1) $\neq$ (2), therefore transitive property is not satisfied. Now assume $\lambda = 0.8$. Then the crisp relation formed is

$$R_{0.8} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Now $(x_1, x_2) \in R$, $(x_2, x_5) \in R$, but $(x_1, x_5) \notin R$ $(x_1, x_5) \notin R$. Hence $R_{0.8}$ is a crisp tolerance relation. Thus λ-cut relation for a fuzzy tolerance relation is a crisp tolerance relation.

9. Show that λ-cut relation of a fuzzy equivalence relation results in a crisp equivalence relation.

Solution: Consider the following fuzzy equivalence relation:

$$\underset{\sim}{R} = \begin{bmatrix} 1 & 0.8 & 004 & 0.5 & 0.8 \\ 0.8 & 1 & 0.4 & 0.5 & 0.9 \\ 0.4 & 0.4 & 1 & 0.4 & 0.4 \\ 0.5 & 0.5 & 0.4 & 1 & 0.5 \\ 0.8 & 0.9 & 0.4 & 0.5 & 1 \end{bmatrix}$$

The relation $\underset{\sim}{R}$ satisfies transitive property, i.e.,

$$\mu_{\underset{\sim}{R}}(x_1, x_2) = 0.8, \quad \mu_{\underset{\sim}{R}}(x_2, x_5) = 0.9$$

From the relation $\underset{\sim}{R}$, we have

$$\mu_{\underset{\sim}{R}}(x_1, x_5) = 0.8 \qquad (1)$$

On calculating we obtain

$$\mu_{\underset{\sim}{R}}(x_1, x_5) = \min\left[\mu_{\underset{\sim}{R}}(x_1, x_2), \mu_{\underset{\sim}{R}}(x_2, x_5)\right]$$
$$= \min[0.8, 0.9] = 0.8 \qquad (2)$$

As (1)-= (2), therefore transitive property satisfied; hence it forms an equivalence relation. Now assume $\lambda = 0.8$. Then the crisp relation formed is

$$R_{0.8} = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Now $(x_1, x_2) \in R$, $(x_2, x_5) \in R$ and $(x_1, x_5) \in R$. Hence, λ-cut relation of a fuzzy equivalence relation results in a crisp equivalence relation.

10. For the given membership function as shown in Figure 1 below, determine the defuzzified output value by seven methods.

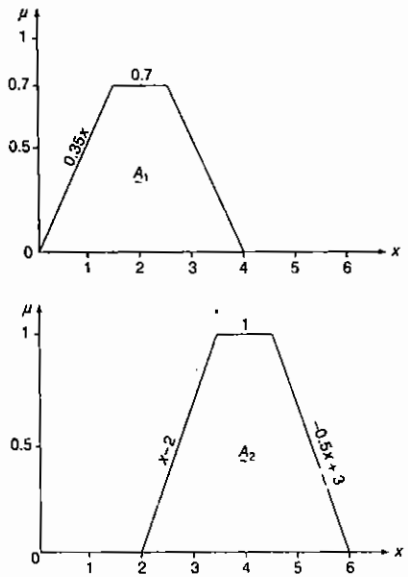


Figure 1 Membership functions.

Solution: The defuzzified output value can be obtained by the following methods.

- Centroid method

The two points are (0, 0) and (2, 0.7). The straight line is given by $(y - y_1) = m(x - x_1)$. Hence,

$$y - 0 = \frac{0.7}{2}(x - 0)$$
$$A_{11} \Rightarrow y = 0.35x$$
$$A_{12} \Rightarrow y = 0.7$$
$$A_{13} \Rightarrow \text{not necessary}$$
$$A_{21} \Rightarrow \text{the two points are } (2, 0), (3, 1)$$
$$y = x - 2$$
$$A_{22} \Rightarrow y = 1$$

$A_{23} \Rightarrow$ the two points are (4, 1), (6, 0)
we get $y = -0.5x + 3$

(A) From A_{12} we obtain $y = 0.7$.
(B) From A_{21} we obtain $y = x - 2$. On substituting the value $y = 0.7$ in (B), we obtain

$$x - 2 = 0.7 \Rightarrow x = 2.7$$
$$y = 0.7$$

The centroid method defuzzified output is

$$x^* = \int \frac{\mu_{\underset{\sim}{C}}(x)x\,dx}{\mu_{\underset{\sim}{C}}(x)\,dx}$$

$$= \frac{\left[\int_0^2 0.35x^2\,dx + \int_2^{2.7} 0.7x\,dx + \int_{2.7}^3 (x^2 - 2)\,dx + \int_3^4 x\,dx + \int_4^6 (-0.5x^2 + 3x)\,dx\right]}{\left[\int_0^2 0.35x^2\,dx + \int_2^{2.7} 0.7x\,dx + \int_{2.7}^3 (x^2 - 2)\,dx + \int_3^4 dx + \int_4^6 (-0.5x^2 + 3x)\,dx\right]}$$

$$= \frac{10.78}{3.445} = 3.187$$

- *Weighted average method:* The defuzzified value here is given by

$$x^* = \frac{2(0.7) + 4(1)}{0.7 + 1} = 3.176$$

- *Mean-max method:* The crisp output value here is given by

$$x^* = \frac{a + b}{2} = \frac{2.5 + 3.5}{2} = 3$$

- *Center of sums method:* The defuzzified value x^* is given by

$$x^* = \frac{\int_x x \sum_{i=1}^n \mu_{C_i}(x)\,dx}{\int_x \sum_{i=1}^n \mu_{C_i}(x)\,dx}$$

$$= \frac{\left[\int_0^6 \left[\frac{1}{2} \times 0.7 \times (3 + 2) \times 2 + \frac{1}{2} \times 1 \times (2 + 4) \times 4\right]dx\right]}{\left[\int_0^6 \left[\frac{1}{2} \times 0.7 \times (3 + 2) + \frac{1}{2} \times 1 \times (2 + 4) \times 4\right]dx\right]}$$

$$= \frac{\int_0^6 (3.5 + 12)\,dx}{\int_0^6 (1.75 + 3)\,dx} = 2.84$$

- *Center of largest area:*

$$\text{Area of I} = \frac{1}{2} \times 0.7 \times (2.7 + 0.7) = 1.19$$

$$\text{Area of II} = \frac{1}{2} \times 1 \times (2 + 3) \times \frac{1}{2} \times 0.7 = 2.255$$

Area of II is found to be larger; therefore the defuzzified output value is given by

$$x^* = \frac{\int \mu_{C_j}(x)x\,dx}{\int \mu_{C_j}(x)\,dx}$$

$$= \frac{\left[\int_{2.7}^3 \frac{1}{2} \times 0.3 \times 0.3 \times 2.85\,dx + \int_3^4 1 \times 1 \times 3.5\,dx + \int_4^6 \frac{1}{2} \times 2 \times 1\,dx\right]}{\left[\int_{2.7}^3 \frac{1}{2} \times 0.3 \times 0.3\,dx + \int_3^4 1 \times 1\,dx + \int_4^6 \frac{1}{2} \times 2 \times 1\,dx\right]}$$

$$= \frac{\int_{2.7}^3 0.12825\,dx + \int_3^4 3.5\,dx + \int_4^6 5\,dx}{\int_{2.7}^3 0.045\,dx + \int_3^4 dx + \int_4^6 dx} = 4.49$$

- *First of maxima:* The defuzzified output value is

$$x^* = 3$$

- *Last of maxima:* The defuzzified output value is

$$x^* = 4$$

10.7 Review Questions

1. Define defuzzification.
2. State the necessity of defuzzification process.
3. Write short note on lambda-cut for fuzzy sets.
4. List the properties of lambda-cut for fuzzy sets.
5. How is a fuzzy relation converted into a crisp relation using lambda-cut process?
6. Mention the properties of lambda-cut for fuzzy relations.
7. What are the different methods of defuzzification process?
8. Explain in detail the methods employed for converting fuzzy form into crisp form.
9. Compare first of maxima and last of maxima method.
10. What is the difference between centroid method and center of largest area method?
11. Differentiate between center of sums and weighted average method.
12. Which of the seven methods of the defuzzification technique is the best?

10.8 Exercise Problems

1. Two fuzzy sets defined on X, $\underset{\sim}{A}$ and $\underset{\sim}{B}$, are as follows:

$\mu(x_i)$	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$\underset{\sim}{A}$	0	0.1	0.2	0.3	0.4	0.5	0.6
$\underset{\sim}{B}$	1	0.9	0.8	0.7	0.6	0.5	0.4

Express the following λ-cut sets using Zadeh's notation:

(a) $(\overline{\underset{\sim}{A}})_{0.2}$; (b) $(\overline{\underset{\sim}{B}})_{0.6}$; (c) $(\underset{\sim}{A} \cup \underset{\sim}{B})_{0^+}$;
(d) $(\underset{\sim}{A} \cap \underset{\sim}{B})_{0.5}$; (e) $(\underset{\sim}{A} \cup \overline{\underset{\sim}{A}})_{0.7}$; (f) $(\underset{\sim}{A} \cap \overline{\underset{\sim}{A}})_{0.8}$;
(g) $(\underset{\sim}{B} \cup \overline{\underset{\sim}{B}})_{0.9}$; (h) $(\underset{\sim}{B} \cap \overline{\underset{\sim}{B}})_{0.4}$

2. Using Zadeh's notation, determine the λ-cut sets for the given fuzzy sets:

$$\underset{\sim}{M}_1 = \left\{\frac{0.1}{10} + \frac{0.4}{20} + \frac{0.35}{30} + \frac{0.7}{40} + \frac{1.0}{50}\right\}$$

$$\underset{\sim}{M}_2 = \left\{\frac{0}{10} + \frac{0.9}{20} + \frac{0.84}{30} + \frac{0.27}{40} + \frac{0.33}{50}\right\}$$

Express the following for λ = 0.2, 0.3 and 0.7:

(a) $(\underset{\sim}{M}_1 \cup \underset{\sim}{M}_2)$; (b) $(\underset{\sim}{M}_1 \cap \underset{\sim}{M}_2)$;
(c) $(\overline{\underset{\sim}{M}_1 \cup \underset{\sim}{M}_2})$; (d) $(\overline{\underset{\sim}{M}_1 \cap \underset{\sim}{M}_2})$;
(e) $(\overline{\underset{\sim}{M}_1} \cap \overline{\underset{\sim}{M}_2})$; (f) $(\overline{\underset{\sim}{M}_1} \cup \overline{\underset{\sim}{M}_2})$; (g) $\overline{\underset{\sim}{M}_1}$;
(h) $\overline{\underset{\sim}{M}_2}$; (i) $(\underset{\sim}{M}_1 \cup \overline{\underset{\sim}{M}_1})$; (j) $(\underset{\sim}{M}_1 \cap \overline{\underset{\sim}{M}_1})$

3. Consider the two fuzzy sets

$$\underset{\sim}{A} = \left\{\frac{0.35}{0.7} + \frac{0.625}{0.725} + \frac{0.256}{0.75}\right\}$$

$$\underset{\sim}{B} = \left\{\frac{0.95}{0.7} + \frac{0.815}{0.725} + \frac{0.67}{0.75}\right\}$$

Using Zadeh's notation, express the fuzzy sets as λ-cut sets for λ = 0.2 and λ = 0.8 for the following operations:

(a) $\overline{\underset{\sim}{A}}$; (b) $\overline{\underset{\sim}{B}}$; (c) $\underset{\sim}{A} \cup \underset{\sim}{B}$; (d) $\underset{\sim}{A} \cap \underset{\sim}{B}$;
(e) $\overline{\underset{\sim}{A} \cup \underset{\sim}{B}}$; (f) $\overline{\underset{\sim}{A} \cap \underset{\sim}{B}}$

4. Consider the fuzzy sets

$$\underset{\sim}{S}_1 = \left\{\frac{1}{100} + \frac{0.8}{200} + \frac{0.5}{300} + \frac{0.2}{400}\right\}$$

$$\underset{\sim}{S}_2 = \left\{\frac{0}{100} + \frac{0.7}{200} + \frac{0.4}{300} + \frac{0.1}{400}\right\}$$

Using Zadeh's notation, express the fuzzy sets as λ-cut sets for $\lambda = 0.2i$, $i = 1$ to 5, for the following operations:

(a) $\overline{\underset{\sim}{S}_1}$; (b) $\overline{\underset{\sim}{S}_2}$; (c) $\overline{\underset{\sim}{S}_1} \cap \underset{\sim}{S}_1$;
(d) $\overline{\underset{\sim}{S}_1} \cup \underset{\sim}{S}_1$; (e) $\overline{\underset{\sim}{S}_2} \cap \underset{\sim}{S}_2$; (f) $\overline{\underset{\sim}{S}_2} \cup \underset{\sim}{S}_2$;
(g) $(\overline{\underset{\sim}{S}_1 \cup \underset{\sim}{S}_2})$; (h) $(\overline{\underset{\sim}{S}_1 \cap \underset{\sim}{S}_2})$; (i) $(\overline{\underset{\sim}{S}_1} \cup \overline{\underset{\sim}{S}_2})$;
(j) $(\overline{\underset{\sim}{S}_1} \cap \overline{\underset{\sim}{S}_2})$

5. Consider the discrete fuzzy set defined on the universe $X = \{a, b, c, d, e, f\}$ as

$$\underline{B} = \left\{\frac{0}{a} + \frac{0.5}{b} + \frac{0.2}{c} + \frac{0.4}{d} + \frac{1}{e} + \frac{0.3}{f}\right\}$$

Using Zadeh's notation, find the λ-cut sets for the values $\lambda = 1, 0.7, 0.2, 0.4, 0^+$ and 0.

6. Determine the crisp λ-cut relation for $\lambda = 0.1, 0^+, 0.3, 0.6, 0.7, 1.0$ for the fuzzy relation given by

$$\underline{R} = \begin{bmatrix} 1 & 0 & 0.2 & 0.1 & 0.4 \\ 0.6 & 0.7 & 0.3 & 0.5 & 0 \\ 0.8 & 0.9 & 0.6 & 0.3 & 0.2 \\ 0.1 & 0 & 1 & 0.9 & 0.7 \end{bmatrix}$$

7. Consider the fuzzy relation

$$\underline{R} = \begin{bmatrix} 0.9 & 1.0 & 0 \\ 0.35 & 0.01 & 0.3 \\ 0.4 & 0.02 & 0.47 \\ 0.6 & 0.8 & 0.4 \\ 0.1 & 0 & 0.23 \\ 0.68 & 0.72 & 0.05 \end{bmatrix}$$

Find the λ-cut relation for $\lambda = 0^+, 0, 1, 0.5, 0.7$.

8. For the fuzzy relation

$$\underline{R} = \begin{bmatrix} 0.25 & 0.35 & 0.75 & 0.62 \\ 0 & 1 & 0.8 & 0.9 \\ 0.5 & 0.3 & 0.6 & 0.7 \\ 0.4 & 0 & 1 & 0.9 \end{bmatrix}$$

find the λ-cut relations for $\lambda = 0.3, 0.5, 0.9, 0.7$.

9. The fuzzy sets $\underline{A}, \underline{B}$ and $\underline{C}$ are all defined on the universe $X = [0, 5]$ with the following membership functions:

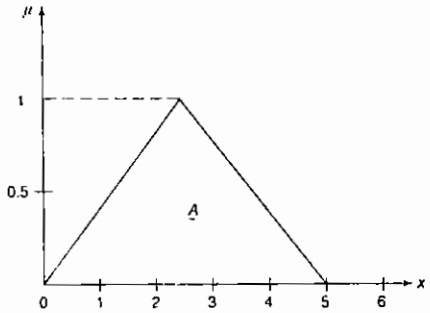
$$\mu_{\underline{A}}(x) = \frac{1}{1 + 2(x-2)^2},$$

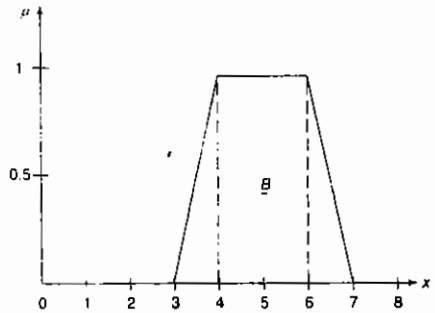
$$\mu_{\underline{B}}(x) = 3^{-x}, \mu_{\underline{C}}(x) = \frac{2x}{x+4}$$

(a) Sketch the membership functions.

(b) Define the intervals along the x-axis corresponding to the λ-cut sets for each of the fuzzy sets $\underline{A}, \underline{B}$ and $\underline{C}$ for $\lambda = 0.2, 0.4, 0.6, 0.9, 1.0$.

10. For the logical union of the membership functions shown below, find the defuzzified value x^* using each of the defuzzification methods.





Fuzzy Arithmetic and Fuzzy Measures

11

Learning Objectives

- Basic concepts of fuzzy arithmetic.
- How interval analysis is performed for uncertain values.
- A note on fuzzy numbers, fuzzy ordering and fuzzy vectors.
- Discusses on extension principle for generalizing crisp sets into fuzzy sets.
- A description on belief, plausibility, probability, possibility and necessity measures.
- Gives a view on fuzzy integrals.

11.1 Introduction

In this chapter, we will discuss the basic concepts involved in fuzzy arithmetic and fuzzy measures. Fuzzy arithmetic is based on the operations and computations of fuzzy numbers. Fuzzy numbers help in expressing fuzzy cardinalities and fuzzy quantifiers. Fuzzy arithmetic is applied in various engineering applications when only imprecise or uncertain sensory data are available for computation. In this chapter we will discuss various forms of fuzzy measures such as belief, plausibility, probability and possibility. A representation of uncertainty can be done using fuzzy measure. All the measures to be discussed are functions applied to crisp subsets, instead of elements, of a universal set.

11.2 Fuzzy Arithmetic

In the present scenario, we experience many applications which perform computation using ambiguous (imprecise) data. In all such cases, the imprecise data from the measuring instruments are generally expressed in the form of intervals, and suitable mathematical operations are performed over these intervals to obtain a reliable data of the measurements (which are also in the form of intervals). This type of computation is called *interval arithmetic* or *interval analysis*. Fuzzy arithmetic is a major concept in possibility theory. Fuzzy arithmetic is also a tool for dealing with fuzzy quantifiers in approximate reasoning (Chapter 12). Fuzzy numbers are an extension of the concept of intervals. Intervals are considered at only one unique level. Fuzzy numbers consider them at several levels varying from 0 to 1.

11.2.1 Interval Analysis of Uncertain Values

Consider a data set to be uncertain. We can locate this uncertain value to be lying on a real line, R, inside a closed interval, i.e., $x \in [a_1, a_2]$ where $a_1 \leq a_2$. The value of x is greater than or equal to a_1 and smaller than or equal to a_2. In interval analysis, the uncertainty of the data is limited between the intervals specified by the

Table 11-1 Set operations on intervals

Conditions	Union, $\cup$	Intersection, $\cap$
$a_1 > b_2$	$[b_1, b_2] \cup [a_1, a_2]$	ϕ
$b_1 > a_2$	$[a_1, a_2] \cup [b_1, b_2]$	ϕ
$a_1 > b_1, a_2 < b_2$	$[b_1, b_2]$	$[a_1, a_2]$
$b_1 > a_1, b_2 < a_2$	$[a_1, a_2]$	$[b_1, b_2]$
$a_1 < b_1 < a_2 < b_2$	$[a_1, b_2]$	$[b_1, a_2]$
$b_1 < a_1 < b_2 < a_2$	$[b_1, a_2]$	$[a_1, b_2]$

lower bound and upper bound. This can be represented as

$$\underset{\sim}{A} = [a_1, a_2] = \{x \mid a_1 \le x \le a_2\}$$

where $\underset{\sim}{A}$ represents an interval $[a_1, a_2]$. Generally, the values a_1 and a_2 are finite. In few cases, $a_1 = -\infty$ and/or $a_2 = +\infty$. If value of x is singleton in R then the interval form is $x = [x, x]$. In general, there are four types of intervals which are as follows:

1. $[a_1, a_2] = \{x \mid a_1 \le x \le a_2\}$ is a closed interval.
2. $[a_1, a_2) = \{x \mid a_1 \le x < a_2\}$ is an interval closed at the left end and open at right end.
3. $(a_1, a_2] = \{x \mid a_1 < x \le a_2\}$ is an interval open at left end and closed at right end.
4. $(a_1, a_2) = \{x \mid a_1 < x < a_2\}$ is an open interval, open at both left end and right end.

The set operations performed on the intervals are shown in Table 11-1. Here $[a_1, a_2]$ and $[b_1, b_2]$ are the upper bounds and lower bounds defined on the two intervals $\underset{\sim}{A}$ and $\underset{\sim}{B}$, respectively, i.e.,

$$\underset{\sim}{A} = [a_1, a_2], \quad \text{where } a_1 \le a_2$$
$$\underset{\sim}{B} = [b_1, b_2], \quad \text{where } b_1 \le b_2$$

The mathematical operations performed on intervals are as follows:

1. **Addition (+):** Let $\underset{\sim}{A} = [a_1, a_2]$ and $\underset{\sim}{B} = [b_1, b_2]$ be the two intervals defined. If $x \in [a_1, a_2]$ and $y \in [b_1, b_2]$, then

$$(x + y) \in [a_1 + b_1, a_2 + b_2]$$

This can be written as

$$\underset{\sim}{A} + \underset{\sim}{B} = [a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$$

2. **Subtraction (−):** The subtraction for the two intervals of confidence is given by

$$\underset{\sim}{A} - \underset{\sim}{B} = [a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1]$$

That is, we subtract the larger value out of b_1 and b_2 from a_1 and the smaller value out of b_1 and b_2 from a_2.

3. **Multiplication (·):** Let the two intervals of confidence be $\underset{\sim}{A} = [a_1, a_2]$ and $\underset{\sim}{B} = [b_1, b_2]$ defined on non-negative real line. The multiplication of these two intervals is given by

$$\underset{\sim}{A} \cdot \underset{\sim}{B} = [a_1, a_2] \cdot [b_1, b_2] = [a_1 \cdot b_1, a_2 \cdot b_2]$$

If we multiply an interval with a non-negative real number α, then we get

$$\alpha \cdot \underset{\sim}{A} = [\alpha, \alpha] \cdot [a_1, a_2] = [\alpha \cdot a_1, \alpha \cdot a_2]$$
$$\alpha \cdot \underset{\sim}{B} = [\alpha, \alpha] \cdot [b_1, b_2] = [\alpha \cdot b_1, \alpha \cdot b_2]$$

4. **Division (÷):** The division of two intervals of confidence defined on a non-negative real line is given by

$$\underset{\sim}{A} \div \underset{\sim}{B} = [a_1, a_2] \div [b_1, b_2] = \left[\frac{a_1}{b_2}, \frac{a_2}{b_1}\right]$$

If $b_1 = 0$ then the upper bound increases to $+\infty$. If $b_1 = b_2 = 0$, then interval of confidence is extended to $+\infty$.

5. **Image ($\overline{A}$):** If $x \in [a_1, a_2]$ then its image $-x \in [-a_2, -a_1]$. Also if $\underset{\sim}{A} = [a_1, a_2]$ then its image $\overline{A} = [-a_2, -a_1]$. Note that

$$\underset{\sim}{A} + \overline{\underset{\sim}{A}} = [a_1, a_2] + [-a_2, -a_1] = [a_1 - a_2, a_2 - a_1] \neq 0$$

That is, with image concept, the subtraction becomes addition of an image.

6. **Inverse (A^{-1}):** If $x \in [a_1, a_2]$ is a subset of a positive real line, then its inverse is given by

$$\left(\frac{1}{x}\right) \in \left[\frac{1}{a_2}, \frac{1}{a_1}\right]$$

Similarly, the inverse of $\underset{\sim}{A}$ is given by

$$\underset{\sim}{A}^{-1} = [a_1, a_2]^{-1} = \left[\frac{1}{a_2}, \frac{1}{a_1}\right]$$

That is, with inverse concept, division becomes multiplication of an inverse. For division by a non-negative number $\alpha > 0$, i.e. $(1/\alpha) \cdot \underset{\sim}{A}$, we obtain

$$\underset{\sim}{A} \div \alpha = \underset{\sim}{A} \cdot \left[\frac{1}{\alpha}, \frac{1}{\alpha}\right] = \left[\frac{a_1}{\alpha}, \frac{a_2}{\alpha}\right]$$

7. **Max and min operations:** Let two intervals of confidence be $\underset{\sim}{A} = [a_1, a_2]$ and $\underset{\sim}{B} = [b_1, b_2]$. Their max and min operations are defined by

$$\text{Max: } \underset{\sim}{A} \vee \underset{\sim}{B} = [a_1, a_2] \vee [b_1, b_2] = [a_1 \vee b_1, a_2 \vee b_2]$$
$$\text{Min: } \underset{\sim}{A} \wedge \underset{\sim}{B} = [a_1, a_2] \wedge [b_1, b_2] = [a_1 \wedge b_1, a_2 \wedge b_2]$$

The algebraic properties of the intervals are shown in Table 11-2.

Table 11-2 Algebraic properties of intervals

Property	Addition (+)	Multiplication (·)
Commutativity	$\underset{\sim}{A} + \underset{\sim}{B} = \underset{\sim}{B} + \underset{\sim}{A}$	$\underset{\sim}{A} \cdot \underset{\sim}{B} = \underset{\sim}{B} \cdot \underset{\sim}{A}$
Associativity	$(\underset{\sim}{A} + \underset{\sim}{B}) + \underset{\sim}{C} = \underset{\sim}{A} + (\underset{\sim}{B} + \underset{\sim}{C})$	$(\underset{\sim}{A} \cdot \underset{\sim}{B}) \cdot \underset{\sim}{C} = \underset{\sim}{A} \cdot (\underset{\sim}{B} \cdot \underset{\sim}{C})$
Neutral number	$\underset{\sim}{A} + 0 = 0 + \underset{\sim}{A} = \underset{\sim}{A}$	$\underset{\sim}{A} \cdot 1 = 1 \cdot \underset{\sim}{A} = \underset{\sim}{A}$
Image and inverse	$\underset{\sim}{A} + \overline{\underset{\sim}{A}} = \overline{\underset{\sim}{A}} + \underset{\sim}{A} \neq 0$	$\underset{\sim}{A} \cdot \underset{\sim}{A}^{-1} = \underset{\sim}{A}^{-1} \cdot \underset{\sim}{A} \neq 1$

11.2.2 Fuzzy Numbers

A fuzzy number is a normal, convex membership function on the real line R. Its membership function is piecewise continuous. That is, every λ-cut set $A_\lambda, \lambda \in [0, 1]$, of a fuzzy number $\underset{\sim}{A}$ is a closed interval of R and the highest value of membership of $\underset{\sim}{A}$ is unity. For two given fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$ in R, for a specific $\lambda_1 \in [0, 1]$, we obtain two closed intervals:

$$\underset{\sim}{A}_{\lambda_1} = \left[a_1^{(\lambda_1)}, a_2^{(\lambda_2)}\right] \text{ from fuzzy number } \underset{\sim}{A}$$

$$\underset{\sim}{B}_{\lambda_1} = \left[b_1^{(\lambda_1)}, b_2^{(\lambda_2)}\right] \text{ from fuzzy number } \underset{\sim}{B}$$

The interval arithmetic discussed can be applied to both these closed intervals. Fuzzy number is an extension of the concept of intervals. Instead of accounting intervals at only one unique level, fuzzy numbers consider them at several levels with each of these levels corresponding to each λ-cut of the fuzzy numbers. The notation $\underset{\sim}{A}_\lambda = [a_1^{(\lambda)}, a_2^{(\lambda)}]$ can be used to represent a closed interval of a fuzzy number $\underset{\sim}{A}$ at a λ-level.

Let us discuss the interval arithmetic for closed intervals of fuzzy numbers. Let $(*)$ denote an arithmeric operation, such as addition, subtraction, multiplication or division, on fuzzy numbers. The result $\underset{\sim}{A} * \underset{\sim}{B}$, where $\underset{\sim}{A}$ and $\underset{\sim}{B}$ are two fuzzy numbers is given by

$$\mu_{\underset{\sim}{A}*\underset{\sim}{B}}(z) = \bigvee_{z=x*y} [\mu_{\underset{\sim}{A}}(x), \mu_{\underset{\sim}{B}}(y)]$$

Using extension principle (see Section 11.3), where $x, y \in R$, for min $(\wedge)$ and max $(\vee)$ operation, we have

$$\mu_{\underset{\sim}{A}*\underset{\sim}{B}}(z) = \sup_{z=x*y} [\mu_{\underset{\sim}{A}}(x) * \mu_{\underset{\sim}{B}}(y)]$$

Using λ-cut, the above two equations become

$$(\underset{\sim}{A} * \underset{\sim}{B})_\lambda = \underset{\sim}{A}_\lambda * \underset{\sim}{B}_\lambda \text{ for all } \lambda \in [0, 1]$$

where $\underset{\sim}{A}_\lambda = [a_1^{(\lambda)}, a_2^{(\lambda)}]$ and $\underset{\sim}{B}_\lambda = [b_1^{(\lambda)}, b_2^{(\lambda)}]$. Note that for $a_1, a_2 \in [0, 1]$, if $a_1 > a_2$, then $\underset{\sim}{A}_{a_1} \subset \underset{\sim}{A}_{a_2}$.

On extending the *addition and subtraction* operations on intervals to two fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$ in R, we get

$$\underset{\sim}{A}_\lambda + \underset{\sim}{B}_\lambda = \left[a_1^\lambda + b_1^\lambda, a_2^\lambda + b_2^\lambda\right]$$

$$\underset{\sim}{A}_\lambda - \underset{\sim}{B}_\lambda = \left[a_1^\lambda - b_2^\lambda, a_2^\lambda - b_1^\lambda\right]$$

Similarly, on extending the *multiplication and division* operations on two fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$ in R^+ (non-negative real line) $= [0, \infty]$, we get

$$\underset{\sim}{A}_\lambda \cdot \underset{\sim}{B}_\lambda = \left[a_1^\lambda \cdot b_1^\lambda, a_2^\lambda \cdot b_2^\lambda\right]$$

$$\underset{\sim}{A}_\lambda \div \underset{\sim}{B}_\lambda = \left[\frac{a_1^\lambda}{b_2^\lambda}, \frac{a_2^\lambda}{b_1^\lambda}\right], \quad b_2^\lambda > 0$$

Table 11-3 Algebraic properties of addition and multiplication on fuzzy numbers

Property	Addition	Multiplication
Fuzzy numbers	$A, B, C \subset R$.	$A, B, C \subset R^+$
Commutativity	$A + B = B + A$	$A \cdot B = B \cdot A$
Associativity	$(A + B) + C = A + (B + C)$	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$
Neutral number	$A + 0 = 0 + A = A$	$A \cdot 1 = 1 \cdot A = A$
Image and inverse	$A + \overline{A} = \overline{A} + A \neq 0$	$A \cdot A^{-1} = A^{-1} \cdot A \neq 1$

The *multiplication* of a fuzzy number $\underset{\sim}{A} \subset R$ by an ordinary number $\beta \in R^+$ can be defined as

$$(\beta \cdot \underset{\sim}{A})_\lambda = \left[\beta a_1^\lambda, \beta a_2^\lambda\right]$$

The *support* for a fuzzy number, say $\underset{\sim}{A}$, is given by

$$\text{supp } \underset{\sim}{A} = \{x | \mu_{\underset{\sim}{A}}(x) > 0\}$$

which is an interval on the real line, denoted symbolically as A. The support of the fuzzy number resulting from the arithmetic operation $\underset{\sim}{A} * \underset{\sim}{B}$, i.e.,

$$\underset{\underset{\sim}{A}*\underset{\sim}{B}}{\text{supp}}(z) = A * B$$

is the arithmetic operation on the two individual supports, A and B, for fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$, respectively.

In general, arithmetic operations on fuzzy numbers based on λ-cut are given by (as mentioned earlier)

$$(\underset{\sim}{A} * \underset{\sim}{B})_\lambda = A_\lambda * B_\lambda$$

The algebraic properties of fuzzy numbers are listed in Table 11-3. The operations on fuzzy numbers possess the following properties as well.

1. If A and B are fuzzy numbers in R, then $(A + B)$ and $(A - B)$ are also fuzzy numbers. Similarly if A and B are fuzzy numbers in R^+, then $(A \cdot B)$ and $(A \div B)$ are also fuzzy numbers.
2. There exist no image and inverse fuzzy numbers, $\overline{A}$ and A^{-1}, respectively.
3. The inequalities given below stand true:

$$(A - B) + B \neq A \quad \text{and} \quad (A + B) \cdot B \neq A$$

11.2.3 Fuzzy Ordering

There exist several methods to compare two fuzzy numbers. The technique for fuzzy ordering is based on the concept of possibility measure.

For a fuzzy number $\underset{\sim}{A}$, two fuzzy sets $\underset{\sim}{A}_1$ and $\underset{\sim}{A}_2$ are defined. For this number, the set of numbers that are possibly greater than or equal to $\underset{\sim}{A}$ is denoted as $\underset{\sim}{A}_1$ and is defined as

$$\mu_{\underset{\sim}{A}_1}(w) = \prod\nolimits_{\underset{\sim}{A}}(-\infty, w) = \sup_{u \leq w} \mu_{\underset{\sim}{A}}(u)$$

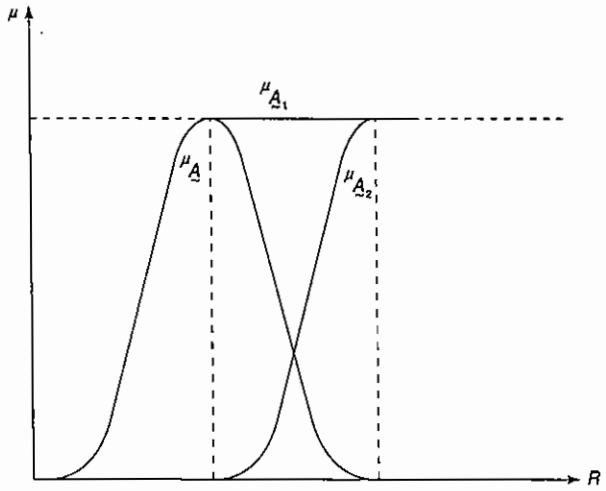


Figure 11-1 Fuzzy number $\underset{\sim}{A}$ and its associated fuzzy sets.

In a similar manner, the set of numbers that are necessarily greater than $\underset{\sim}{A}$ is denoted as $\underset{\sim}{A}_2$ and is defined as

$$\mu_{\underset{\sim}{A}_2}(u) = N_{\underset{\sim}{A}}(-\infty, u) = \inf_{u \geq w}[1 - \mu_{\underset{\sim}{A}}(u)]$$

where $\prod_A$ and N_A are possibility and necessity measures (see Section 11.4.3). Figure 11-1 shows the fuzzy number and its associated fuzzy sets $\underset{\sim}{A}_1$ and $\underset{\sim}{A}_2$.

When we try to compare two fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$ to check whether $\underset{\sim}{A}$ is greater than $\underset{\sim}{B}$, we split both the numbers into their associated fuzzy sets. We can compare $\underset{\sim}{A}$ with $\underset{\sim}{B}_1$ and $\underset{\sim}{B}_2$ by index of comparison such as the possibility or necessity measure of a fuzzy set. That is, we can calculate the possibility and necessity measures, in the set $\mu_{\underset{\sim}{A}}$, of fuzzy sets $\underset{\sim}{B}_1$ and $\underset{\sim}{B}_2$. On the basis of this, we obtain four fundamental indices of comparison which are given below.

1. $\prod_{\underset{\sim}{A}}(\underset{\sim}{B}_1) = \sup_u \min(\mu_{\underset{\sim}{A}}(u), \sup_{v \leq u} \mu_{\underset{\sim}{B}}(v)) = \sup_{u \geq v} \min(\mu_{\underset{\sim}{A}}(u), \mu_{\underset{\sim}{B}}(v))$
   This shows the possibility that the largest value X can take is at least equal to smallest value that Y can take.
2. $\prod_{\underset{\sim}{A}}(\underset{\sim}{B}_2) = \sup_u \min(\mu_{\underset{\sim}{A}}(u), \inf_{v \geq u}[1 - \mu_{\underset{\sim}{B}}(v)]) = \sup_u \inf_{v \geq u} \min(\mu_{\underset{\sim}{A}}(u), [1 - \mu_{\underset{\sim}{B}}(v)])$
   This shows the possibility that the largest value X can take is greater than the largest value that Y can take.
3. $N_{\underset{\sim}{A}}(\underset{\sim}{B}_1) = \inf_u \max(1 - \mu_{\underset{\sim}{A}}(v), \sup_{v \leq u} \mu_{\underset{\sim}{B}}(v)) = \inf_u \sup_{v \leq u} \max(1 - \mu_{\underset{\sim}{A}}(u), \mu_{\underset{\sim}{B}}(v))$
   This shows the possibility that the smallest value X can take is at least equal to smallest value that Y can take.
4. $N_{\underset{\sim}{A}}(\underset{\sim}{B}_2) = \inf_u \max(1 - \mu_{\underset{\sim}{A}}(u), \inf_{v \geq u}[1 - \mu_{\underset{\sim}{B}}(v)]) = 1 - \sup_{u \leq v} \min[\mu_{\underset{\sim}{A}}(u), \mu_{\underset{\sim}{B}}(v)]$
   This shows the possibility that the smallest value X can take is greater than the largest value that Y can take.

11.2.4 Fuzzy Vectors

A vector $\underset{\sim}{P} = (P_1, P_2, \ldots, P_n)$ is called a fuzzy vector if for any element we have $0 \leq P_i \leq 1$ for $i = 1$ to n. Similarly, the transpose of the fuzzy vector $\underset{\sim}{P}$, denoted by $\underset{\sim}{P}^{\mathrm{T}}$, is a column vector if $\underset{\sim}{P}$ is a row vector, i.e.,

$$\underset{\sim}{P}^{\mathrm{T}} = \begin{bmatrix} P_1 \\ P_2 \\ \cdot \\ \cdot \\ \cdot \\ P_n \end{bmatrix}$$

Let us define $\underset{\sim}{P}$ and $\underset{\sim}{Q}$ as fuzzy vectors of length n and $\underset{\sim}{P} \cdot \underset{\sim}{Q}^{\mathrm{T}} = \bigvee_{i=1}^{n} (P_i \wedge Q_i)$ as the fuzzy inner product of $\underset{\sim}{P}$ and $\underset{\sim}{Q}$. Then the fuzzy outer product of $\underset{\sim}{P}$ and $\underset{\sim}{Q}$ is defined by

$$\underset{\sim}{P} \oplus \underset{\sim}{Q}^{\mathrm{T}} = \bigwedge_{i=1}^{n} (P_i \vee Q_i)$$

The component of the fuzzy vector is defined as

$$\overline{\underset{\sim}{P}} = (1 - P_1, 1 - P_2, \ldots, 1 - P_n) = (\overline{P_1}, \overline{P_2}, \overline{P_3}, \ldots, \overline{P_n})$$

The fuzzy complement vector $\overline{\underset{\sim}{P}}$ has the constraint $0 \leq \overline{P_i} \leq 1$, for $i = 1$ to n, and it is also a fuzzy vector.

The largest component $\hat{P}$ in the fuzzy vector $\underset{\sim}{P}$ is defined as its upper bound, i.e.,

$$\hat{P} = \max_i(P_i)$$

The smallest component $\underset{\wedge}{P}$ of the fuzzy vector $\underset{\sim}{P}$ is defined by its lower bound, i.e.,

$$\underset{\wedge}{P} = \min_i(P_i)$$

The properties that the two fuzzy vectors $\underset{\sim}{P}$ and $\underset{\sim}{Q}$, both of length n, are given as follows:

1. $\overline{\underset{\sim}{P} \cdot \underset{\sim}{Q}^{\mathrm{T}}} = \overline{\underset{\sim}{P}} \oplus \overline{\underset{\sim}{Q}}^{\mathrm{T}}$
2. $\overline{\underset{\sim}{P} \oplus \underset{\sim}{Q}^{\mathrm{T}}} = \overline{\underset{\sim}{P}} \cdot \overline{\underset{\sim}{Q}}^{\mathrm{T}}$
3. $\underset{\sim}{P} \cdot \underset{\sim}{Q}^{\mathrm{T}} \leq (\hat{P} \wedge \hat{Q})$
4. $\underset{\sim}{P} \oplus \underset{\sim}{Q}^{\mathrm{T}} = (\underset{\wedge}{P} \vee \underset{\wedge}{Q})$
5. $\underset{\sim}{P} \cdot \underset{\sim}{P}^{\mathrm{T}} = \hat{P}$
6. $\underset{\sim}{P} \oplus \underset{\sim}{P}^{\mathrm{T}} \geq \underset{\wedge}{P}$
7. If $\underset{\sim}{P} \subseteq \underset{\sim}{Q}$ then $\underset{\sim}{P} \cdot \underset{\sim}{Q}^{\mathrm{T}} = \hat{P}$ and if $\underset{\sim}{Q} \subseteq \underset{\sim}{P}$ then $\underset{\sim}{P} \oplus \underset{\sim}{Q}^{\mathrm{T}} = \underset{\wedge}{P}$
8. $\underset{\sim}{P} \cdot \overline{\underset{\sim}{P}} \leq \frac{1}{2}$
9. $\underset{\sim}{P} \oplus \overline{\underset{\sim}{P}} \leq \frac{1}{2}$

It should be noted that when two separate fuzzy vectors are identical, i.e., $\underset{\sim}{P} = \underset{\sim}{Q}$, the inner product $\underset{\sim}{P}\underset{\sim}{Q}^{\mathrm{T}}$ reaches a maximum value while the outer product $\underset{\sim}{P} \oplus \underset{\sim}{Q}^{\mathrm{T}}$ reaches a minimum value.

11.3 Extension Principle

Extension principle was introduced by Zadeh in 1978 and is a very important tool of fuzzy set theory. This extension principle allows the generalization of crisp sets into the fuzzy set framework and extends point-to-point mappings to mappings for fuzzy sets. This principle allows any function f – that maps an n-tuple $(x_1, x_2, \ldots, x_n)$ in the crisp set U to a point in the crisp set V – to be generalized as a set that maps n fuzzy subsets in U to a fuzzy set in V. Thus, any mathematical relationship between nonfuzzy crisp elements can be extended to deal with fuzzy entities. The extension principle is also useful to deal with set-theoretic operations for higher order fuzzy sets.

Given a function $f: M \to N$ and a fuzzy set in M, where

$$\underset{\sim}{A} = \frac{\mu_1}{x_1} + \frac{\mu_2}{x_2} + \cdots + \frac{\mu_n}{x_n}$$

the extension principle states that

$$f(\underset{\sim}{A}) = f\left(\frac{\mu_1}{x_1} + \frac{\mu_2}{x_2} + \cdots + \frac{\mu_n}{x_n}\right) = \frac{\mu_1}{f(x_1)} + \frac{\mu_2}{f(x_2)} + \cdots + \frac{\mu_n}{f(x_n)}$$

If f maps several elements of M to the same element y in N (i.e., many-to-one mapping), then the maximum among their membership grades is taken. That is,

$$\mu_{f(\underset{\sim}{A})}(y) = \max_{\substack{x_i \in M \\ f(x_i) = y}} [\mu_{\underset{\sim}{A}}(x_i)]$$

where x_i's are the elements mapped to same element y. The function f maps n-tuples in M to a point in N.

Let M be the Cartesian product of universes $M = M_1 \times M_2 \times \cdots \times M_n$ and $\underset{\sim}{A}_1, \underset{\sim}{A}_2, \ldots, \underset{\sim}{A}_n$ be n fuzzy sets in $M_1, M_2, \ldots, M_n$, respectively. The function f maps an n-tuple $(x_1, x_2, \ldots, x_n)$ in the crisp set M to a point y in the crisp set V, i.e., $y = f(x_1, x_2, \ldots, x_n)$. The function $f(x_1, x_2, \ldots, x_n)$ to be extended to act on the n fuzzy subsets of $M, \underset{\sim}{A}_1, \underset{\sim}{A}_2, \ldots, \underset{\sim}{A}_n$ is permitted by the extension principle such that

$$\underset{\sim}{I} = f(\underset{\sim}{A})$$

where $\underset{\sim}{I}$ is the fuzzy image of $\underset{\sim}{A}_1, \underset{\sim}{A}_2, \ldots, \underset{\sim}{A}_n$ through $f(\cdot)$. The fuzzy set $\underset{\sim}{B}$ is defined by

$$\underset{\sim}{B} = \{(y, \mu_{\underset{\sim}{B}}(y)) | y = f(x_1, x_2, \ldots, x_n), (x_1, x_2, \ldots, x_n) \in M\}$$

where

$$\mu_{\underset{\sim}{B}}(y) = \sup_{\substack{(x_1, x_2, \ldots, x_n) \in M \\ y = f(x_1, x_2, \ldots, x_n)}} \min[\mu_{\underset{\sim}{A}_1}(x_1), \mu_{\underset{\sim}{A}_2}(x_2), \ldots, \mu_{\underset{\sim}{A}_n}(x_n)]$$

with a condition that $\mu_{\underset{\sim}{B}}(y) = 0$ if there exists no $(x_1, x_2, \ldots, x_n) \in M$ such that $y = f(x_1, x_2, \ldots, x_n)$.

The extension principle helps in propagating fuzziness through generalized relations that are discrete mappings of ordered pairs of elements from input universes to ordered pairs of elements from other universe. The extension principle is also useful for mapping fuzzy inputs through continuous-valued functions. The process employed is same as for a discrete-valued function, but it involves more computation.

11.4 Fuzzy Measures

A fuzzy measure explains the imprecision or ambiguity in the assignment of an element a to two or more crisp sets. For representing uncertainty condition, known as ambiguity, we assign a value in the unit interval [0, 1] to each possible crisp set to which the element in the problem might belong. The value assigned represents the degree of evidence or certainty or belief of the element's membership in the set. The representation of uncertainty of this manner is called fuzzy measure. In sum, a fuzzy measure assigns a value in the unit interval [0, 1] to each classical set of the universal set signifying the degree of belief that a particular element x belongs to the crisp set. In this section several different fuzzy measures such as belief measures, plausibility measure, probability measure, necessity measure and possibility measure are covered. All these measures are functions applied to crisp subsets, instead of elements of a universal set.

The difference between a fuzzy measure and a fuzzy set on a universe of elements is that, in fuzzy measure, the imprecision is in the assignment of an element to one of two or more crisp sets, and in fuzzy sets, the imprecision is in the prescription of the boundaries of a set.

A fuzzy measure is defined by a function

$$g: P(X) \to [0, 1]$$

which assigns to each crisp subset of a universe of discourse X a number in the unit interval [0, 1], where $P(X)$ is power set of X. A fuzzy measure is obviously a set function. To qualify a fuzzy measure, the function g should possess certain properties. A fuzzy measure is also described as follows:

$$g: B \to [0, 1]$$

where $B \subset P(X)$ is a family of crisp subsets of X. Here B is a Borel field or a σ field. Also, g satisfies the following three axioms of fuzzy measures:

Axiom 1: Boundary Conditions (g1)

$$g(\phi) = 0; g(X) = 1$$

Axiom 2: Monotonicity (g2) – For every classical set $A, B \in P(X)$, if $A \subseteq B$, then $g(A) \leq g(B)$.

Axiom 3: Continuity (g3) – For each sequence $(A_i \in P(X) | i \in N)$ of subsets of X, if either $A_1 \subseteq A_2 \subseteq \ldots$ or $A_1 \supseteq A_2 \supseteq \ldots$, then

$$\lim_{i \to \infty} g(A_i) = g\left(\lim_{i \to \infty} A_i\right)$$

where N is the set of all positive integers.

A σ field or Borel field satisfies the following properties:

1. $X \in B$ and $\phi \in B$.
2. If $A \in B$, then $\overline{A} \in B$.
3. B is closed under set union operation, i.e., if $A \in B$ and $B \in B$ (σ field), then $A \cup B \in B$ (σ field)

The fuzzy measure excludes the additive property of standard measures, h. The additive property states that when two sets A and B are disjoint, then

$$h(A \cup B) = h(A) + h(B)$$

The probability measure possesses this additive property. Fuzzy measures are also defined by another weaker axiom: subadditivity. The other basic properties of fuzzy measures are the following:

1. Since $\underset{\sim}{A} \subseteq \underset{\sim}{A} \cup \underset{\sim}{B}$ and $\underset{\sim}{B} \subseteq \underset{\sim}{A} \cup \underset{\sim}{B}$, and because fuzzy measure g possesses monotonic property, we have

$$g(\underset{\sim}{A} \cup \underset{\sim}{B}) \geq \max[g(\underset{\sim}{A}), g(\underset{\sim}{B})]$$

2. Since $\underset{\sim}{A} \cap \underset{\sim}{B} \subseteq \underset{\sim}{A}$ and $\underset{\sim}{A} \cap \underset{\sim}{B} \subseteq \underset{\sim}{B}$, and because fuzzy measure g possesses monotonic property, we have

$$g(\underset{\sim}{A} \cap \underset{\sim}{B}) \leq \min[g(\underset{\sim}{A}), g(\underset{\sim}{B})]$$

11.4.1 Belief and Plausibility Measures

The belief measure is a fuzzy measure that satisfies three axioms g1, g2 and g3 and an additional axiom of subadditivity. A belief measure is a function

$$\text{bel} : B \to [0, 1]$$

satisfying axioms g1, g2 and g3 of fuzzy measures and subadditivity axiom. It is defined as follows:

$$\text{bel}(A_1 \cup A_2 \cup \cdots \cup A_n) \geq \sum_i \text{bel}(A_i) - \sum_{i<j} \text{bel}(A_i \cap A_j) + \cdots + (-1)^{n-1} \text{bel}(A_1 \cap A_2 \cap \cdots \cap A_n)$$

for every $n \in N$ and every collection of subsets of X. N is set of all positive integer. This is called axiom 4 (g4).

For $n = 2$, g4 is of the form

$$\text{bel}(A_1 \cup A_2) \geq \text{bel}(A_1) + \text{bel}(A_2) - \text{bel}(A_1 \cap A_2)$$

For $n = 2$, if $A_1 = A$ and $A_2 = \overline{A}$, axiom g4 indicates

$$\text{bel}(A_1 \cup A_2) = \text{bel}(A \cup \overline{A})$$
$$\text{bel}(A \cup \overline{A}) \geq \text{bel}(A) + \text{bel}(\overline{A}) - \text{bel}(A \cap \overline{A})$$

Since $A \cup \overline{A} = X$ and $A \cap \overline{A} = \phi$, we have

$$\text{bel}(X) \geq \text{bel}(A) + \text{bel}(\overline{A})$$
$$\text{bel}(A) + \text{bel}(\overline{A}) \leq 1$$

On the basis of the belief measure, one can define a plausibility measure Pl as

$$\text{Pl}(A) = 1 - \text{bel}(\overline{A})$$

for all $A \in B(\text{CP}(X))$. On the other hand, based on plausibility measure, belief measure can be defined as

$$\text{bel}(A) = 1 - \text{Pl}(\overline{A})$$

Plausibility measure can also be defined independent of belief measure. A plausibility measure is a function

$$\text{Pl} : B \to [0, 1]$$

satisfying axioms g1, g2, g3 of fuzzy measures and the following additional subadditivity axiom (axiom g5):

$$\text{Pl}(A_1 \cap A_2 \cap \cdots \cap A_n) \leq \sum_i \text{Pl}(A_i) - \sum_{i<j} \text{Pl}(A_i \cup A_j) + \cdots + (-1)^{n-1} \text{Pl}(A_1 \cup A_2 \cup \cdots \cup A_n)$$

for every $n \in N$ and all collection of subsets of X. For $n = 2$, consider $A_1 = A$ and $A_2 = \overline{A}$, then we have

$$\text{Pl}(A \cap \overline{A}) \leq \text{Pl}(A) + \text{Pl}(\overline{A}) - \text{Pl}(A \cup \overline{A})$$
$$\Rightarrow \quad \text{Pl}(A) + \text{Pl}(\overline{A}) \geq 1$$

The belief measure and the plausibility measure are mutually dual, so it will be beneficial to express both of them in terms of a set function m, called a basic probability assignment. The basic probability assignment m is a set function,

$$m : B \to [0, 1]$$

such that $m(\phi = 0)$ and $\sum_{A \in B} m(A) = 1$. The basic probability assignments are not fuzzy measures. The quantity $m(A) \in [0, 1], A \in B(\text{CP}(X))$, is called A's basic probability number. Given a basic assignment m, a belief measure and a plausibility measure can be uniquely determined by

$$\text{bel}(A) = \sum_{B \subseteq A} m(B)$$

$$\text{Pl}(A) = \sum_{B \cap A \neq 0} m(B)$$

for all $A \in B(\text{CP}(X))$.

The relations among $m(A)$, bel(A) and Pl(A) are as follows:

1. $m(A)$ measures the belief that the element $(x \in X)$ belongs to set A alone, not the total belief that the element commits in A.
2. bel(A) indicates total evidence that the element $(x \in X)$ belongs to set A and to any other special subsets of A
3. Pl(A) includes the total evidence that the element $(x \in X)$ belongs to set A or to other special subsets of A plus the additional evidence or belief associated with sets that overlap with A.

Based on these relations, we have

$$\text{Pl}(A) \geq \text{bel}(A) \geq m(A) \quad \forall A \in B\ (\sigma \text{ field})$$

Belief and plausibility measure are dual to each other. The corresponding basic assignment m can be obtained from a given plausibility measure Pl:

$$m(A) = \sum_{B \subseteq A} (-1)^{(A-B)} [1 - \text{Pl}(\overline{B})] \quad \forall A \in B\ (\sigma \text{ field})$$

Every set $A \in B(\text{CP}(X))$ for which $m(A) > 0$ is called a focal element of m. Focal elements are subsets of X on which the available evidence focuses.

11.4.2 Probability Measures

On replacing the axiom of subadditivity (axiom g4) with a stronger axiom of additivity (axiom g6),

$$\text{bel}(A \cup B) = \text{bel}(A) + \text{bel}(B) \text{ whenever } A \cap B = \phi\,;\, A, B \in B(\sigma \text{ field})$$

we get the crisp probability measures (or Bayesian belief measures). In other words, the belief measure becomes the crisp probability measure under the additive axiom.

A probability measure is a function

$$P : B \to [0, 1]$$

satisfying the three axioms g1, g2 and g3 of fuzzy measures and the additivity axiom (axiom g6) as follows:

$$P(A \cup B) = P(A) + P(B) \text{ whenever } A \cap B = \phi\,, A, B \in B$$

With axiom g6, the theorem given below relates the belief measure and the basic assignment to the probability measure.

"A belief measure bel on a finite σ-field B, which is a subset of P(X), is a probability measure if and only if its basic probability assignment m is given by $m(\{x\}) = \text{bel}(\{x\})$ and $m(A) = 0$ for all subsets of X that are not singletons."

The theorem mentioned is very significant. The theorem indicates that a probability measure on finite sets can be represented uniquely by a function defined on the elements of the universal set X rather than its subsets. The probability measures on finite sets can be fully represented by a function,

$$P : X \to [0, 1] \quad \text{such that} \quad P(x) = m(\{x\})$$

This function $P(X)$ is called probability distribution function. Within probability measure, the total ignorance is expressed by the uniform probability distribution function

$$P(x) = m(\{x\}) = \frac{1}{|X|} \text{ for all } x \in X$$

The plausibility and belief measures can be viewed as upper and lower probabilities that characterize a set of probability measures.

11.4.3 Possibility and Necessity Measures

In this section, let us discuss two subclasses of belief and plausibility measures, which focus on nested focal elements. A group of subsets of a universal set is nested if these subsets can be ordered in a way that each is contained in the next; i.e., $A_1 \subset A_2 \subset A_3 \subset \cdots \subset A_n, A_i \in P(X)$ are nested sets. When the focal elements of a body of evidence (E, m) are nested, the linked belief and plausibility measures are called consonants, because here the degrees of evidence allocated to them do not conflict with each other. The belief and plausibility measures are characterized by the following theorem:

Theorem: Consider a consonant body of evidence (E, m), the associated consonant belief and plausibility measures posses the following properties:

$$\text{bel}(A \cap B) = \min[\text{bel}(A), \text{bel}(B)]$$
$$\text{Pl}(A \cup B) = \max[\text{Pl}(A), \text{Pl}(B)]$$

for all $A, B \in B(\text{CP}(X))$.

Consonant belief and plausibility measures are referred to as necessity and possibility measures and are denoted by N and $\prod$, respectively. The possibility and necessity measures are defined independently as follows:

The possibility measure $\prod$ and necessity measure N are functions

$$\prod : B \to [0, 1]$$
$$N : B \to [0, 1]$$

such that both $\prod$ and N satisfy the axioms g1, g2 and g3 of fuzzy measures and the following additional axiom (g7):

$$\prod(A \cup B) = \max(\prod(A), \prod(B)) \qquad \forall A, B \in B$$
$$N(A \cap B) = \min(N(A), N(B)) \qquad \forall A, B \in B$$

As necessity and possibility measures are special subclasses of belief and plausibility measures, respectively, they are related to each other by

$$\prod(A) = 1 - N(\overline{A})$$
$$N(A) = 1 - \prod(\overline{A}) \quad \forall A \in \sigma \text{ field}$$

The properties given below are based on the axiom *g7* and above set of equations.

1. $\min[N(A), N(\overline{A})] = N(A \cap \overline{A}) = 0$. This implies that A or $\overline{A}$ is not necessary at all.
2. $\max[\prod(A), \prod(\overline{A})] = \prod(A \cup \overline{A}) = \prod(X) = 1$. This implies that either A or $\overline{A}$ is completely possible.
3. $\prod(A) \geq N(A)\ \forall A \subseteq \sigma$ field.
4. If $N(A) > 0$ then $\prod(A) = 1$ and if $\prod(A) < 1$ then $N(A) = 0$.

The two equations indicate that if an event is necessary then it is completely possible. If it is not completely possible then it is not necessary. Every possibility measure $\prod$ on $B \subset P(x)$ can be uniquely determined by a possibility distribution function

$$\prod : x \to [0, 1]$$

using the formula

$$\prod(A) = \max_{x \in A} \prod(x) \qquad \forall x \in \sigma \text{ field}$$

The necessity and possibility measure are mutually dual with each other. As a result we can obtain the necessity measure from the possibility distribution function. This is given as

$$N(A) = 1 - \prod(\overline{A}) = 1 - \max_{x \notin A} \prod(x)$$

The total ignorance can be expressed in terms of the possibility distribution by $\prod(x_n) = 1$ and $\prod(x_i) = 0$ for $i = 1$ to $n - 1$, corresponding to $\prod(A_n) = \prod(X) = 1$ and $\prod(A) = 0$.

11.5 Measures of Fuzziness

The concept of fuzzy sets is a base frame for dealing with vagueness. In particular, the fuzzy measures concept provides a general mathematical framework to deal with ambiguous variables. Thus, fuzzy sets and fuzzy measures are tools for representing these ambiguous situations. Measures of uncertainty related to vagueness are referred to as measures of fuzziness.

Generally, a measure of fuzziness is a function

$$f : P(X) \to R$$

where R is the real line and $P(X)$ is the set of all fuzzy subsets of X. The function f satisfies the following axioms:

1. *Axiom 1* (f1): $f(A) = 0$ if and only if A is a crisp set.
2. *Axiom 2* (f2): If A (shp) B, then $f(A) \leq f(B)$, where A (shp) B denotes that A is sharper than B.
3. *Axiom 3* (f3): $f(A)$ takes the maximum value if and only if A is maximally fuzzy.

Axiom f1 shows that a crisp set has zero degree of fuzziness in it. Axioms f2 and f3 are based on concept of "sharper" and "maximal fuzzy," respectively.

1. The first fuzzy measure can be defined by the function:

$$f(A) = -\sum_{x \in A} \{\mu_A(x) \log_2 [\mu_A(x)]\ [1-\mu_A(x)] \log_2 [1-\mu_A(x)]\}$$

It can be normalized as

$$f'(A) = \frac{f(A)}{|x|}$$

where $|x|$ is cardinality of universal set X. This measure of fuzziness can be considered as the entropy of a fuzzy set

2. A (shp) B, A is sharper than B, is defined as

$$\mu_A(x) \leq \mu_B(x) \quad \text{for } \mu_B \leq 0.5$$
$$\mu_A(x) \geq \mu_B(x) \quad \text{for } \mu_B(x) \geq 0.5 \quad \forall x \in X$$

3. A is maximally fuzzy if

$$\mu_A(x) = 0.5 \qquad \text{for all } x \in X$$

11.6 Fuzzy Integrals

Sugeno in the year 1977 defined fuzzy integral using fuzzy measures based on a Lebesque integral, which is defined using "measures."

Let K be a mapping from X to $[0,1]$. The fuzzy integral, in the sense of fuzzy measure g, of K over a subset A of X is defined as

$$\int_A K(x) \cdot g = \sup_{\alpha \in [0,1]} \min[\beta, g(A \cap H_\beta)]$$

where $H_\beta = \{x \in x | K(x) \geq \beta\}$. Here, A is called the domain of integration. If $k = a \in [0,1]$ is a constant, then its fuzzy integral over X is "a" itself, because $g(X \cap H_\beta) = 1$ for $\beta \leq a$ and $g(X \cap H_\beta) = 0$ for $\beta > a$, i.e.,

$$\int_x a \bullet g = a, \qquad a \in [0,1]$$

Consider X to be a finite set such that $X = \{x_1, x_2, \ldots, x_n\}$. Without loss of generality, assuming the function to be integrated, k can be obtained such that $k(x_1) \geq k(x_2) \geq \cdots \geq k(x_n)$. This is obtained after proper ordering. The basic fuzzy integral then becomes

$$\int_X k(x) \cdot g = \max_{i=1 \text{ to } n} \min[k(x_i), g(H_i)]$$

where $H_i = \{x_1, x_2, \ldots, x_i\}$. The calculation of the fuzzy measure "g" is a fundamental point in performing a fuzzy integration.

11.7 Summary

In this chapter we discussed *fuzzy arithmetic*, which is considered as an extension of interval arithmetic. The chapter provides a general methodology for extending crisp concepts to address fuzzy quantities, such as real algebraic operations on fuzzy numbers. One of the important tools of fuzzy set theory introduced by Zadeh is the extension principle, which allows any mathematical relationship between nonfuzzy elements to be extended to fuzzy entities. This principle can be applied to algebraic operations to define set-theoretic operations for higher order fuzzy sets. The operations and properties of fuzzy vectors were discussed in this chapter for their use in similarity metrics. Also, we have discussed the concept of fuzzy measures and the axioms that must be satisfied by a set function in order for it to be a fuzzy measure. We also discuss *belief and plausibility measures* which are based on the dual axioms of subadditivity. The belief and plausibility measures can be expressed by the basic probability assignment m, which assigns degree of evidence or belief indicating that a particular element of X belongs only to set A and not to any subset of A. Focal elements are the subsets that are assigned with nonzero degrees of evidence. The main characteristic of *probability measures* is that each of them can be distinctly represented by a probability distribution function defined on the elements of a universal set apart from its subsets. Also the *necessity and possibility measures*, which are consonant belief measures and consonant plausibility measures, respectively, are characterized distinctly by functions defined on the elements of the universal set rather than on its subsets. The fuzzy integrals defined by Sugeno (1977) are also discussed. Fuzzy integrals are used to perform integration of fuzzy functions. The measures of fuzziness were also discussed. The definitions of measures of fuzziness dealt in this chapter can be extended to noninfinite supports by replacing the summation by integration appropriately.

11.8 Solved Problems

1. Perform the following operations on intervals:
   (a) $[3,2] + [4,3]$ (b) $[2,1] \times [1,3]$
   (c) $[4,6] \div [1,2]$ (d) $[3,5] - [4,5]$

Solution: The operations were performed on the basis of the interval analysis.

(a) $[3,2] + [4,3] = [a_1, a_2] + [b_1, b_2]$
$= [a_1 + b_1, a_2 + b_2]$
$= [3+4, 2+3] = [7,5]$

(b) $[2,1] \times [1,3] = [a_1, a_2] \cdot [b_1, b_2]$
$= [a_1 \cdot b_1, a_2 \cdot b_2]$
$= [2 \cdot 1, 1 \cdot 3] = [2,3]$

(c) $[4,6] \div [1,2] = [a_1, a_2] \div [b_1, b_2]$

$$= \left[\frac{a_1}{b_2}, \frac{a_2}{b_1}\right]$$

$$= \left[\frac{4}{2}, \frac{6}{1}\right] = [2,6]$$

(d) $[3,5] - [4,5] = [a_1, a_2] - [b_1, b_2]$

$$= [a_1 - b_2, a_2 - b_1]$$

$$= [3-5, 5-4] = [-2, 1]$$

2. For the interval $\underset{\sim}{A} = [5,3]$, find its image and inverse.

Solution: The given interval is

$$\underset{\sim}{A} = [5,3] = [a_1, a_2]$$

(a) Image $\overline{A} = [-a_2, -a_1] = [-3, -5]$

(b) Inverse $A^{-1} = \left[\frac{1}{a_2}, \frac{1}{a_1}\right] = \left[\frac{1}{3}, \frac{1}{5}\right]$

$$= [0.333,\ 0.2]$$

3. Given the two intervals $\underset{\sim}{E} = [2,4]$, $\underset{\sim}{F} = [-4,5]$, perform the max and min operations over these intervals.

Solution: The given intervals are $\underset{\sim}{E} = [a_1, a_2] = [2,4]$ and $\underset{\sim}{F} = [b_1, b_2] = [-4,5]$.

(a) Max operation

$$\underset{\sim}{E} \vee \underset{\sim}{F} = [a_1, a_2] \vee [b_1, b_2] = [a_1 \vee b_1, a_2 \vee b_2]$$

$$= [2 \vee -4, 4 \vee 5] = [2,5]$$

(b) Min operation

$$\underset{\sim}{E} \wedge \underset{\sim}{F} = [a_1, a_2] \wedge [b_1, b_2] = [2,4] \wedge [-4,5]$$

$$= [2 \wedge -4, 4 \wedge 5] = [-4,4]$$

4. Consider a fuzzy number $\underset{\sim}{1}$, the normal convex membership function defined on integers

$$\underset{\sim}{1} = \left\{\frac{0.5}{0} + \frac{1}{1} + \frac{0.5}{2}\right\}$$

Perform addition of two fuzzy numbers, i.e., add $\underset{\sim}{1}$ to $\underset{\sim}{1}$ using extension principle.

Solution:

$$\underset{\sim}{1} + \underset{\sim}{1} = \left(\frac{0.5}{0} + \frac{1}{1} + \frac{0.5}{2}\right) + \left(\frac{0.5}{0} + \frac{1}{1} + \frac{0.5}{2}\right)$$

$$\underset{\sim}{2} = \left\{\frac{\min(0.5, 0.5)}{0} + \frac{\max[\min(0.5,1), \min(1,0.5)]}{1} + \frac{\max[\min(0.5,0.5), \min(1,1), \min(0.5,0.5)]}{2} + \frac{\max[\min(1,0.5), \min(0.5,1)]}{3} + \frac{\min(0.5,0.5)}{4}\right\}$$

$$= \left\{\frac{0.5}{0} + \frac{\max[0.5,0.5]}{1} + \frac{\max[0.5,1,0.5]}{2} + \frac{\max[0.5,0.5]}{3} + \frac{\min(0.5,0.5)}{4}\right\}$$

$$\underset{\sim}{2} = \left\{\frac{0.5}{0} + \frac{0.5}{0} + \frac{1}{2} + \frac{0.5}{3} + \frac{0.5}{4}\right\}$$

5. The two fuzzy vectors of length 4 are defined as

$$\underset{\sim}{a} = (0.5, 0.2, 1.0, 0.8)$$

and $$\underset{\sim}{b} = (0.8, 0.1, 0.9, 0.3)$$

Find the inner product and outer product for these two fuzzy vectors.

Solution:

(a) *Inner product:*

$$\underset{\sim}{a} \cdot \underset{\sim}{b}^T = (0.5, 0.2, 1.0, 0.8) \begin{pmatrix} 0.8 \\ 0.1 \\ 0.9 \\ 0.3 \end{pmatrix}$$

$$= (0.5 \wedge 0.8) \vee (0.2 \wedge 0.1) \vee (1.0 \wedge 0.9) \vee (0.8 \wedge 0.3)$$

$$= 0.5 \vee 0.1 \vee 0.9 \vee 0.3 = 0.9$$

(b) *Outer product:*

$$\underset{\sim}{a} \oplus \underset{\sim}{b}^T = (0.5, 0.2, 1.0, 0.8) \begin{pmatrix} 0.8 \\ 0.1 \\ 0.9 \\ 0.3 \end{pmatrix}$$

$$= (0.5 \vee 0.8) \wedge (0.2 \vee 0.1) \wedge (1.0 \vee 0.9) \wedge (0.8 \vee 0.3)$$

$$= (0.8) \wedge (0.2) \wedge (1.0) \wedge (0.8) = 0.2$$

6. Let X be the universal set and let A, B, and C be the subsets of X. The basic assignments for the corresponding focal elements are mentioned in the following table. Determine the corresponding belief measure.

Focal elements	$m(\cdot)$
P	0.04
B	0.04
E	0.04
$P \cup B$	0.12
$P \cup E$	0.08
$B \cup E$	0.04
$P \cup B \cup E$	0.64

Solution: The belief measures are obtained as follows:

$$\text{bel}(P) = m(P) = 0.04$$

$$\text{bel}(B) = m(B) = 0.04$$

$$\text{bel}(E) = m(E) = 0.04$$

$$\text{bel}(P \cup B) = m(P \cup B) + m(P) + m(B)$$

$$= 0.12 + 0.04 + 0.04 = 0.2$$

$$\text{bel}(P \cup E) = m(P \cup E) + m(P) + m(E)$$

$$= 0.08 + 0.04 + 0.04 = 0.16$$

$$\text{bel}(B \cup E) = m(B \cup E) + m(B) + m(E)$$

$$= 0.04 + 0.04 + 0.04 = 0.12$$

$$\text{bel}(P \cup B \cup E) = m(P \cup B \cup E) + m(P \cup B) + m(P \cup E) + m(B \cup E) + m(P) + m(B) + m(E)$$

$$= 0.64 + 0.12 + 0.08 + 0.04 + 0.04 + 0.04 + 0.04$$

$$= 1.0$$

11.9 Review Questions

1. State the importance of fuzzy arithmetic.
2. How is an interval analysis obtained in fuzzy arithmetic?
3. List the set operations performed on intervals.
4. Discuss the mathematical operations performed on intervals.
5. What are the properties of performing addition and multiplication on intervals?
6. Define fuzzy numbers.
7. Mention the properties of addition and multiplication on fuzzy numbers.
8. Write short note on fuzzy ordering.
9. Explain in detail the concept of fuzzy vectors.
10. State the extension principle in fuzzy set theory.
11. What are fuzzy measures?
12. Explain in detail the belief and plausibility measures.
13. How are necessity and possibility measures obtained from belief and plausibility measures?
14. Discuss in detail:
    - Probability measure;
    - Fuzzy integrals.
15. Mention the measures of fuzziness in detail.

11.10 Exercise Problems

1. Perform the following operations on intervals
   (a) $[5,3]+[4,2]$ (b) $[6,9]-[2,4]$
   (c) $[1,2]\times[5,3]$ (d) $[7,3]\div[3,6]$
   (e) $[5,3]$ (f) $[6,5]^{-1}$
2. Perform the max and min operations over the intervals $\underset{\sim}{E}=[5,6]$ and $\underset{\sim}{G}=[9,2]$.
3. Given the following fuzzy numbers and using Zadeh's extension principle, calculate $\underset{\sim}{K}=\underset{\sim}{A}\cdot\underset{\sim}{B}$ and show why $\underset{\sim}{1}0$ is nonconvex.

$$\underset{\sim}{A}=\underset{\sim}{2}=\frac{0.2}{2}+\frac{1}{3}+\frac{0.1}{4}$$
$$\underset{\sim}{B}=\underset{\sim}{2}=\frac{0.1}{1}+\frac{1}{2}+\frac{0.2}{3}$$

4. Given

$$\underset{\sim}{A}=\frac{0.4}{0.2}+\frac{1}{0.4}+\frac{0.4}{0.6}$$
$$\underset{\sim}{B}=\frac{1}{0.2}+\frac{0.4}{0.4}+\frac{0.5}{0.6}$$

   calculate the following: $\underset{\sim}{A}+\underset{\sim}{B}$, $\underset{\sim}{A}-\underset{\sim}{B}$, $\underset{\sim}{A}*\underset{\sim}{B}$, $\underset{\sim}{A}\div\underset{\sim}{B}$.
5. For the two triangular fuzzy numbers $\underset{\sim}{A}$ and $\underset{\sim}{B}$, whose membership functions are respectively

$$\mu_{\underset{\sim}{A}}(x)=\begin{cases}2-x & \text{if } -1\le x\le 0\\ \dfrac{x-2}{5} & \text{if } 0\ge x\ge 0\\ 0 & \text{otherwise}\end{cases}$$

$$\mu_{\underset{\sim}{B}}(x)=\begin{cases}x+1 & \text{if } -1\le x\le 0\\ \dfrac{3-x}{5} & \text{if } 0\ge x\ge 0\\ 0 & \text{otherwise}\end{cases}$$

   compute the following:
   (a) $\underset{\sim}{A}+\underset{\sim}{B}$, $\underset{\sim}{A}-\underset{\sim}{B}$
   (b) $\underset{\sim}{A}\wedge\underset{\sim}{B}$, $\underset{\sim}{A}\vee\underset{\sim}{B}$
   (c) $\underset{\sim}{A}\div\underset{\sim}{B}$, $\underset{\sim}{A}\div\underset{\sim}{B}$
6. Consider the three fuzzy sets $\underset{\sim}{A}$, $\underset{\sim}{B}$ and $\underset{\sim}{C}$ and their membership functions:

$$\mu_{\underset{\sim}{A}}(x)=\frac{1}{1+10x},\quad \mu_{\underset{\sim}{B}}(x)=\left(\frac{1}{1+x}\right),$$
$$\mu_{\underset{\sim}{C}}(x)=\left(\frac{1}{1+2x}\right)^{0.5}$$

   Order the fuzzy sets. Take $x\ge 0$.
7. The two fuzzy vectors of length 6 are defined as

$$\underset{\sim}{a}=(0.5,\ 0.7,\ 0.2,\ 0.3,\ 1,\ 0.8)$$
$$\underset{\sim}{b}=(0,\ 0.2,\ 0.1,\ 0.4,\ 0.6,\ 1.0)$$

   Find the inner product and outer product of two vectors.
8. Determine the corresponding belief and plausibility measures from the table below:

Focal elements	m
P	0.05
B	0.05
E	0.05
$P\cup B$	0.50
$P\cup E$	0.15
$B\cup E$	0.05
$P\cup B\cup E$	0.15

9. Consider the possibility distribution induced by the proposition "x is an even integer" is

$$\prod_X=\{(1,1),(2,3),(3,0.5),(4,0.4),$$
$$(5,0.6),(6,0.3)\}$$

   If $A=\{1,2,3\}$ is a crisp set, then find the possibility and necessity measures of A.
10. With suitable example, show that the maximum measure of fuzziness is $|X|$.

Fuzzy Rule Base and Approximate Reasoning

12

Learning Objectives

- Discusses on various fuzzy propositions.
- This chapter gives an idea of how to form the fuzzy rules, decompose and aggregate them.
- Different modes of fuzzy approximate reasoning.
- A note on fuzzy inference system and its types.
- An overview of fuzzy expert system.

12.1 Introduction

This chapter focuses on formation of fuzzy rules and reasoning. The degree of an element in a fuzzy set corresponds to the truth value of a proposition in fuzzy logic systems. The chapter continues with using natural language in the expression of various knowledge forms; such systems are known as rule-based systems. Thereafter we address concepts such as formation, decomposition and aggregation of fuzzy rules. We explore and discuss not only the different modes of fuzzy reasoning but also introduce the basic concepts of fuzzy inference system, along with its two different types. The chapter closes with a basic overview of fuzzy expert system.

12.2 Truth Values and Tables in Fuzzy Logic

Fuzzy logic uses linguistic variables. The values of a linguistic variable are words or sentences in a natural or artificial language. For example, height is a linguistic variable if it takes values such as tall, medium, short and so on. The linguistic variable provides approximate characterization of a complex problem. The name of the variable, the universe of discourse and a fuzzy subset of universe of discourse characterize a fuzzy variable. A linguistic variable is a variable of a higher order than a fuzzy variable and its values are taken to be fuzzy variables. A linguistic variable is characterized by

1. name of the variable (x);
2. term set of the variable $t(x)$;
3. syntactic rule for generating the values of x;
4. semantic rule for associating each value of x with its meaning.

Apart from the linguistic variables, there exists what are called as *linguistic hedges* (*linguistic modifiers*). For example, in the fuzzy set "very tall", the word "very" is a linguistic hedge. A few popular linguistic hedges include: very, highly, slightly, moderately, plus, minus, fairly, rather.

Reasoning has logic as its basis, whereas *propositions* are text sentences expressed in any language and are generally expressed in an canonical form as

$$z \text{ is } P$$

where z is the symbol of the subject and P is the predicate designing the characteristics of the subject. For example, "London is in United Kingdom" is a proposition in which "London" is the subject and "in United Kingdom" is the predicate, which specifies a property of "London," i.e., its geographical location in United Kingdom. Every proposition has its opposite, called *negation*. For assuming opposite truth values, a proposition and its negation are required.

Truth tables define logic functions of two propositions. Let X and Y be two propositions, either of which can be true or false. The basic logic operations performed over the propositions are the following:

1. *Conjunction* ($\wedge$) : X AND Y.
2. *Disjunction* ($\vee$) : X OR Y.
3. *Implication or conditional* ($\Rightarrow$): IF X THEN Y.
4. *Bidirectional or equivalence* ($\Leftrightarrow$): X IF AND ONLY IF Y.

On the basis of these operations on propositions, inference rules can be formulated. Few inference rules are as follows:

$$[X \wedge (X \Rightarrow Y)] \Rightarrow Y$$
$$[\overline{Y} \wedge (X \Rightarrow Y)] \Rightarrow \overline{X}$$
$$[(X \Rightarrow Y) \wedge (Y \Rightarrow Z)] \Rightarrow (X \Rightarrow Z)$$

The above rules produce certain propositions that are always true irrespective of the truth values of propositions X and Y. Such propositions are called *tautologies*. An extension of set-theoretic bivalence logic is the fuzzy logic where the truth values are terms of the linguistic variable "truth."

The truth values of propositions in fuzzy logic are allowed to range over the unit interval [0, 1]. A truth value in fuzzy logic "very true" may be interpreted as a fuzzy set in [0, 1]. The truth value of the proposition "Z is A," or simply the truth value of A, denoted by tv(A) is defined by a point in [0, 1] (called the numerical truth value) or a fuzzy set in [0, 1] (called the linguistic truth value).

The truth value of a proposition can be obtained from the logic operations of other propositions whose truth values are known. If tv(X) and tv(Y) are numerical truth values of propositions X and Y, respectively, then

$$\text{tv}(X \text{ AND } Y) = \text{tv}(X) \wedge \text{tv}(Y) = \min\{\text{tv}(X), \text{tv}(Y)\} \quad \text{(Intersection)}$$
$$\text{tv}(X \text{ OR } Y) = \text{tv}(X) \vee \text{tv}(Y) = \max\{\text{tv}(X), \text{tv}(Y)\} \quad \text{(Union)}$$
$$\text{tv}(\text{NOT } X) = 1 - \text{tv}(X) \quad \text{(Complement)}$$
$$\text{tv}(X \Rightarrow Y) = \text{tv}(X) \Rightarrow \text{tv}(B) = \max\{1 - \text{tv}(X), \min[\text{tv}(X), \text{tv}(Y)]\}$$

12.3 Fuzzy Propositions

For extending the reasoning capability, fuzzy logic uses fuzzy predicates, fuzzy-predicate modifiers, fuzzy quantifiers and fuzzy qualifiers in the fuzzy propositions. The fuzzy propositions make the fuzzy logic differ

from classical logic. The fuzzy propositions are as follows:

1. *Fuzzy predicates*: In fuzzy logic the predicates can be fuzzy, for example, tall, short, quick. Hence, we have proposition like "Peter is tall." It is obvious that most of the predicates in natural language are fuzzy rather than crisp.
2. *Fuzzy-predicate modifiers*: In fuzzy logic, there exists a wide range of predicate modifiers that act as hedges, for example, very, fairly, moderately, rather, slightly. These predicate modifiers are necessary for generating the values of a linguistic variable. An example can be the proposition "Climate is moderately cool," where "moderately" is the fuzzy predicate modifier.
3. *Fuzzy quantifiers*: The fuzzy quantifiers such as most, several, many, frequently are used in fuzzy logic. Employing these, we can have proposition like "Many people are educated." A fuzzy quantifier can be interpreted as a fuzzy number or a fuzzy proposition, which provides an imprecise characterization of the cardinality of one or more fuzzy or nonfuzzy sets. Fuzzy quantifiers can be used to represent the meaning of propositions containing probabilities; as a result, they can be used to manipulate probabilities within fuzzy logic.
4. *Fuzzy qualifiers*: There are four modes of qualification in fuzzy logic, which are as follows:
   - *Fuzzy truth qualification*: It is expressed as "x is τ," in which τ is a fuzzy truth value. A fuzzy truth value claims the degree of truth of a fuzzy proposition. Consider the example,

     (Paul is Young) is NOT VERY True.

     Here the qualified proposition is (Paul is Young) and the qualifying fuzzy truth value is "NOT Very True."
   - *Fuzzy probability qualification*: It is denoted as "x is λ," where λ is fuzzy probability. In conventional logic, probability is either numerical or an interval. In fuzzy logic, fuzzy probability is expressed by terms such as likely, very likely, unlikely, around and so on. Consider the example,

     (Paul is Young) is Likely.

     Here the qualifying fuzzy probability is "Likely." These probabilities may be interpreted as fuzzy numbers, which may be manipulated using fuzzy arithmetic.
   - *Fuzzy possibility qualification*: It is expressed as "x is π," where π is a fuzzy possibility and can be of the following forms: possible, quite possible, almost impossible. These values can be interpreted as labels of fuzzy subsets of the real line. Consider the example

     (Paul is Young) is Almost Impossible.

     Here the qualifying fuzzy possibility is "Almost Impossible."
   - *Fuzzy usuality qualification*: It is expressed as "usually (X) = usually (X is F)," in which the subject X is a variable taking values in a universe of discourse U and the predicate F is a fuzzy subset of U and interpreted as a usual value of X denoted by $U(X) = F$. The propositions that are usually true or the events that have high probability of occurrence are related by the concept of usuality qualification.

12.4 Formation of Rules

The general way of representing human knowledge is by forming natural language expressions given by

IF antecedant THEN consequent.

The above expression is referred to as the IF-THEN rule-based form. There are three general forms that exist for any linguistic variable. They are: (a) assignment statements; (b) conditional statements; (c) unconditional statements.

Table 12-1 The canonical form of fuzzy rule-based system

Rule 1:	If condition $\underset{\sim}{C}_1$, THEN restriction $\underset{\sim}{R}_1$
Rule 2:	If condition $\underset{\sim}{C}_2$, THEN restriction $\underset{\sim}{R}_2$
⋮	
Rule n:	If condition $\underset{\sim}{C}_n$, THEN restriction $\underset{\sim}{R}_n$

1. *Assignment statements*: They are of the form

   y = small
   Orange color = orange
   $a = s$
   Paul is not tall and not very short
   Climate = autumn
   Outside temperature = normal

   These statements utilize "=" for assignment.
2. *Conditional statements*: The following are some examples.

   IF y is very cool THEN stop.
   IF A is high THEN B is low ELSE B is not low.
   IF temperature is high THEN climate is hot.

   The conditional statements use the "IF-THEN" rule-based form.
3. *Unconditional statements*: They can be of the form

   Goto sum.
   Stop.
   Divide by a.
   Turn the pressure low.

The assignment statements limit the value of a variable to a specific quantity. The canonical rule formation for a fuzzy rule-based system is given in Table 12-1. Generally, both unconditional as well as conditional statements place some restrictions on the consequent of the rule-based process. Fuzzy sets and relations generally model the restrictions. The restriction statements, irrespective of conditional or unconditional statements, are usually connected by linguistic connectives such as "and," "or" or "else." The restrictions denoted by $R_1, R_2, \ldots, R_n$ apply to the consequent of the rules.

12.5 Decomposition of Rules (Compound Rules)

A compound rule is a collection of many simple rules combined together. Any compound rule structure may be decomposed and reduced to a number of simple canonical rule forms. The rules are generally based on natural language representations. The following are the methods used for decomposition of compound linguistic rules into simple canonical rules.

1. Multiple conjunctive antecedents

   IF x is $\underset{\sim}{A}_1, \underset{\sim}{A}_2, \ldots, \underset{\sim}{A}_n$ THEN y is B_m.

   Assume a new fuzzy subset $\underset{\sim}{A}_m$ defined as

   $$\underset{\sim}{A}_m = \underset{\sim}{A}_1 \cap \underset{\sim}{A}_2 \cap \cdots \cap \underset{\sim}{A}_n$$

   and expressed by means of membership function

   $$\mu_{\underset{\sim}{A}_m}(x) = \min\left[\mu_{\underset{\sim}{A}_1}(x), \mu_{\underset{\sim}{A}_2}(x), \ldots \mu_{\underset{\sim}{A}_n}(x)\right].$$

   In view of the fuzzy intersection operation, the compound rule may be rewritten as

   IF $\underset{\sim}{A}_m$ THEN $\underset{\sim}{B}_m$.
2. *Multiple disjunctive antecedents*

   IF x is $\underset{\sim}{A}_1$ OR x is $\underset{\sim}{A}_2$, ... OR x is $\underset{\sim}{A}_n$ THEN y is $\underset{\sim}{B}_m$.

   This can be written as

   IF x is $\underset{\sim}{A}_n$ THEN y is $\underset{\sim}{B}_m$

   where the fuzzy set $\underset{\sim}{A}_m$ is defined as

   $$\underset{\sim}{A}_m = \underset{\sim}{A}_1 \cup \underset{\sim}{A}_2 \cup \underset{\sim}{A}_3 \cup \cdots \cup \underset{\sim}{A}_n$$

   The membership function is given by

   $$\mu_{\underset{\sim}{A}_m}(x) = \max[\mu_{\underset{\sim}{A}_1}(x), \mu_{\underset{\sim}{A}_2}(x), \ldots, \mu_{\underset{\sim}{A}_n}(x)]$$

   which is based on fuzzy union operation.
3. Conditional statements (with ELSE and UNLESS):
   Statements of the kind

   IF $\underset{\sim}{A}_1$ THEN ($\underset{\sim}{B}_1$ ELSE $\underset{\sim}{B}_2$)

   can be decomposed into two simple canonical rule forms, connected by "OR":

   IF $\underset{\sim}{A}_1$ THEN $\underset{\sim}{B}_1$
   OR
   IF NOT $\underset{\sim}{A}_1$ THEN $\underset{\sim}{B}_2$
   IF $\underset{\sim}{A}_1$ (THEN $\underset{\sim}{B}_1$) UNLESS $\underset{\sim}{A}_2$

   can be decomposed as

   IF $\underset{\sim}{A}_1$ THEN $\underset{\sim}{B}_1$
   OR
   IF $\underset{\sim}{A}_2$ THEN NOT $\underset{\sim}{B}_1$
   IF $\underset{\sim}{A}_1$ THEN ($\underset{\sim}{B}_1$) ELSE IF $\underset{\sim}{A}_2$ THEN ($\underset{\sim}{B}_2$)

   can be decomposed into the form

   IF $\underset{\sim}{A}_1$ THEN $\underset{\sim}{B}_1$
   OR
   IF NOT $\underset{\sim}{A}_1$ AND IF $\underset{\sim}{A}_2$ THEN $\underset{\sim}{B}_2$

4. Nested-IF-THEN rules:
   The rule "IF $\underset{\sim}{A}_1$ THEN [IF $\underset{\sim}{A}_2$ THEN ($\underset{\sim}{B}_1$)]" can be of the form

$$\text{IF } \underset{\sim}{A}_1 \text{ AND } \underset{\sim}{A}_2 \text{ THEN } \underset{\sim}{B}_1$$

Thus, based on all the above-mentioned methods compound rules can be decomposed into series of canonical simple rules.

12.6 Aggregation of Fuzzy Rules

The rule-based system involves more than one rule. *Aggregation of rules* is the process of obtaining the overall consequents from the individual consequents provided by each rule. The following two methods are used for aggregation of fuzzy rules:

1. *Conjunctive system of rules*: For a system of rules to be jointly satisfied, the rules are connected by "and" connectives. Here, the aggregated output, y, is determined by the fuzzy intersection of all individual rule consequents, y_i, where $i = 1$ to n, as

$$y = y_1 \text{ and } y_2 \text{ and } \ldots \text{ and } y_n$$

or

$$y = y_1 \cap y_2 \cap y_3 \cap \cdots \cap y_n$$

This aggregated output can be defined by the membership function

$$\mu_y(y) = \min\left[\mu_{y_1}(y),\ \mu_{y_2}(y), \ldots, \mu_{y_n}(y)\right] \quad \text{for} \quad y \in Y$$

2. *Disjunctive system of rules*: In this case, the satisfaction of at least one rule is required. The rules are connected by "or" connectives. Here, the fuzzy union of all individual rule contributions determines the aggregated output, as

$$y = y_1 \text{ or } y_2 \text{ or } \ldots \text{ or } y_n$$

or

$$y = y_1 \cup y_2 \cup y_3 \cup \cdots \cup y_n$$

Again it can be defined by the membership function

$$\mu_y(y) = \max\left[\mu_{y_1}(y),\ \mu_{y_2}(y), \ldots \mu_{y_n}(y)\right] \text{ for } y \in Y$$

12.7 Fuzzy Reasoning (Approximate Reasoning)

Fuzzy reasoning is the collection of topics discussed in Sections 12.4–12.6. In fuzzy logic both the antecedents and consequents are allowed to be fuzzy propositions. There exist four modes of fuzzy approximate reasoning, which include:

1. categorical reasoning;
2. qualitative reasoning;
3. syllogistic reasoning;
4. dispositional reasoning.

12.7.1 Categorical Reasoning

In this type of reasoning, the antecedents contain no fuzzy quantifiers and fuzzy probabilities. The antecedents are assumed to be in canonical form. For understanding the inference rules of categorical reasoning in fuzzy logic, one should take note of the following notations:
$\underset{\sim}{L}, \underset{\sim}{M}, \underset{\sim}{N}, \ldots$ = fuzzy variables taking in the universes U, V, W;
$\underset{\sim}{A}, \underset{\sim}{B}, \underset{\sim}{C}$ = fuzzy predicates.

1. The *projection rule* of inference is defined by

$$\frac{\underset{\sim}{L}, \underset{\sim}{M}, \text{ is } \underset{\sim}{R}}{\underset{\sim}{L} \text{ is } [\underset{\sim}{R} \downarrow \underset{\sim}{L}]}$$

where $[\underset{\sim}{R} \downarrow \underset{\sim}{L}]$ denotes the projection of fuzzy relation $\underset{\sim}{R}$ on $\underset{\sim}{L}$.

2. The *conjunction rule* of inference is given by

$$\underset{\sim}{L} \text{ is } \underset{\sim}{A}, \underset{\sim}{L} \text{ is } \underset{\sim}{B} \Rightarrow \underset{\sim}{L} \text{ is } \underset{\sim}{A} \cap \underset{\sim}{B}$$

$$(\underset{\sim}{L}, \underset{\sim}{M}) \text{ is } \underset{\sim}{A}, \mathrm{L} \text{ is } \underset{\sim}{B} \Rightarrow (\underset{\sim}{L}, \underset{\sim}{M}) \text{ is } \underset{\sim}{A} \cap (\underset{\sim}{B} \times \underset{\sim}{V})$$

$$(\underset{\sim}{L}, \underset{\sim}{M}) \text{ is } \underset{\sim}{A}, (\underset{\sim}{M}, \underset{\sim}{N}) \text{ is } \underset{\sim}{B} \Rightarrow (\underset{\sim}{L}, \underset{\sim}{M}, \underset{\sim}{N}) = (\underset{\sim}{A} \times \underset{\sim}{W}) \cap (\underset{\sim}{U} \times \underset{\sim}{B})$$

3. The *disjunction rule* of inference is given by

$$\underset{\sim}{L} \text{ is } \underset{\sim}{A} \text{ OR } \underset{\sim}{L} \text{ is } \underset{\sim}{B} \Rightarrow \underset{\sim}{L} \text{ is } \underset{\sim}{A} \times \underset{\sim}{B}$$

$$\underset{\sim}{L} \text{ is } \underset{\sim}{A}, \underset{\sim}{M} \text{ is } \underset{\sim}{B} \Rightarrow (\underset{\sim}{L}, \underset{\sim}{M}) \text{ is } \underset{\sim}{A} \times \underset{\sim}{B}$$

4. The *negative rule* of inference is given by

$$\text{NOT } (\underset{\sim}{L} \text{ is } \underset{\sim}{A}) \Rightarrow \underset{\sim}{L} \text{ is } \overline{\underset{\sim}{A}}$$

5. The *compositional rule* of inference is given by

$$\underset{\sim}{L} \text{ is } \underset{\sim}{A}, (\underset{\sim}{L}, \underset{\sim}{M}) \text{ is } \underset{\sim}{R} \Rightarrow \underset{\sim}{M} \text{ is } \underset{\sim}{A} \cdot \underset{\sim}{R}$$

where $\underset{\sim}{A} \cdot \underset{\sim}{R}$ denotes the max-min composition of a fuzzy set A and a fuzzy relation R given by

$$\mu_{\underset{\sim}{A} \cdot \underset{\sim}{R}}(v) = \max_u \min\left[\mu_{\underset{\sim}{A}}(u), \mu_{\underset{\sim}{R}}(u, v)\right]$$

6. The *extension principle* is defined as

$$\underset{\sim}{L} \text{ is } \underset{\sim}{A} \Rightarrow f(\underset{\sim}{L}) \text{ is } f(\underset{\sim}{A})$$

where "f" is a mapping from u to v so that L is mapped into $f(\underset{\sim}{L})$; and based on the extension principle, the membership function of $f(\underset{\sim}{A})$ is defined as

$$\mu_{f(\underset{\sim}{A})}(v) = \sup_{v = f(u)} \mu_{\underset{\sim}{A}}(u), \quad u \in U, v \in V$$

12.7.2 Qualitative Reasoning

In qualitative reasoning the input–output relationship of a system is expressed as a collection of fuzzy IF THEN rules where the antecedents and consequents involve fuzzy linguistic variables. Qualitative reasoning is widely used in control system analysis. Let $\underset{\sim}{A}$ and $\underset{\sim}{B}$ be the fuzzy input variables and $\underset{\sim}{C}$ be the fuzzy output variable. The relation among $\underset{\sim}{A}$, $\underset{\sim}{B}$ and $\underset{\sim}{C}$ may be expressed as

$$\text{If } \underset{\sim}{A} \text{ is } x_1 \text{ AND } \underset{\sim}{B} \text{ is } y_1, \text{ THEN } \underset{\sim}{C} \text{ is } z_1$$
$$\text{If } \underset{\sim}{A} \text{ is } x_2 \text{ AND } \underset{\sim}{B} \text{ is } y_2, \text{ THEN } \underset{\sim}{C} \text{ is } z_2$$
$$\vdots$$
$$\text{If } \underset{\sim}{A} \text{ is } x_n \text{ AND } \underset{\sim}{B} \text{ is } y_n, \text{ THEN } \underset{\sim}{C} \text{ is } z_n$$

where x_i, y_i and z_i, $i = 1$ to n, are fuzzy subsets of their respective universe of discourse. This is similar to the canonical rule formation shown in Table 12-1.

12.7.3 Syllogistic Reasoning

In syllogistic reasoning, antecedents with fuzzy quantifiers are related to inference rules. A fuzzy syllogism can be expressed as follows:

$$\frac{\begin{array}{l} x = k_1 \text{ A's are B's} \\ y = k_2 \text{ C's are D's} \end{array}}{z = k_3 \text{ E's are F's}}$$

In the above A, B, C, D, E and F are fuzzy predicates; k_1 and k_2 are the given fuzzy quantifiers and k_3 is the fuzzy quantifier which has to be decided. All the fuzzy predicates provide a collection of fuzzy syllogisms. These syllogisms create a set of inference rules, which combines evidence through conjunction and disjunction. Given below are some important fuzzy syllogisms.

1. *Produce syllogism*: $C \cdot A \wedge B, F = C \wedge D$
2. *Chaining syllogism*: $C = B, F = D, E = A$
3. *Consequent conjunction syllogism*: $F = B \wedge D, A = C = E$
4. *Consequent disjunction syllogism*: $F = B \vee D, A = C = E$
5. *Precondition conjunction syllogism*: $E = A \wedge C, B = D = F$
6. *Precondition disjunction syllogism*: $E = A \vee C, B = D = F$

12.7.4 Dispositional Reasoning

In this kind of reasoning, the antecedents are dispositions that may contain, implicitly or explicitly, the fuzzy quantifier "usually." Usuality plays a major role in dispositional reasoning and it links together the dispositional and syllogistic modes of reasoning. The important inference rules of dispositional reasoning are the following:

1. Dispositional projection rule of inference:

$$\text{usually } ((L, M) \text{ is } R) \Rightarrow \text{ usually } (L \text{ is } [R \downarrow L])$$

where $[R \downarrow L]$ is the projection of fuzzy relation R on L.

2. Dispositional chaining hypersyllogism: $k_1 A$'s are B's, $k_2 B$'s are C's, usually $(B \subset A)$

$$\text{usually } (\rightarrow \cdot (Q_1(\cdot)Q_2) \text{ A's are C's})$$

The fuzzy quantifier "usually" is applied to the containment relation $B \subset A$

3. Dispositional consequence conjunction syllogism:

$$\text{usually (A's are B's), usually (A's are C's)} \Rightarrow 2 \text{ usually } (-) \text{ 1 (A's are (B and C)'s)}$$

is a specific case of dispositional reasoning.

4. Dispositional entailment rule of inference:

$$\text{usually } (x \text{ is } A), \text{ A} \subset \text{B} \Rightarrow \text{usually } (x \text{ is } B)$$
$$x \text{ is } A, \text{ usually (A} \subset \text{B)} \Rightarrow \text{usually } (x \text{ is } B)$$
$$\text{usually } (x \text{ is } A), \text{ usually (A} \subset \text{B)} \Rightarrow \text{usually}^2 (x \text{ is } B)$$

is the dispositional entailment rule of inference. Here "usually^2" is less specific than "usually."

12.8 Fuzzy Inference Systems (FIS)

Fuzzy rule-based systems, fuzzy models, and fuzzy expert systems are generally known as fuzzy inference systems. The key unit of a fuzzy logic system is FIS. The primary work of this system is decision making. FIS uses "IF ... THEN" rules along with connectors "OR" or "AND" for making necessary decision rules. The input to FIS may be fuzzy or crisp, but the output from FIS is always a fuzzy set. When FIS is used as a controller, it is necessary to have crisp output. Hence, there should be a defuzzification unit for converting fuzzy variables into crisp variables along FIS. The entire FIS is discussed in detail in following subsections.

12.8.1 Construction and Working Principle of FIS

A FIS is constructed of five functional blocks (Figure 12-1). They are:

1. A *rule base* that contains numerous fuzzy IF-THEN rules.
2. A *database* that defines the membership functions of fuzzy sets used in fuzzy rules.
3. *Decision-making unit* that performs operation on the rules.
4. *Fuzzification interface unit* that converts the crisp quantities into fuzzy quantities.
5. *Defuzzification interface unit* that converts the fuzzy quantities into crisp quantities.

The working methodology of FIS is as follows. Initially, in the fuzzification unit, the crisp input is converted into a fuzzy input. Various fuzzification methods are employed for this. After this process, rule base is formed. Database and rule base are collectively called the *knowledge base*. Finally, defuzzification process is carried out to produce crisp output. Mainly, the fuzzy rules are formed in the rule base and suitable decisions are made in the decision-making unit.

12.8.2 Methods of FIS

There are two important types of FIS. They are:

1. Mamdani FIS (1975);
2. Sugeno FIS (1985).

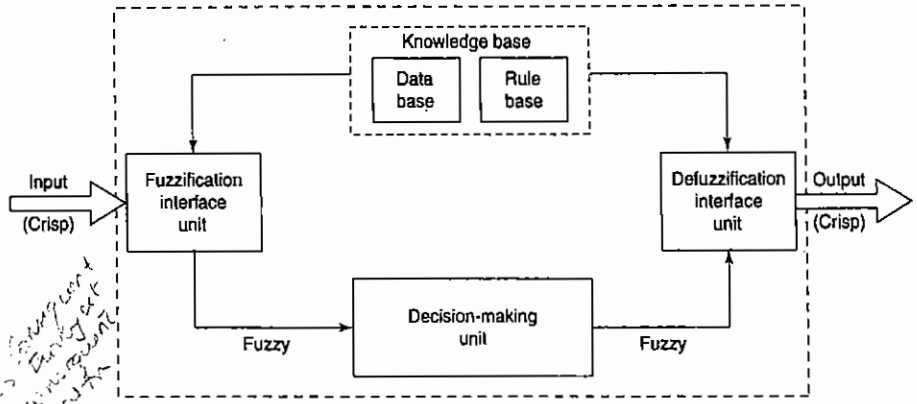


Figure 12-1 Block diagram of FIS.

The difference between the two methods lies in the consequent of fuzzy rules. Fuzzy sets are used as rule consequents in Mamdani FIS and linear functions of input variables are used as rule consequents in Sugeno's method. Mamdani's rule finds a greater acceptance in all universal approximators than Sugeno's model.

12.8.2.1 Mamdani FIS

Ebsahim Mamdani proposed this system in the year 1975 to control a steam engine and boiler combination by synthesizing a set of fuzzy rules obtained from people working on the system. In this case, the output membership functions are expected to be fuzzy sets. After aggregation process, each output variable contains a fuzzy set, hence defuzzification is important at the output stage. The following steps have to be followed to compute the output from this FIS:

Step 1: Determine a set of fuzzy rules.
Step 2: Make the inputs fuzzy using input membership functions.
Step 3: Combine the fuzzified inputs according to the fuzzy rules for establishing a rule strength.
Step 4: Determine the consequent of the rule by combining the rule strength and the output membership function.
Step 5: Combine all the consequents to get an output distribution.
Step 6: Finally, a defuzzified output distribution is obtained.

The fuzzy rules are formed using "IF-THEN" statements and "AND/OR" connectives. The consequence of the rule can be obtained in two steps:

1. by computing the rule strength completely using the fuzzified inputs from the fuzzy combination;
2. by clipping the output membership function at the rule strength.

The outputs of all the fuzzy rules are combined to obtain one fuzzy output distribution. From FIS, it is desired to get only one crisp output. This crisp output may be obtained from defuzzification process. The common techniques of defuzzification used are *center of mass* and *mean of maximum*.

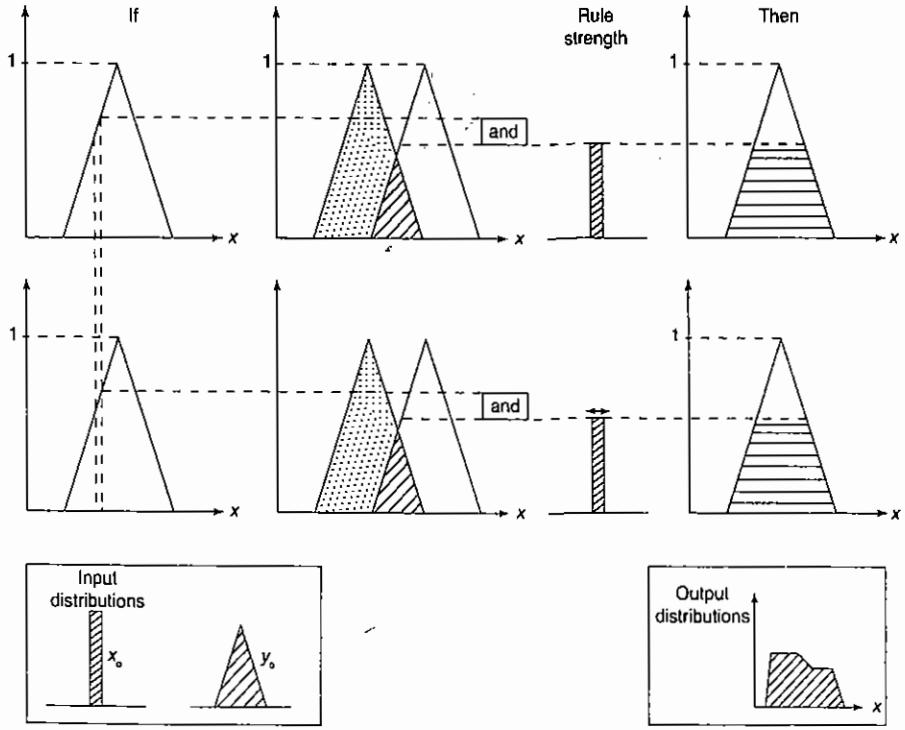


Figure 12-2 A two-input, two-rule Mamdani FIS with a fuzzy input.

Consider a two-input Mamdani FIS with two rules. The model fuzzifies the two inputs by finding the intersection of two crisp input values with the input membership function. The minimum operation is used to compute the fuzzy input "and" for combining the two fuzzified inputs to obtain a rule strength. The output membership function is clipped at the rule strength. Finally, the maximum operator is used to compute the fuzzy output "or" for combining the output of the two rules. This process is illustrated in Figure 12-2.

12.8.2.2 Takagi–Sugeno Fuzzy Model (TS Method)

Sugeno fuzzy method was proposed by Takagi, Sugeno and Kang in the year 1985. The format of the fuzzy rule of a Sugeno fuzzy model is given by

$$\text{IF } x \text{ is } A \text{ and } y \text{ is } B \text{ THEN } z = f(x, y)$$

where AB are fuzzy sets in the antecedents and $z = f(x, y)$ is a crisp function in the consequent. Generally, $f(x, y)$ is a polynomial in the input variables x and y. If $f(x, y)$ is a first-order polynomial, we get first-order Sugeno fuzzy model. If f is a constant, we get zero-order Sugeno fuzzy model. A zero-order Sugeno fuzzy model is functionally equivalent to a radial basis function network under certain minor constraints.

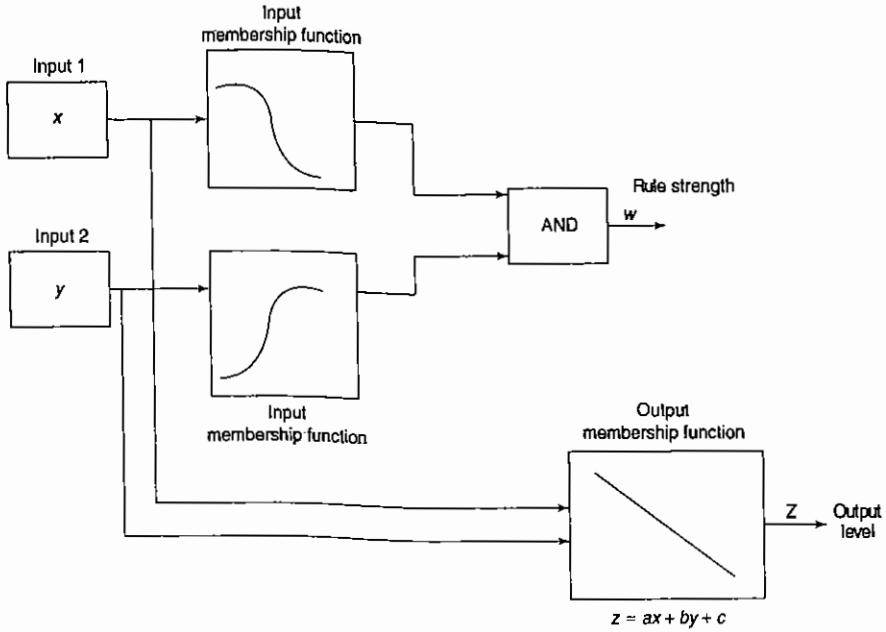


Figure 12-3 Sugeno rule.

The main steps of the fuzzy inference process namely,

1. fuzzifying the inputs;
2. applying the fuzzy operator

are exactly the same. The main difference between Mamdani's and Sugeno's methods is that Sugeno output membership functions are either linear or constant.

The rule format of Sugeno form is given by

$$\text{"If } 3 = x \text{ and } 5 = y \text{ then output is } z = ax + by + c."$$

For a Sugeno model of zero order, the output level z is a constant. The operation of a Sugeno rule is as shown in Figure 12-3.

Sugeno's method can act as an interpolating supervisor for multiple linear controllers, which are to be applied, because of the linear dependence of each rule on the input variables of a system. A Sugeno model is suited for smooth interpolation of linear gains that would be applied across the input space and for modeling nonlinear systems by interpolating between multiple linear models. The Sugeno system uses adaptive techniques for constructing fuzzy models. The adaptive techniques are used to customize the membership functions.

12.8.2.3 Comparison between Mamdani and Sugeno Method

The main difference between Mamdani and Sugeno methods lies in the output membership functions. The Sugeno output membership functions are either linear or constant. The difference also lies in the consequents

of their fuzzy rules and as a result their aggregation and defuzzification procedures differ suitably. A large number of fuzzy rules must be employed in Sugeno method for approximating periodic or highly oscillatory functions. The configuration of Sugeno fuzzy systems can be reduced and it becomes smaller than that of Mamdani fuzzy systems if nontriangular or nontrapezoidal fuzzy input sets are used. Sugeno controllers have more adjustable parameters in the rule consequent and the number of parameters grows exponentially with the increase of the number of input variables. There exist several mathematical results for Sugeno fuzzy controllers than for Mamdani controllers. Formation of Mamdani FIS is more easier than Sugeno FIS.

The main advantages of Mamdani method are:

1. it has widespread acceptance;
2. it is well-suitable for human input;
3. it is intuitive.

On the other hand, the advantages of Sugeno method include:

1. It is computationally efficient.
2. It is compact and works well with linear technique, optimization technique and adaptive technique.
3. It is best suited for mathematical analysis.
4. It has a guaranteed continuity of the output surface

The most important modeling tool based on fuzzy set theory is FIS, and is widely used in various applications.

12.9 Overview of Fuzzy Expert System

An expert fuzzy system is a concept that is much like an expert for a particular problem in humans. There are two major functions of expert systems:

1. It is expected to deal with uncertain and incomplete information.
2. It possess user-interaction function, which contains an explanation of systems intentions and desires as well as decisions during and after the application has been solved.

The basic block diagram of an expert system is shown in Figure 12-4. From Figure 12-4, it can be noticed that an expert system contains three major blocks:

1. *Knowledge base* that contains the knowledge specific to the domain of application.

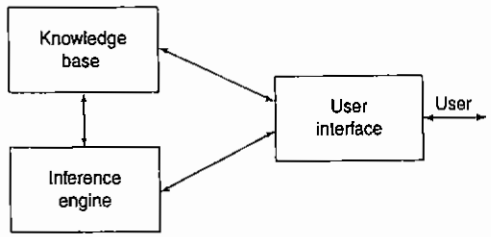


Figure 12-4 Block diagram of an expert system.

2. *Inference engine* that uses the knowledge in the knowledge base for performing suitable reasoning for user's queries.
3. *User interface* that provides a smooth communication between the user and the system.

This also helps the user for understanding entire problem-solving method carried out by the inference engine. An example of an expert system is MYCIN, which introduces the concept of certainty factors for dealing uncertainty. MYCIN rules have a strength, called as certainty factor. This factor lies in the unit interval [0, 1]. When a rule is fired, its prestate condition is evaluated and a firing strength, a value between -1 and $+1$, is associated with the prestate condition. For the firing strength higher than the previously mentioned threshold interval, the consequent of the rule is determined and the conclusion is made with a certainty. The obtained conclusion and its certainty are the evidence provided by this fired rule for the hypotheses given by user. The hypotheses evidence from different rules is combined into belief measures and disbelief measures which are values lying in the interval [0, 1] and [−1, 0], respectively. If belief measure lies above a threshold value, a hypothesis is believed, and if disbelief measure is below a threshold value, a hypothesis is disbelieved. The use of fuzzy logic in traditional expert systems leads to fuzzy expert systems. Fuzzy expert systems are those systems that incorporate fuzzy sets and/or fuzzy logic for their reasoning process and knowledge representation scheme. The fuzzy sets and possibility theory applications to rule-based expert system are mainly developed along the following line.

1. Generalization of certainty factor in MYCIN: enlarging the operations to be used for combining the uncertainty coefficients or by allowing the use of linguistic certainty values along with conventional numerical certainty values.
2. Method of handling of vague predicates in the expression of expert rules or available information.

Fuzzy expert systems effectively handle both uncertainty and vagueness (imprecision). Examples of fuzzy expert system include Z-II, MILORD, etc. Researchers are in the process of developing a wide variety of fuzzy expert systems. One such system is SPERIL, which is a special fuzzy expert system for analyzing earthquake damages.

12.10 Summary

In fuzzy logic, the linguistic variable "truth" plays an important role. The various forms of fuzzy propositions and fuzzy IF-THEN rules that are a useful paradigm for the implementation of human knowledge are discussed. This provides a means for sharing, communicating and transferring the human knowledge to systems and processes. Fuzzy rules are presented in canonical form. The decomposition of fuzzy compound rules and aggregation of fuzzy rules were also discussed, as also four methods of approximate reasoning thereby creating fuzzy inference rules. The Mamdani and Sugeno FIS give a base for building fuzzy rule base system. The comparisons between the two methods are also included. Finally, we provide an overview of fuzzy expert system, which deals with certainty factor.

12.11 Review Questions

1. Define linguistic variable.
2. State the importance of truth values and truth tables.
3. What is meant by linguistic hedges?
4. What are the characteristics of a linguistic variable?
5. List the basic logic operations performed over the propositions.
6. Write short note on fuzzy propositions.
7. How is a canonical rule formed based on the human knowledge?
8. Mention the general forms that exist for a linguistic variable.
9. In what ways is the decomposition of compound linguistic rules established?
10. Discuss the methods of aggregation of fuzzy rules.
11. Why is approximate reasoning important in fuzzy logic?
12. What are four modes of approximate reasoning?
13. Explain in detail: categorical reasoning and qualitative reasoning.
14. How is a fuzzy syllogism expressed and list the important fuzzy syllogism used generally?
15. State the inference rules of dispositional reasoning.
16. What is fuzzy inference system (FIS)?
17. With suitable block diagram, explain the working principle of an FIS.
18. List the methods of FIS.
19. Describe in detail of formation of inference rules in a Mamdani FIS.
20. Discuss in brief on Takagi–Sugeno FIS.
21. State the advantages and disadvantages of Mamdani FIS.
22. List the application of Sugeno FIS.
23. Differentiate between Mamdani FIS and Sugeno FIS.
24. Define expert system. How is a fuzzy expert system formed? State its importance.
25. Mention a few fuzzy expert systems used in current scenario.

12.12 Exercise Problems

1. The membership functions for the linguistic variables "tall" and "short" are given below.

$$\text{"tall"} = \left\{\frac{0.2}{5} + \frac{0.3}{7} + \frac{0.7}{9} + \frac{0.9}{11} + \frac{1.0}{12}\right\}$$

$$\text{"short"} = \left\{\frac{0.3}{0} + \frac{0}{30} + \frac{1}{60} + \frac{0.5}{90} + \frac{0}{120}\right\}$$

Develop membership functions for the following linguistic phrases:

(a) Very tall;
(b) Fairly tall;
(c) Not very short.

2. Develop an FIS editor for a liquid level controller model (Mamdani and Sugeno fuzzy inference models).
3. Develop an FIS (Mamdani) model for controlling temperature in an air conditioner.
4. With a suitable case study, demonstrate the canonical rule formation, aggregation of the fuzzy rules and decomposition of compound rules formed.
5. Give an example for the following propositional principles:
   (a) Fuzzy truth qualification
   (b) Fuzzy possibility qualification
   (c) Fuzzy probability qualification
   (d) Fuzzy usuality qualification
6. Provide examples for fuzzy propositions including fuzzy predicates and fuzzy quantifiers.
7. Give an example for each of the following approximate reasoning rules:
   (a) Compositional rule of inference
   (b) Conjunction rule of inference
   (c) Disjunction rule of inference

8. Change the following symbolic rule to canonical form:
   If L_1 is R_1 (THEN M_1 AND M_2 (IF L_2 is R_2 (THEN M_3 (IF L_3 is R_3 THEN M_4))))
9. Develop a Sugeno FIS for a satellite tracking control system.
10. With suitable application case study, analyze MILORD fuzzy expert system. Compare its performance with conventional fuzzy system.

13 Fuzzy Decision Making

Learning Objectives

- Discusses on variable paradigms available for fuzzy decision making.
- The importance of multiobjective and multiperson decision making.
- How evaluation of alternatives are carried out using the attributes of the object.
- An overview on fuzzy Bayesian decision making.

13.1 Introduction

Decision making is a very important social, economical and scientific endeavor. Decision-making activities are the steps taken to choose a suitable alternative from those that are needed for realizing a certain goal. The decision-making process involves three steps:

1. determining the set of alternatives;
2. evaluating alternatives;
3. comparison between alternatives.

In any decision process, the information about the outcome is considered and a suitable path has to be chosen from two or more alternatives for subsequent action; when good decisions are made, good output is expected. If a decision is made under certainty, then the outcome for each process can be determined precisely; one should note that whenever decision is made, it is under risk condition. The prime domain for fuzzy decision making is the existing uncertainty. There are several situations under the decision-making process. There may be situations when even though decisions made are good, the output may be adverse or vice-versa. When good decisions are made continuously for a longer period, advantageous situations may prevail.

When there are several objectives to be realized in making a decision, the decision making is called multiobjective decision making. The knowledge of experts becomes very essential when decision making is very tedious. The information may be available for the following: the possible outcomes, change in conditions with respect to time about value of new information, when the priority for each action is typically ambiguous, vague and otherwise fuzzy. Obtaining an evaluation structure for selecting alternatives and establishing selection standards are very important stages. The evaluation of alternatives may be carried out based on several attributes of the object; such a decision making is called multiattribute decision making. In this chapter we would discuss the various paradigms available for decision making.

13.2 Individual Decision Making

A decision-making model in this situation is characterized by the following:

1. set of possible actions;
2. set of goals $G_i(i \in X_n)$;
3. set of constraints $C_j(j \in X_m)$.

The goals and constraints are expressed in terms of fuzzy sets. These fuzzy sets in individual decision making are not defined directly on the set of actions, but by means of other sets that characterize relevant states of nature. Consider a set A. Then the goal and constraint for this set are given by

$$G_i(a) = \text{Composition}[G_i(a)] = G_i^1(G_i(a)) \text{ with } G_i^1$$
$$C_j(a) = \text{Composition}[C_j(a)] = C_j^1(C_j(a)) \text{ with } C_j^1$$

for $a \in A$. The fuzzy decision in this case is given by

$$F_D = \min\left[\inf_{i \in X_n} G_i(a), \inf_{j \in X_m} C_j(a)\right]$$

13.3 Multiperson Decision Making

Decision making in this case includes several persons. The expert knowledge from various persons is utilized to make decisions. The difference between the individual decision making and multiperson decision making is: The goals of individual decision makers differ, i.e., each places a different ordering arrangement. On the other hand, in multiperson decision making, the decision makers have access to different information upon which to base their decision.

Here, each member of a group of "n" individual decision makers has a preference ordering $\text{PO}_k, k \in x_n$, which totally or partially orders a set X. A social choice (sc) function has to be found, given the individual preference ordering. The fuzzy relation for a social choice preference function is given by

$$\text{SC} : X \times X \rightarrow [0, 1]$$

which has a membership of $SC(X_i, X_j)$, which indicates the preference of alternative X_i over X_j. Let

$$\text{Number of persons preferring } X_i \text{ to } X_j = N(X_i, X_j)$$
$$\text{Total number of decision makers} = n$$

Then,

$$\text{SC}(x_i, x_j) = \frac{\text{N}(x_i, x_j)}{n}$$

The multiperson decision making is also given by

$$\text{SC}(x_i, x_j) = \begin{cases} 1 & \text{if } x_i >_k x_j \text{ for some } k \\ 0 & \text{otherwise} \end{cases}$$

13.4 Multiobjective Decision Making

In making a decision when there are several objectives to be realized, then the decision making is called multiobjective decision making. Many decision processes may be based on single objectives such as cost minimization, time consumption, profit maximization and so on. However, if all the above-mentioned objectives are to be considered for a decision-making process, then it becomes multiobjective decision making. The main issues in multiobjective decision making are:

1. to acquire proper information related to the satisfaction of the objectives by various alternatives;
2. to weigh the relative importance of each objective.

Multiobjective decision making involves selection of one alternative a_i from universe of alternatives A given a collection of objectives $\{o\}$ that are important for a decision maker. It is necessary to evaluate how best each alternative satisfies each objective. The main aim here is to combine the weighted objectives into an overall decision function in some way. The decision function represents a mapping of alternatives in A to an ordinal set of ranks. In order to make suitable decisions, the process needs to weigh the relative importance of each objective.

Let us define a universe of n alternatives as

$$A = \{a_1, a_2, \ldots, a_i, \ldots, a_n\}$$

and a set of "m" objectives as

$$O = \{o_1, o_2, \ldots, o_i, \ldots, o_m\}$$

where o_i indicates the ith objective. The degree of membership of alternative a in o_i, denoted $\mu o_i(a)$, is the degree to which alternative a satisfies the criteria mentioned for this objective. A decision function is formed, which simultaneously satisfies all the decision objectives. As a result, the decision function, DF is given by the intersection of all the set of objectives, i.e.,

$$\text{DF} = o_1 \cap o_2 \cap \cdots \cap o_i \cap \cdots \cap o_m$$

The grade of membership that DF has for each alternative a is defined by

$$m_{\text{DF}}(a) = \min[\mu o_1, (a), \mu o_2(a), \ldots, \mu o_i(a), \ldots, \mu o_m(a)]$$

The optimal decision, a^*, will then be the alternative that satisfies the equation

$$m_{\text{DF}}(a^*) = \max_{a \in A}[\mu_{\text{DF}}(a)]$$

Let $\{P\}$ be the set of preferences – linear and ordinal. The element of the preference set will possess linguistic values or will have values in the interval $[0, 1]$, or in the intervals $[-1, 1], [1, 10]$, etc. The preferences are attached to each of the objectives in order to notify the decision maker about the influence that each objective should possess on the chosen alternative. The preference set P contains the parameters $b_i, i = 1$ to m, i.e.,

$$\{P\} = \{b_1, b_2, \ldots, b_i, \ldots, b_m\}$$

Thus for each object, we have a measure as to say how important it is to the decision maker for a given decision. The decision function is then defined by decision measure (DM), which involves objectives and preferences. The intersection of m-tuples of DM gives the decision function:

$$\text{DM} = \text{DM}(o_i, b_i) \rightarrow \text{DM(objectives, preferences)}$$
$$\text{DF} = \text{DM}(o_1, b_1) \wedge \text{DM}(o_2, b_2) \wedge \cdots \wedge \text{DM}(o_i, b_i) \wedge \cdots \wedge \text{DM}(o_m, b_m)$$

The DM for a particular alternative, a_i, is given by

$$\mathrm{DM}(o_i(a), b_i) = b_i \rightarrow o_i(a) = \overline{b}_i \vee o_i(a)$$

where $\overline{b}_i = 1 - b_i$ and $b_i \rightarrow o_i$ indicates a distinct relationship between a preference and its corresponding objective. Neverthless, several objectives can have the same preferences weighting in a cardinal sense; however, they will be distinct in an ordinal sense, even though the equality situation $b_i = b_j$ for $i \neq j$ can exist for certain objectives. A joint intersection of "m" decision measures will give an appropriate decision model:

$$\mathrm{DF} = \prod_{i=1}^{m} (\overline{b}_i \cup o_i)$$

The optimal solution, a^*, is the alternative that maximizes the decision function. When we define

$$C_i = \overline{b}_i \cup o_i$$
$$\mu_{c_i}(a) = \max[\mu_{\overline{b}_i}(a), \mu_{o_i}(a)]$$

the optimal solution in membership form is given by

$$\mu_{\mathrm{DF}}(a^*) = \max_{a \in A}[\min\{\mu_{c_1}(a), \mu_{c_2}(a), \ldots, \mu_{c_i}(a), \ldots, \mu_{c_m}(a)]$$

When ith objective becomes very important in the final decision, b_i increases, so $\overline{b}_i$ tends to decrease. As a result C_i(a) decreases, thereby increasing the likelihood that $C_i(a) - o_i$(a), where $o_i(a)$ at present will be the value of the decision function, is DF, denoting alternative a. When this process is repeated for several alternatives a, the largest value $o_i(a)$ for other alternatives will automatically result in the choice of the optimum solution, a^*. The multiobjective decision-making process works in this manner.

13.5 Multiattribute Decision Making

When there are several objectives to be realized in making a decision, then the decision-making process is known as multiobjective decision making. On the other hand, the evaluation of alternatives can be carried out based on several attributes of the object, in which case the decision-making process is called multiattribute decision making. The attributes may be classified into numerical data, linguistic data and qualitative data. In case of multiattribute measurement and evaluation of alternatives, which have the addition of probabilistic noise, probabilistic statistical methods are used to identify the structures. The problem of decision-making structure in multiattributes deals with determination of an evaluation structure for the multiattribute decision making from the available multiattribute data $X_i(i = 1$ to $n)$ shown in Table 13-1 and alternative evaluations Y. It can also be said that the multiattribute evaluation is carried out on the basis of the linear equation

$$Y = A_1X_1 + A_2X_2 + \cdots + A_iX_i + \cdots + A_rX_r$$

and this is the determination of the weight of each attribute. In Table 13-1, x_{ij} is the value for attribute i of alternative j. The term y_j is the evaluation of alternative j. For $j = 1$ to n and $i = 1$ to r, each value is obtained here as a numerical value or a linguistic expression based on their respective problems.

It is necessary to determine the coefficient A_i for linear multiattribute evaluation which best estimates the evaluation of the alternative for the given object. A few vector expressions are given below:

$$\overline{Y} = [y_1, y_2, \ldots, y_j, \ldots, y_n]$$

Table 13-1 Multiattribute data evaluation

Alternative number	Alternative evaluation	Evaluation of alternative attributes
j	y	$x_1 \ldots x_i \ldots x_r$
1	y_1	$x_{11} \ldots x_{i1} \ldots x_{r1}$
2	y_2	$x_{12} \ldots x_{i2} \ldots x_{r2}$
.	.	. . . . .
.	.	. . . . .
.	.	. . . . .
j	y_j	$x_{1j} \ldots x_{ij} \ldots x_{rj}$
.	.	. . . . .
.	.	. . . . .
.	.	. . . . .
n	y_n	$x_{1n} \ldots x_{in} \ldots x_{rn}$

$$X = \begin{bmatrix} x_{11} & \cdots & x_{i1} & \cdots & x_{r1} \\ . & & & & . \\ . & & & & . \\ . & & & & . \\ x_{1n} & \cdots & x_{in} & \cdots & x_{rn} \end{bmatrix}$$

$$\overline{A} = [a_1, a_2, \ldots, a_i, \ldots, a_r]$$

Triangular fuzzy numbers are used to explain possibilistic regression analysis. The triangular fuzzy number A is given by

$$\mu_A(x) = \begin{cases} 1 - \dfrac{|a - x|}{f}; & a - f \leq x \leq a + f \\ o\,; & \text{otherwise} \end{cases}$$

A is a fuzzy number with center a and width f. In this case it can be written as $A = (a, f)$. The possibilistic linear multiattribute evaluation equation is expressed by

$$Y = A_1X_1 + A_2X_2 + \cdots + A_iX_i + \cdots + A_nX_n$$

Using extension principle, its membership function can be calculated as

$$\mu_Y(y) = \begin{cases} 1 - \dfrac{|y - x^{\mathrm{T}}a|}{f^{\mathrm{T}}|x|}; & x \neq 0 \\ 1; & x = 0, y = 0 \\ 0; & x = 0, y \neq 0 \end{cases}$$

Here, $x = (x_1, x_2, \ldots, x_n)$, $a = (a_1, a_2, \ldots, a_n)$ and $f = (f_1, f_2, \ldots, f_n)$, and x^{T} gives a transposition of vector x. Also note that here, y and A_i are fuzzy numbers. Additionally, for y such that $c^{\mathrm{T}}|x| < |y - x^{\mathrm{T}}a|$, $\mu(y) = 0$. For determining this kind of possibilistic evaluation function, a measure for minimizing the possibility width

$$\delta = f_0 + f_1 + \cdots + f_i + \cdots + f_n$$

is used. To determine the possibilistic evaluation, the following linear equation has to be solved:

$$\min_{a|c} \delta = \min_{a|c}(f_0 + \cdots + f_i + \cdots + f_n)$$

Here

$$(1-k)\sum f_j|x_{ij}| + \sum a_j \cdot x_{ij} \geq y_i$$
$$(1-k)\sum f_j|x_{ij}| - \sum a_j \cdot x_{ij} \geq -y_i, \quad i = 1 \text{ to } n$$

where $k = [0, 1]$ indicates the congruence of the possibilistic regression model. Thus, for evaluation of multiattribute decision making, possibilistic regression analysis is effective.

13.6 Fuzzy Bayesian Decision Making

In classical Bayesian decision-making method, the future states of the nature are characterized as probability events. Conventionally, the probabilities sum to unity. The problem with the fuzzy Bayesian scheme is that the events are ambiguous.

Consider the formation of probabilistic decision analysis. Let the set of possible states of nature be given by $\delta = \{s_1, s_2, \ldots, s_n\}$. Then the vector representation of probabilities of these states is

$$P = \{P(s_1), P(s_2), \ldots, P(s_n)\} \text{ where } \sum_{i=1}^{n} P(s_i) = 1$$

These probabilities are called "prior probabilities." The decision maker can choose from "m" alternatives,

$$A = \{a_1, a_2, \ldots, a_i, \ldots, a_m\}$$

For a given alternative a_j, a utility value μ_{ji} is assigned, if the future state of nature becomes state s_i. The decision maker determines these utility values. These values express the value or cost for each alternative state pair, i.e., for each $a_j - s_i$ combination. The expected utility with the jth alternative is given by

$$\text{EX}(u_j) = \sum_{i=1}^{n} u_{ji} P(s_i)$$

The common decision criterion is the maximum expected utility among all the alternatives, i.e.,

$$\text{EX}(u^*) = \max_j \text{EX}(u_j)$$

This leads to the selection of alternatives a_k if $u^* = \text{EX}(u_k)$. Let the information regarding the true states of nature δ be from n experiments and let it be given by a data vector $X = \{x_1, x_2, \ldots, x_n\}$. This information is used in Bayesian approach for updating the prior probabilities $P(s_i)$. Based on the new information, conditional probabilities are formed, where the probability of each piece of data is determined according to where the true state of nature s_i is known; these probabilities are the presumptions of the future.

The conditional probabilities are also known as likelihood values, given by $P(x_n|s_i)$. This conditional probabilities are used as weights over the previous information, i.e., prior probabilities $P(s_i)$, to determine

updated probabilities called posterior probabilities, $P(x_i, x_n)$. Bayes rule is used to determine the posterior probabilities:

$$P(s_i|x_n) = \frac{P(x_n|s_i)}{P(x_n)} P(s_i)$$

$P(x_n)$ is marginal probability of data (x_n) and is found using the total probability theorem,

$$P(x_n) = \sum_{i=1}^{m} P(x_n|s_i)\, P(s_i)$$

For a given data x_n, the expected utility for the alternative is found from the posterior probabilities:

$$\text{EX}(u_j|x_n) = \sum_{i=1}^{m} u_{ji} P(s_i|x_n)$$

and the maximum expected utility for a given data x_n is given by

$$\text{EX}(u^*|x_n) = \max_j \text{EX}(u_j|x_n)$$

For determining the unconditional maximum expected utility, it is necessary to weigh each of the n conditional compacted withies of the above equation by the respective marginal probabilities for each x_n, i.e., $P(x_n)$:

$$E_x(U_x^*) = \sum_{n=1}^{m} \text{EX}(U^*|x_n) P(x_n)$$

At this stage, a notion called value of information, $v(x)$, is introduced. There exist certain uncertainty in the new information $X = \{x_1, x_2, \ldots, x_n\}$ called as imperfect information. This value of information $V(X)$ is found by the difference between the maximum expected utility without any new information and the maximum expected utility with the new information, i.e.,

$$V(X) = \text{EX}(U_x^*) - \text{EX}(U^*)$$

There exist perfect information as well. For information to be perfect, the conditional probabilities are free of dissonance. The perfect information is represented by the posterior probabilities of 0 or 1, i.e.,

$$P(S_i|x_n) = \begin{cases} 1 \\ 0 \end{cases}$$

The perfect information is denoted by x_p. For this perfect information, the maximum expected utility becomes

$$\text{EX}(U^*) = \sum_{n=1}^{r} \text{EX}(u_{x_p}^*|x_n) P(x_n)$$

and the value of perfect information becomes

$$V(x_p) = \text{EX}(u_{x_p}^*) - \text{EX}(u^*)$$

Let the new information $X = \{x_1, x_2, \ldots, x_i, \ldots, x_n\}$ be a universe of discourse in the units appropriate for the new information. Then the corresponding events (fuzzy events $\underset{\sim}{E}$ on this information are defined. The membership for the fuzzy event may be given by $\mu_{\underset{\sim}{E}}(x_r), x = 1$ to n. Let us define the idea of a "probability of a fuzzy event," i.e., the probability of $\underset{\sim}{E}$, as

$$P(\underset{\sim}{E}) = \sum_{r=1}^{n} \mu_{\underset{\sim}{E}}(x_r) P(x_r)$$

If the fuzzy event, for the above equation is crisp, i.e. $\underset{\sim}{E} = E$, then the probability reduces to

$$P(E) = \sum_{x_r \in E} P(x_r)$$

$$\mu_E = \begin{cases} 1, & x_r \in M \\ 0, & \text{otherwise} \end{cases}$$

This equation describes the probability of a crisp event as the sum of the marginal probabilities of those data points, x_r, which are defined to be in the event, E. The posterior probability of S_i, given fuzzy information $\underset{\sim}{E}$, is

$$P(S_i|\underset{\sim}{E}) = \frac{\sum_{r=1}^{n} P(x_r|s_i)\mu_{\underset{\sim}{E}}(x_r)P(s_i)}{P(\underset{\sim}{E})} = \frac{P(\underset{\sim}{E}|s_i)P(s_i)}{P(\underset{\sim}{E})}$$

where

$$P(\underset{\sim}{E}|S_i) = \sum_{r=1}^{n} P(x_r|S_i)\mu_{\underset{\sim}{E}}(x_r)$$

Defining the collection of all the fuzzy events describing fuzzy information as an orthogonal fuzzy information system, we have $\phi = \{\underset{\sim}{E}_1, \underset{\sim}{E}_2, \ldots, \underset{\sim}{E}_m\}$.

The orthogonal means the sum of the membership values for each fuzzy event $\underset{\sim}{E}_i$, for every data point in the universe of information, x_r, equals unity, i.e.,

$$\sum_{i=1}^{m} \mu_{\underset{\sim}{E}_i}(x_r) = 1 \text{ for all } x_r \in X$$

When the fuzzy events on the new information universe are orthogonal, we extend the Bayesian approach for considering fuzzy information. The fuzzy equivalents for the posterior probability, maximum expected utility and the marginal probability are given by, for a fuzzy event $\underset{\sim}{E}_t$,

$$E(u_j|\underset{\sim}{E}_t) = \sum_{i=1}^{n} u_{ji} P(S_i|\underset{\sim}{E}_t)$$

$$E = (u^*|\underset{\sim}{E}_t) = \max_j E(u_j|\underset{\sim}{E}_t)$$

$$E(u_\phi^*) = \sum_{t=1}^{g} E(u^*|\underset{\sim}{E}_t) P(\underset{\sim}{E}_t)$$

The value of fuzzy information can be determined as

$$v(\phi) = E(u_\phi^*) - E(u^*)$$

13.7 Summary

In this chapter, various fuzzy decision-making methods are discussed. One of the decision-making method – fuzzy Bayesian decision making – is given to accept both fuzzy and random uncertainty. Based on the several objectives to be realized in making a decision, multiobjective decision making was included. The evaluation of alternatives based on several attributes of the object can be carried out; this process called multiattribute decision making is discussed. Also based on the decision of persons involved, individual decision making and multiperson decision making are also dealt with. The main processes involved in decision making are the determination of set of alternatives, evaluating alternatives and comparison between alternatives. In many decision-making situations, the goals, constraints and consequences of the defined alternatives are known imprecisely, which is due to ambiguity and vagueness. Methods for addressing this form of imprecision are important for dealing with many of the uncertainties as we deal within human systems.

13.8 Review Questions

1. What are the steps involved in decision-making process?
2. Write shore note on individual decision making.
3. Differentiate between individual decision maker and multiperson decision maker.
4. Discuss multiperson decision making in detail.
5. What is meant by multiobjective decision making?
6. State the decision function for a multiobjective decision making.
7. Explain multiattribute decision making in detail.
8. Compare and contrast multiobjective decision making and multiattribute decision making.
9. Discuss fuzzy Bayesian decision making in detail.
10. What are the advantages of fuzzy Bayesian decision-making process?

13.9 Exercise Problems

1. Evaluate three different approaches for controlling conditions of a metal smelting cell. The control approaches are:

a_1 = FT, fast tuning
a_2 = MT, medium tuning
a_3 = ST, slow tuning

There are several objectives to consider which are given below:

ϱ_1 = less power consumption
ϱ_2 = over all efficiency
ϱ_3 = error reduction

The control approaches are rated as

$$\varrho_1 = \left\{\frac{0.35}{\text{FT}} + \frac{0.7}{\text{MT}} + \frac{0.2}{\text{ST}}\right\}$$

$$\varrho_2 = \left\{\frac{0.1}{\text{FT}} + \frac{0.4}{\text{MT}} + \frac{0.6}{\text{ST}}\right\}$$

$$\varrho_3 = \left\{\frac{0.5}{\text{FT}} + \frac{0.65}{\text{MT}} + \frac{0.3}{\text{ST}}\right\}$$

The preferences are given by $b_1 = 0.6, b_2 = 0.5$ and $b_3 = 0.4$. What is the best choice of control?

2. With a suitable decision-making algorithm, help a water authority to decide whether or not to build dive for preventing flooding in case of excess rainfall. Assume necessary parameters and membership functions.

14 Fuzzy Logic Control Systems

Learning Objectives

- Need for a fuzzy logic controller.
- How the control system design has to be carried out?
- The basic architecture and operation involved in a fuzzy logic controller system.
- A brief note on fuzzy logic controller model.
- Application of fuzzy logic controller to aircraft landing control problem.

14.1 Introduction

Fuzzy logic control (FLC) is the most active research area in the application of fuzzy set theory, fuzzy reasoning and fuzzy logic. The application of FLC extends from industrial process control to biomedical instrumentation and securities. Compared to conventional control techniques, FLC has been best utilized in complex ill-defined problems, which can be controlled by efficient human operator without knowledge of their underlying dynamics.

A control system is an arrangement of physical components designed to alter another physical system so that this system exhibits certain desired characteristics. There exist two types of control systems: open-loop and closed-loop control systems. In open-loop control systems, the input control action is independent of the physical system output. On the other hand, in closed-loop control system, the input control action depends on the physical system output. Closed-loop control systems are also known as *feedback control systems*. The first step toward controlling any physical variable is to measure it. A sensor measures the controlled signal. A *plant* is the physical system under control. In a closed-loop control system, forcing signals of the system – called inputs – are determined by the output responses of the system. The basic control problem is given as follows: The output of the physical system under control is adjusted by the help of error signal. The difference between the actual response (calculated) of the plant and the desired response gives the error signal. For obtaining satisfactory responses and characteristics for the closed-loop control system, an additional system, called as *compensator* or *controller*, can be added to the loop. The basic block diagram of closed-loop control system is shown in Figure 14-1.

The basic concept behind FLC is to utilize the expert knowledge and experience of a human operator for designing a controller for controlling an application process whose input–output relationship is given by a collection of fuzzy control rules using linguistic variables instead of a complicated dynamic model. The fuzzy control rules are basically IF-THEN rules. The linguistic variables, fuzzy control rules and fuzzy appropriate reasoning are best utilized for designing the controller.

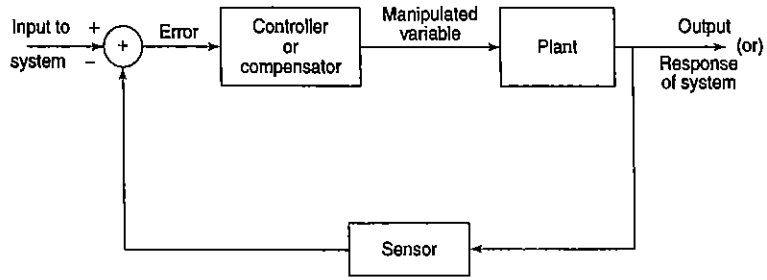


Figure 14-1 Block diagram of a closed-loop control system.

In this chapter we shall introduce the basic structure and design methodologies of an FLC model. FLC is strongly based on the concepts of fuzzy sets, fuzzy relations, fuzzy membership functions, defuzzification, fuzzy rule-based systems and approximate reasoning discussed in the previous chapters.

14.2 Control System Design

Designing a controller for a complex physical system involves the following steps:

1. Decomposing the large-scale system into a collection of various subsystems.
2. Varying the plant dynamics slowly and linearizing the nonlinear plant dynamics about a set of operating points.
3. Organizing a set of state variables, control variables or output features for the system under consideration.
4. Designing simple P, PD, PID controllers for the subsystems. Optimal controllers can also be designed.

Apart from the first four steps, there may be uncertainties occurring due to external environmental conditions. The design of the controller should be made as close as possible to the optimal controller design based on the expert knowledge of the control engineer. This may be done by various numerical observations of the input–output relationship in the form of linguistic, intuitive and other kinds of related information related to the dynamics of plant and external environment.

Finally, a supervisory control system, either manual operator or automatic, forms an extra feedback control loop to tune and adjust the parameters of the controller, for compensating the variational effects caused by nonlinear and unmodeled dynamics.

In comparison with a conventional control system design, an FLC system design should have the following assumptions made, in case it is selected. The plant under,consideration should be observable and controllable. A wide range of knowledge comprising a set of expert linguistic rules, basic engineering common sense, a set of data for input/output or a controller analytic model, which can be fuzzified and from which the fuzzy rule base can be formed, should exist.

Also, for the problem under consideration, a solution should exist and it should be such that the control engineer is working for a "good" solution and not especially looking for an optimum solution. The controller in this case should be designed to the best of our ability and within an acceptable range of precision. It should be noted that the problems of stability and optimality are ongoing problems in fuzzy controller design.

In designing a fuzzy logic controller, the process of forming fuzzy rules plays a vital role. There are four structures of fuzzy production rule system (Weiss and Dónnel, 1979) which are as follows:

1. A *set of rules* that represents the policies and heuristic strategies of the expert decision maker.
2. A *set of input data* that are assessed immediately prior to the actual decision.
3. A method for evaluating any proposed action in terms of its conformity to the expressed rules when there is available data.
4. A method for generating promising actions and determining when to stop searching for better ones.

All the necessary parameters used in fuzzy logic controller are defined by membership functions. The rules are evaluated using techniques such as approximate reasoning or interpolative reasoning. These four structures of fuzzy rules help in obtaining the control surface that relates the control action to the measured state or output variable. The control surface can then be sampled down to a finite number of points and based on this information, a look-up table may be constructed. The look-up table comprises the information about the control surface which can be downloaded into a read-only memory chip. This chip would constitute a fixed controller for the plant.

14.3 Architecture and Operation of FLC System

The basic architecture of a fuzzy logic controller is shown in Figure 14-2. The principal components of an FLC system are: a fuzzifier, a fuzzy rule base, a fuzzy knowledge base, an inference engine and a defuzzifier. It also includes parameters for normalization. When the output from the defuzzifier is not a control action for a plant, then the system is a fuzzy logic decision system. The fuzzifier present converts the crisp quantities into fuzzy quantities. The fuzzy rule base stores the knowledge about the operation of the process of domain expertise. The fuzzy knowledge base stores the knowledge about all the input–output fuzzy relationships. It includes the membership functions defining the input variables to the fuzzy rule base and the output variables to the plant under control. The inference engine is the kernel of an FLC system, and it possess the capability to simulate human decisions by performing approximate reasoning to achieve a desired control strategy. The defuzzifier converts the fuzzy quantities into crisp quantities from an inferred fuzzy control action by the inference engine.

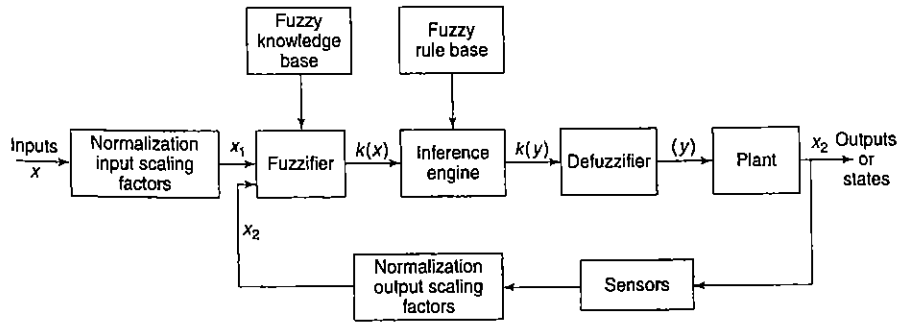


Figure 14-2 Basic architecture of an FLC system.

The various steps involved in designing a fuzzy logic controller are as follows:

Step 1: Locate the input, output and state variables of the plant under consideration.

Step 2: Split the complete universe of discourse spanned by each variable into a number of fuzzy subsets, assigning each with a linguistic label. The subsets include all the elements in the universe.

Step 3: Obtain the membership function for each fuzzy subset.

Step 4: Assign the fuzzy relationships between the inputs or states of fuzzy subsets on one side and the outputs of fuzzy subsets on other side, thereby forming the rule base.

Step 5: Choose appropriate scaling factors for the input and output variables for normalizing the variables between [0, 1] and [−1, 1] interval.

Step 6: Carry out the fuzzification process.

Step 7: Identify the output contributed from each rule using fuzzy approximate reasoning.

Step 8: Combine the fuzzy outputs obtained from each rule.

Step 9: Finally, apply defuzzification to form a crisp output.

The above steps are performed and executed for a simple FLC system. The following design elements are adopted for designing a general FLC system:

1. Fuzzification strategies and the interpretation of a fuzzifier.
2. Fuzzy knowledge base:
   normalization of the parameters involved;
   partitioning of input and output spaces;
   selection of membership functions of a primary fuzzy set.
3. Fuzzy rule base:
   selection of input and output variables;
   source from which fuzzy control rules are to be derived;
   types of fuzzy control rules;
   completeness of fuzzy control rules.
4. Decision-making logic:
   proper definition of fuzzy implication;
   interpretation of connective "and";
   interpretation of connective "or";
   inference engine.
5. Defuzzification strategies and the interpretation of a defuzzifier.

When all the above five design parameters are fixed, the FLC system is simple. Based on all this, the features of a simple FLC system are as follows:
fixed and uniform input and output scaling factors for normalization;
fixed and noninteractive rules;
fixed membership functions;
only limited number of rules, which increases exponentially with the number of input variables;

fixed expertise knowledge;
no hierarchical rule structure and low-level control.

14.4 FLC System Models

There are two different forms of FLC system models:

1. fuzzy rule-based structures;
2. fuzzy relational equations.

Fuzzy rule-based models have already been discussed in a previous chapter. The fuzzy relational equation describing a commonly used FLC model can be of the following forms:

The basic fuzzy model for a first-order discrete system with input a, which is described in state-space representation, is of the form

$$x_{k+1} = x_k \circ u_k \circ \underset{\sim}{R} \quad \text{for } k = 1, 2, \ldots, n$$

where $\circ$ is the composition and $\underset{\sim}{R}$ is the fuzzy system transfer relation. Consider a discrete pth order system with single input u represented in state-space form. The basic fuzzy model of such a system is given by (for $k = 1$ to n)

$$x_{k+p} = x_k \circ x_{k+1} \circ \cdots \circ x_{k+p-1} \circ u_{k+p-1} \circ \underset{\sim}{R}$$
$$y_{k+p} = x_{k+p}$$

where $\underset{\sim}{R}$ is the fuzzy system transfer relation and y_{k+p} is the single output of the system considered.

A second-order system with complete state feedback is given by the fuzzy system equation as (for $k = 1$ to 2)

$$u_k = x_k \circ x_{k-1} \circ \underset{\sim}{R}$$
$$y_k = x_k$$

where y_k is the output of the system. Consider a discrete pth order single-input–single-output system with complete state feedback. The fuzzy model of such a system has the following form:

$$u_{k+p} = y_k \circ y_{x+1} \circ \cdots \circ y_{k+p-1} \circ \underset{\sim}{R} \quad \text{for } k = 1 \text{ to } n$$

The stability of a fuzzy system can be tested by Lyapunov's stability theorem.

14.5 Application of FLC Systems

FLC systems find a wide range of application in various industrial and commercial products and systems. In several applications – related to nonlinear, time-varying, ill-defined systems and also complex systems – FLC systems have proved to be very efficient in comparison with other conventional control systems. The applications of FLC systems include:

1. traffic control;
2. steam engine;
3. aircraft flight control;
4. missile control;
5. adaptive control;

6. liquid-level control;
7. helicopter model;
8. automobile speed controller;
9. braking system controller;
10. process control (includes cement kiln control);
11. robotic control;
12. elevator (auto lift) control;
13. automatic tuning control;
14. cooling plant control;
15. water treatment;
16. boiler control;
17. nuclear reactor control;
18. power systems control;
19. air conditioner control (temperature controller);
20. biological processes;
21. knowledge based system;
22. fault detection control unit;
23. fuzzy hardware implementation and fuzzy computers.

Amidst all these practical applications, the best performance was noticed in cement kiln control system. FLC system has also been successfully implemented to automatic tuning operations and container crane system. The application of an FLC system to household purposes include: washing machines, air conditioners, microwave ovens, cameras, television, palmtop computers and many others. The companies that manufacture fuzzy logic technique based appliances as commercial products are Mitsubishi, Hitachi, Sony, Toshiba, Matsushita, Canon, Sanyo and so on. In the next part of the section, as an illustration of fuzzy logic controller we discuss the application of fuzzy logic in aircraft landing control problem in more detail.

Consider an aircraft landing approach (Figure 14-3). It is necessary to simulate the final descent approach. When the aircraft lands onto the ground, the downward velocity is proportional to the square of the height. Hence, at higher attitudes, a large downward velocity is desired. When the height starts decreasing, the desired downward velocity goes on decreasing. As the height becomes negligibly small, the downward velocity goes to zero. In this manner, the flight descends from attitude promptly but touches the land very gently. The plot for desired downward velocity vs. attitude is shown in Figure 14-4.

The variables utilized for performing this simulation are as follows:

1. height above ground, h;
2. vertical velocity of aircraft, v.

The output to be controlled is the force "f." When this force is applied to the aircraft, it will alter the aircrafts height "h" and velocity "v." It is necessary to derive the differential equation for analyzing.

From Figure 14-5, the momentum "a" for a particle of mass "m" moving with a velocity "v" is given by the product of mass and velocity, i.e. $a = mv$. When an external force "f" is applied in a time interval dt and the particle of mass "m" continues in the same direction with the same velocity "v", then the change in velocity is given by $\Delta v = f\Delta t/m$. When $\Delta t = 1$ s and $m = 1.0$, we get the change in velocity directly

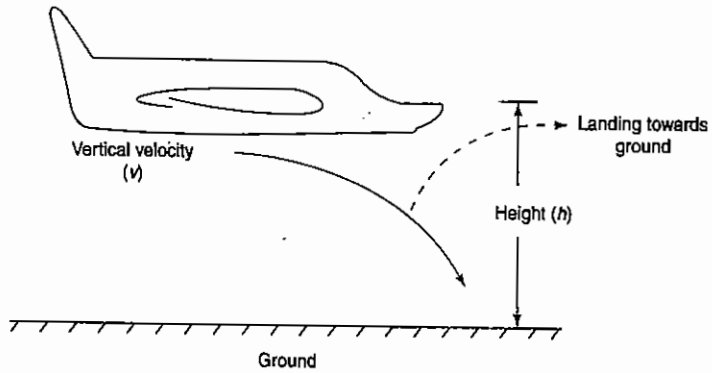


Figure 14-3 Aircraft landing problem.

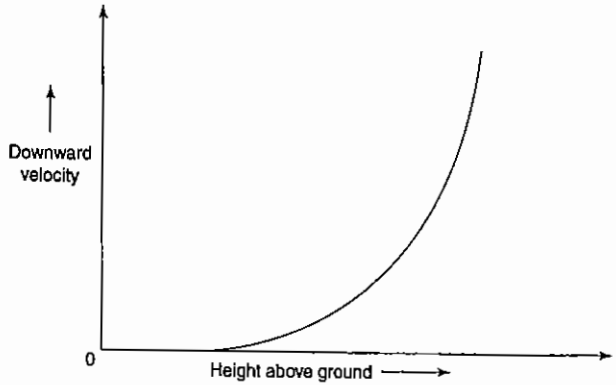


Figure 14-4 Plot of desired downward velocity vs. height.

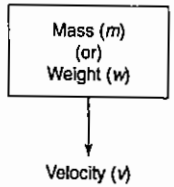


Figure 14-5 Principle of mass and velocity ($a = mv$).

proportional to the applied force. Based on this we obtain the following set of equations:

$$v_{i+1} = v_i + f_i; \quad h_{i+1} = h_i + v_i$$

where v_{i+1} is the new velocity; v_i the old velocity; f_i the force; h_{i+1} the new height; h_i the old height. To implement an FLC model for this, the following steps should be adopted.

Table 14-1 Membership values for height

Height (F)	0	100	200	300	400	500	600	700	800	900
Large (L)	0	0	0	0	0	0.2	0.4	0.6	0.8	1.0
Medium (M)	0	0	0.2	0.4	0.6	0.8	1	0.8	0.6	0.4
Small (S)	1	0.8	0.6	0.4	0.2	0	0	0	0	0

1. Define the fuzzy membership functions for the state variables (height and velocity).
2. Define the fuzzy membership function for the output variable (force).
3. Form the fuzzy rule base system model.
4. Based on the fuzzy rules, form the fuzzy associative memory (FAM) table. The values in the FAM table give the output (force).
5. Define the initial conditions and carry out simulation for one cycle. Several cycles of simulation can be carried out. Let the aircraft be started at an altitude of 900 feet with a downward velocity of -20 ft s^{-1}. The equations used for updation of state variables are (for each cycle)

$$v_{i+1} = v_i + f_i; \quad h_{i+1} = h_i + v_i$$

The membership values for height are given in Table 14-1 and its triangular membership construction is shown in Figure 14-6. The membership values for velocity are given in Table 14-2 and its triangular membership construction is shown in Figure14-7. The membership values for control force are given in Table 14-3 and its triangular membership construction is shown in Figure 14-8. The fuzzy rules are formed as follows:

1. IF height is L AND velocity is D, then control force is Z.
2. If height is L AND velocity is DS, then control force is DS.

In a similar manner, the other rules are formed. There are three linguistic variables defined for height and five linguistic variables defined for velocity; based on these 15 fuzzy rules are formed. The rules are stored in FAM table (Table 14-4). Here initial height, $h_0 = 900$ ft; initial velocity, $v_0 = -20$ ft s^{-1}; control force, f_0 = to be computed.

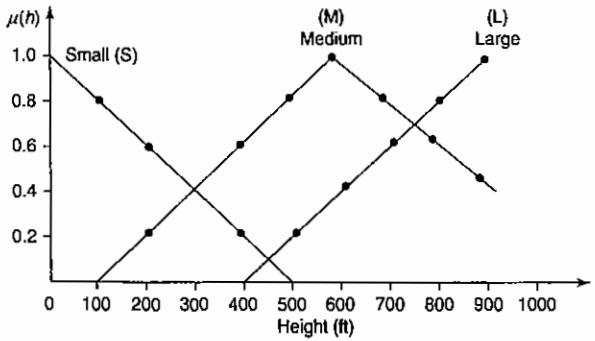


Figure 14-6 Membership function of height (h).

Table 14-2 Membership values for velocity

	Vertical velocity (ft/s)												
	−30	−25	−20	−15	−10	−5	0	5	10	15	20	25	30
Up (U)	0	0	0	0	0	0	0	0	0	0.5	1	1	1
Up small (US)	0	0	0	0	0	0	0	0.5	1	0.5	0	0	0
Zero (Z)	0	0	0	0	0	0.5	1	0.5	0	0	0	0	0
Down small (DS)	0	0	0	0.5	1	0.5	0	0	0	0	0	0	0
Down (D)	1	1	1	0.5	0	0	0	0	0	0	0	0	0

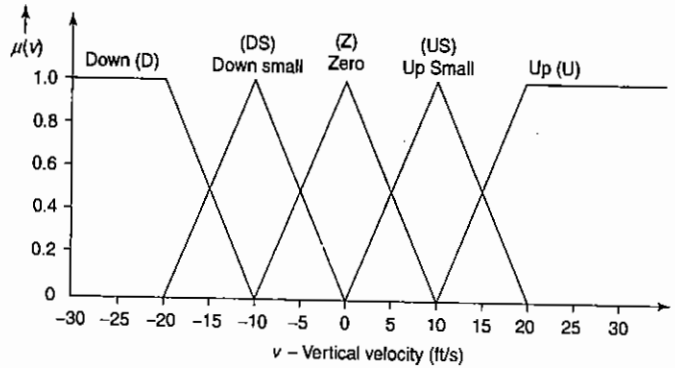


Figure 14-7 Membership function of velocity.

Table 14-3 Membership values for control force

	Output force (lbs)												
	−30	−25	−20	−15	−10	−5	0	5	10	15	20	25	30
Up (U)	0	0	0	0	0	0	0	0	0	0.5	1	1	0
Up small (US)	0	0	0	0	0	0	0	0.5	1	0.5	0	0	0
Zero (Z)	0	0	0	0	0	0.5	1	0.5	0	0	0	0	0
Down small (DS)	0	0	0	0.5	1	0.5	0	0	0	0	0	0	0
Down (D)	1	1	1	0.5	0	0	0	0	0	0	0	0	0

Height h (900) fires L at 1.0 and M at 0.4; velocity v (−20) fires only D at 1.0.

Height		Velocity		Output
L (1.0)	AND	D (1.0)	⇒	Z (1.0)
M (0.4)	AND	D (1.0)	⇒	US (0.4)

The defuzzification can be carried out and the crisp quantity can be extracted. Figure 14-9 shows the consequents truncated and union of fuzzy consequent for cycle 1. The output is $f_0 = 5.2$ lbs (approximately).

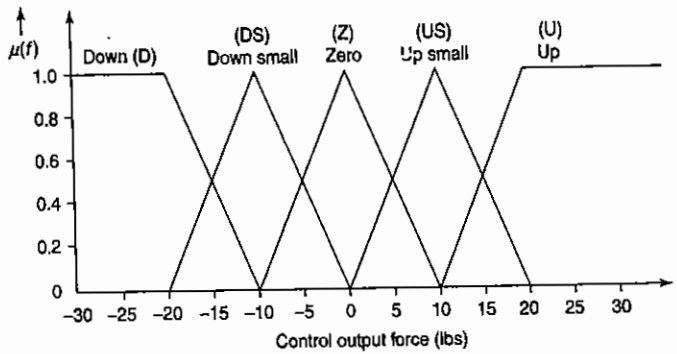


Figure 14-8 Membership value of control force.

Table 14-4 FAM table

Height / Velocity	D	DS	Z	US	V
L	Z	DS	D	D	D
M	US	Z	DS	D	D
S	U	US	Z	DS	DS

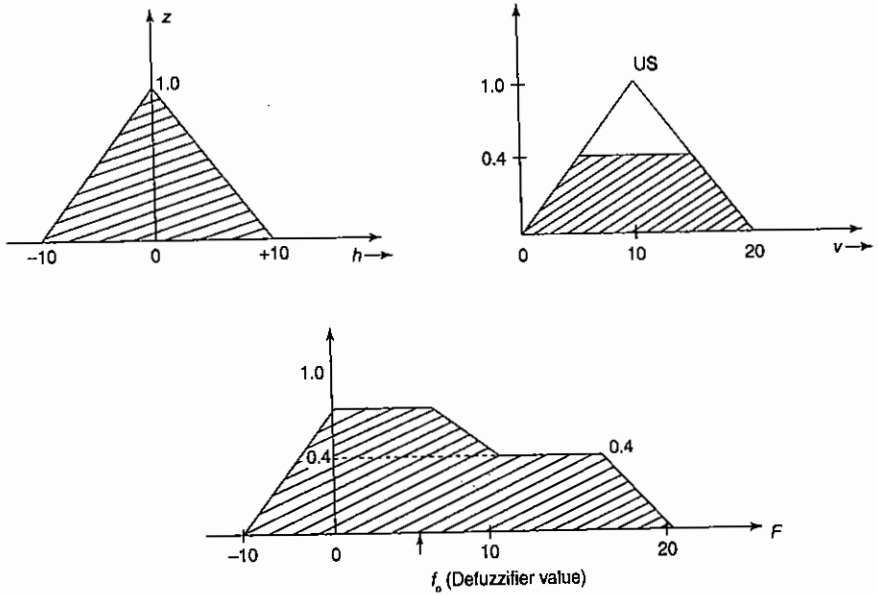


Figure 14-9 Union of fuzzy consequents for cycle 1.

Based on this, the new values of the state variables and output for the next cycle are given by

$$h_1 = h_0 + v_0 = 900 + (-20) = 880 \text{ ft}$$
$$v_1 = v_0 + f_0 = -20 + 5.2 = -14.8 \text{ ft s}^{-1}$$

These are used as the initial values for the next cycle. A number of cycles are carried out until we get a decent profile as shown in Figure 14-3. Generally, a fuzzy logic controller has only a single-layer-rule firing.

14.6 Summary

The basic architecture and design aspects of fuzzy logic controller are introduced in this chapter. Also, an application to aircraft landing problem has been dealt with in detail.

The main key behind the fuzzy logic controller is the set of fuzzy control rules, which describes the input–output relationship of a controlled system. The two types of fuzzy control rules used in the design of fuzzy logic controllers are state evaluation and object evaluation. This chapter mainly focuses on the state evaluation rules, because they find a wide application. The object evaluation fuzzy control rules predict the present and future control actions; in addition the control objectives are evaluated. If these objectives are satisfied, then the control action is applied to the process.

The concepts of stability, observability and controllability are well-established in modern control theory. Owing to the complexity of mathematical analysis of fuzzy logic controllers, the notions of stability and concepts of automatic control theory for fuzzy logic controllers are under research.

14.7 Review Questions

1. State the importance of a control system.
2. What are the two types of control systems?
3. Differentiate between open-loop and closed-loop control systems.
4. List the various control system design aspects.
5. Mention the four structures of fuzzy production rule system.
6. With a neat block diagram, explain the architecture of a fuzzy logic controller.
7. What are the steps involved in designing a fuzzy logic controller?
8. Give the principle design element necessary for the design of general fuzzy logic controller.
9. Mention the features of a simple FLC system.
10. What are the special forms of FLC system models?
11. List the various applications of fuzzy logic controller.
12. With a suitable application case study explain a fuzzy logic controller.

14.8 Exercise Problems

1. Write a computer program to implement a fuzzy logic controller for a aircraft landing problem dealt in Section 14.5.
2. Using fuzzy logic controller, simulate the camera tracking control system.

3. Design a fuzzy logic controller to simulate a temperature control system for a room.
4. Implement a process control application via a fuzzy logic controller.
5. Design and analyze a fuzzy controller for an inverted pendulum as shown in Figure 1.

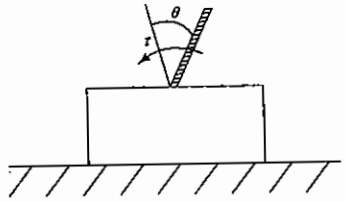


Figure 1 Input pattern.

Genetic Algorithm

15

Learning Objectives

- Gives an introduction to natural evolution.
- Lists the basic operators (selection, crossover, mutation) and other terminologies used in Genetic Algorithms (GAs).
- Discusses the need for schemata approach.
- Details the comparison of traditional algorithm with GA.
- Explains the operational flow of simple GA.
- Description is given of the various classifications of GA – Messy GA, adaptive GA, hybrid GA, parallel GA and independent sampling GA.
- The variants of parallel GA (fine-grained parallel GA and coarse-grained parallel GA) are included.
- Enhances the basic concepts involved in Holland classifier system.
- The various features and operational properties of genetic programming are provided.
- The application areas of GA are also discussed.

Charles R. Darwin says that "Although the belief that an organ so perfect as the eye could have been formed by natural selection is enough to stagger any one; yet in the case of any organ, if we know of a long series of gradations in complexity, each good for its possessor, then, under changing conditions of life, there is no logical impossibility in the acquirement of any conceivable degree of perfection through natural selection."

15.1 Introduction

Charles Darwin has formulated the fundamental principle of natural selection as the main evolutionary tool. He put forward his ideas without the knowledge of basic hereditary principles. In 1865, Gregor Mendel discovered these hereditary principles by the experiments he carried out on peas. After Mendel's work genetics was developed. Morgan experimentally found that chromosomes were the carriers of hereditary information and that genes representing the hereditary factors were lined up on chromosomes. Darwin's natural selection theory and natural genetics remained unlinked until 1920s when it was proved that genetics and selection were in no way contrasting each other. Combination of Darwin's and Mendel's ideas lead to the modern evolutionary theory.

In *The Origin of Species*, Charles Darwin stated the theory of natural evolution. Over many generations, biological organisms evolve according to the principles of natural selection like "survival of the fittest" to reach some remarkable forms of accomplishment. The perfect shape of the albatross wing, the efficiency and the similarity between sharks and dolphins and so on are good examples of what random evolution with absence of intelligence can achieve. So, if it works so well in nature, it should be interesting to simulate natural evolution and try to obtain a method which may solve concrete search and optimization problems.

For a better understanding of this theory, it is important first to understand the biological terminology used in evolutionary computation. It is discussed in Section 15.2.

In 1975, Holland developed this idea in *Adaptation in Natural and Artificial Systems*. By describing how to apply the principles of natural evolution to optimization problems, he laid down the first GA. Holland's theory has been further developed and now GAs stand up as powerful adaptive methods to solve search and optimization problems. Today, GAs are used to resolve complicated optimization problems, such as, organizing the time table, scheduling job shop, playing games.

15.1.1 What are Genetic Algorithms?

GAs are adaptive heuristic search algorithms based on the evolutionary ideas of natural selection and genetics. As such they represent an intelligent exploitation of a random search used to solve optimization problems. Although randomized, GAs are by no means random; instead they exploit historical information to direct the search into the region of better performance within the search space. The basic techniques of the GAs are designed to simulate processes in natural systems necessary for evolution, especially those that follow the principles first laid down by Charles Darwin, "survival of the fittest," because in nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

15.1.2 Why Genetic Algorithms?

They are better than conventional algorithms in that they are more robust. Unlike older AI systems, they do not break easily even if the inputs are changed slightly or in the presence of reasonable noise. Also, in searching a large state-space, multimodal state-space or *n*-dimensional surface, a GA may offer significant benefits over more typical optimization techniques (linear programming, heuristic, depth-first, breath-first and praxis.)

15.2 Biological Background

The science that deals with the mechanisms responsible for similarities and differences in a species is called Genetics. The word "genetics" is derived from the Greek word "genesis" meaning "to grow" or "to become." The science of genetics helps us to differentiate between heredity and variations and accounts for the resemblances and differences during the process of evolution. The concepts of GAs are directly derived from natural evolution and heredity. The terminologies involved in the biological background of species are discussed in the following subsections.

15.2.1 The Cell

Every animal/human cell is a complex of many "small" factories that work together. The center of all this is the cell nucleus. The genetic information is contained in the cell nucleus. Figure 15-1 shows anatomy of the animal cell and cell nucleus.

15.2.2 Chromosomes

All the genetic information gets stored in the chromosomes. Each chromosome is build of deoxyribonucleic acid (DNA). In humans, chromosomes exist in pairs (23 pairs found). The chromosomes are divided into several parts called *genes*. Genes code the properties of species, i.e., the characteristics of an individual. The possibilities of combination of the genes for one property are called *alleles*, and a gene can take different alleles. For example, there is a gene for eye color, and all the different possible alleles are black, brown, blue and green (since no one has red or violet eyes!). The set of all possible alleles present in a particular population forms a *gene pool*. This gene pool can determine all the different possible variations for the future generations. The size of the gene pool helps in determining the diversity of the individuals in the population. The set of all the genes of a specific species is called *genome*. Each and every gene has a unique position on the genome called

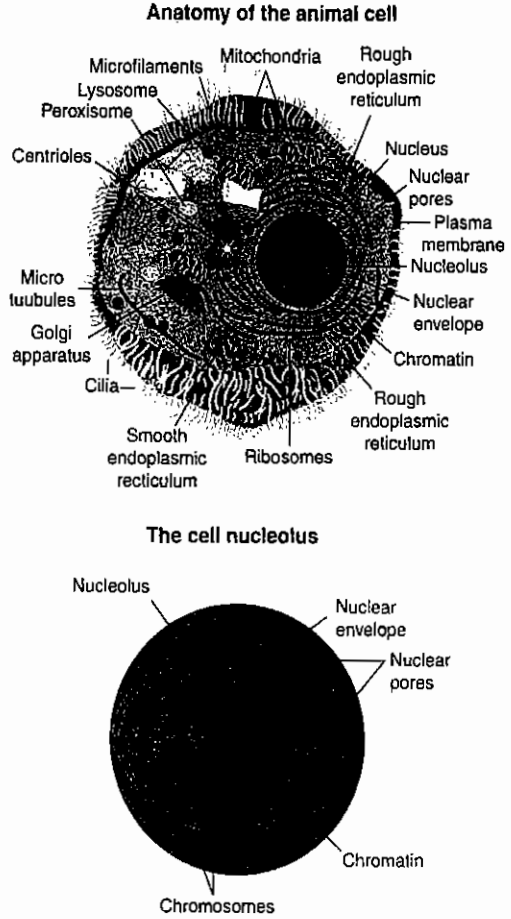


Figure 15-1 Anatomy of animal cell, cell nucleus.

locus. In fact, most living organisms store their genome on several chromosomes, but in the GAs, all the genes are usually stored on the same chromosomes. Thus, chromosomes and genomes are synonyms with one other in GAs. Figure 15-2 shows a model of chromosome.

15.2.3 Genetics

For a particular individual, the entire combination of genes is called *genotype*. The *phenotype* describes the physical aspect of decoding a genotype to produce the phenotype. One interesting point of evolution is that selection is always done on the phenotype whereas the reproduction recombines genotype. Thus, morphogenesis plays a key role between selection and reproduction. In higher life forms, chromosomes contain two sets of genes. These are known as *diploids*. In the case of conflicts between two values of the same pair of genes, the dominant one will determine the phenotype whereas the other one, called *recessive*, will still be present and

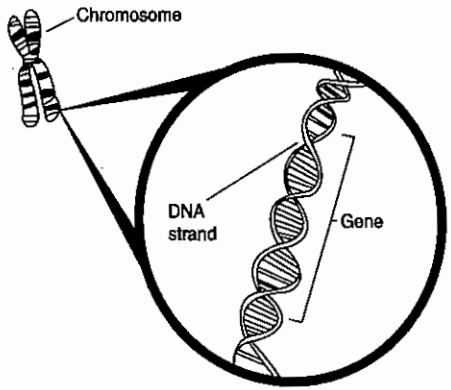


Figure 15-2 Model of chromosome.

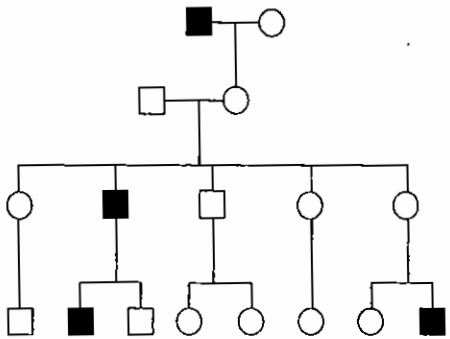


Figure 15-3 Development of genotype to phenotype.

can be passed onto the offspring. *Diploidy* allows a wider diversity of alleles. This provides a useful memory mechanism in changing or noisy environment. However, most GAs concentrate on haploid chromosomes because they are much simple to construct. In haploid representation, only one set of each gene is stored, thus the process of determining which allele should be dominant and which one should be recessive is avoided. Figure 15-3 shows the development of genotype to phenotype.

15.2.4 Reproduction

Reproduction of species via genetic information is carried out by the following:

1. *Mitosis*: In mitosis the same genetic information is copied to new offspring. There is no exchange of information. This is a normal way of growing of multicell structures, such as organs. Figure 15-4 shows mitosis form of reproduction.
2. *Meiosis*: Meiosis forms the basis of sexual reproduction. When meiotic division takes place, two gametes appear in the process. When reproduction occurs, these two gametes conjugate to a zygote which becomes the new individual. Thus in this case, the genetic information is shared between the parents in order to create new offspring. Figure 15-5 shows meiosis form of reproduction.

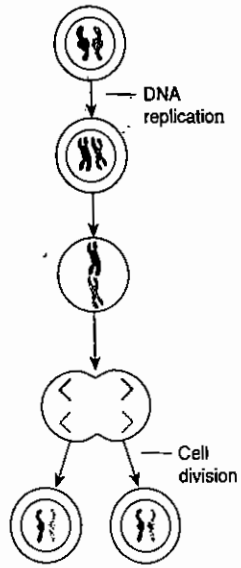


Figure 15-4 Mitosis form of reproduction.

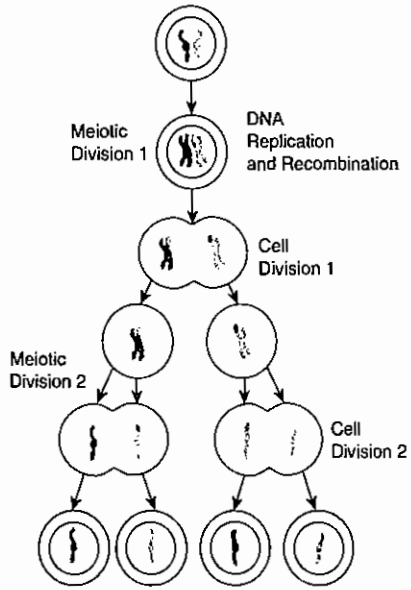


Figure 15-5 Meiosis form of reproduction.

Table 15-1 Comparison of natural evolution and genetic algorithm terminology

Natural evolution	Genetic algorithm
Chromosome	String
Gene	Feature or character
Allele	Feature value
Locus	String position
Genotype	Structure or coded string
Phenotype	Parameter set, a decoded structure

15.2.5 Natural Selection

The origin of species is based on "Preservation of favorable variations and rejection of unfavorable variations." The variation refers to the differences shown by the individual of a species and also by offspring's of the same parents. There are more individuals born than can survive, so there is a continuous struggle for life. Individuals with an advantage have a greater chance of survival, i.e., the survival of the fittest. For example, Giraffe with long necks can have food from tall trees as well from the ground; on the other hand, goat and deer having smaller neck can have food only from the ground. As a result, natural selection plays a major role in this survival process.

Table 15.1 gives a list of different expressions, which are common in natural evolution and genetic algorithm.

15.3 Traditional Optimization and Search Techniques

The basic principle of optimization is the efficient allocation of scarce resources. Optimization can be applied to any scientific or engineering discipline. The aim of optimization is to find an algorithm which solves a given class of problems. There exists no specific method which solves all optimization problems. Consider a function,

$$f(x) : [x^l, x^u] \rightarrow [0, 1] \tag{15.1}$$

where

$$f(x) = \begin{cases} 1 & \text{if } ||x - a|| < \epsilon,\ \epsilon > 0 \\ -1 & \text{elsewhere} \end{cases}$$

For the above function, f can be maintained by decreasing ϵ or by making the interval of $[x^l, x^u]$ large. Thus, a difficult task can be made easier. Therefore, one can solve optimization problems by combining human creativity and the raw processing power of the computers.

The various conventional optimization and search techniques available are discussed in the following subsections.

15.3.1 Gradient-Based Local Optimization Method

When the objective function is smooth and one needs efficient local optimization, it is better to use gradient-based or Hessian-based optimization methods. The performance and reliability of the different gradient methods vary considerably. To discuss gradient-based local optimization, let us assume a smooth objective function (i.e., continuous first and second derivatives). The object function is denoted by

$$f(x) : R^n \rightarrow R \tag{15.2}$$

The first derivatives are contained in the gradient vector $\nabla f(x)$

$$\nabla f(x) = \begin{bmatrix} \partial f(x)/\partial x_1 \\ \vdots \\ \partial f(x)/\partial x_n \end{bmatrix} \tag{15.3}$$

The second derivatives of the object function are contained in the Hessian matrix $H(x)$:

$$H(x) = \nabla^T \nabla f(x) = \begin{pmatrix} \dfrac{\partial^2 f(x)}{\partial^2 x_1} & \cdots & \dfrac{\partial^2 f(x)}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \dfrac{\partial^2 f(x)}{\partial x_1 \partial x_n} & \cdots & \dfrac{\partial^2 f(x)}{\partial^2 x_n} \end{pmatrix} \tag{15.4}$$

Few methods need only the gradient vector, but in the Newton's method we need the Hessian matrix. The general pseudocode used in gradient methods is as follows:

Select an initial guess value x^1 and set $n = 1$.

Repeat

Solve the search direction P^n from Eq. (15.5) or (15.6) below.

Determine the next iteration point using Eq. (15.7) below:

$$X^{n+1} = X^n + \lambda_n P^n$$

Set $n = n + 1$.

Until $\|X^n - X^{n-1}\| < \epsilon$

These gradient methods search for minimum and not maximum. Several different methods are obtained based on the details of the algorithm.

The search direction P^n in conjugate gradient method is found as follows:

$$P^n = -\nabla f(X^n) + \beta_n P^{n-1} \tag{15.5}$$

In secant method,

$$B_n P^n = -\nabla f(x^n) \tag{15.6}$$

is used for finding search direction. The matrix B_n in Eq. (15.6) estimates the Hessian and is updated in each iteration. When B_n is defined as the identity matrix, the steepest descent method occurs. When the matrix B_n is the Hessian $H(x^n)$, we get the Newton's method.

The length λ_n of the search step is computed using:

$$\lambda_n = \arg\min_{\lambda > 0} f(x^n + \lambda P^n) \tag{15.7}$$

The discussed is a one-dimensional optimization problem. The steepest descent method provides poor performance. As a result, conjugate gradient method can be used. If the second derivatives are easy to compute, then Newton's method may provide best results. The secant methods are faster than conjugate gradient methods, but there occurs memory problems. Thus, these local optimization methods can be combined with other methods to get a good link between performance and reliability.

15.3.2 Random Search

Random search is an extremely basic method. It only explores the search space by randomly selecting solutions and evaluates their fitness. This is quite an unintelligent strategy, and is rarely used. Nevertheless, this method is sometimes worth testing. It doesn't take much effort to implement it, and an important number of evaluations can be done fairly quickly. For new unresolved problems, it can be useful to compare the results of a more advanced algorithm to those obtained just with a random search for the same number of evaluations. Nasty surprises might well appear when comparing, for example, GAs to random search. It's good to remember that the efficiency of GA is extremely dependent on consistent coding and relevant reproduction operators. Building a GA which performs no more than a random search happens more often than we can expect. If the reproduction operators are just producing new random solutions without any concrete links to the ones selected from the last generation, the GA is just doing nothing else than a random search.

Random search does have a few interesting qualities. However good the obtained solution may be, if it's not optimal one, it can be always improved by continuing the run of the random search algorithm for long enough. A random search never gets stuck at any point such as a local optimum. Furthermore, theoretically, if the search space is finite, random search is guaranteed to reach the optimal solution. Unfortunately, this result is completely useless. For most of problems we are interested in, exploring the whole search space takes a lot of time.

15.3.3 Stochastic Hill Climbing

Efficient methods exist for problems with well-behaved continuous fitness functions. These methods use a kind of gradient to guide the direction of search. *Stochastic hill climbing* is the simplest method of these kinds. Each iteration consists in choosing randomly a solution in the neighborhood of the current solution and retains this new solution only if it improves the fitness function. Stochastic hill climbing converges towards the optimal solution if the fitness function of the problem is continuous and has only one peak (unimodal function).

On functions with many peaks (multimodal functions), the algorithm is likely to stop on the first peak it finds even if it is not the highest one. Once a peak is reached, hill climbing cannot progress anymore, and that is problematic when this point is a local optimum. Stochastic hill climbing usually starts from a random select point. A simple idea to avoid getting stuck on the first local optimal consists in repeating several hill climbs each time starting from a different randomly chosen point. This method is sometimes known as *iterated hill climbing*. By discovering different local optimal points, chances to reach the global optimum increase. It works well if there are not too many local optima in the search space. However, if the fitness function is very "noisy" with many small peaks, stochastic hill climbing is definitely not a good method to use. Nevertheless, such methods have the advantage of being easy to implement and giving fairly good solutions very quickly.

15.3.4 Simulated Annealing

Simulated annealing (SA) was originally inspired by formation of crystal in solids during cooling. As discovered a long time ago by Iron Age blacksmiths, the slower the cooling, the more perfect is the crystal formed. By cooling, complex physical systems naturally converge towards a state of minimal energy. The system moves randomly, but the probability to stay in a particular configuration depends directly on the energy of the system and on its temperature. This probability is formally given by Gibbs law:

$$p = e^{E/kT} \tag{15.8}$$

where E stands for the energy, k is the Boltzmann constant and T is the temperature. In the mid01970s, Kirkpatrick by analogy of this physical phenomena, laid out the first description of SA.

As in the stochastic hill climbing, the iteration of the SA consists of randomly choosing a new solution in the neighborhood of the actual solution. If the fitness function of the new solution is better than the fitness function of the current one, the new solution is accepted as the new current solution. If the fitness function is not improved, the new solution is retained with a probability:

$$p = e^{-|f(y)-f(x)|/kT} \tag{15.9}$$

where $f(y) - f(x)$ is the difference of the fitness function between the new and the old solution.

The SA behaves like a hill climbing method but with the possibility of going downhill to avoid being trapped at local optima. When the temperature is high, the probability of deteriorate the solution is quite important, and then a lot of large moves are possible to explore the search space. The more the temperature decreases, the more difficult it is to go downhill. The algorithm thus tries to climb up from the current solution to reach a maximum. When temperature is lower, there is an exploitation of the current solution. If the temperature is too low, number deterioration is accepted, and the algorithm behaves just like a stochastic hill climbing method. Usually, the SA starts from a high temperature which decreases exponentially. The slower the cooling, the better it is for finding good solutions. It even has been demonstrated that with an infinitely slow cooling, the algorithm is almost certain to find the global optimum. The only point is that infinitely slow cooling consists in finding the appropriate temperature decrease rate to obtain a good behavior of the algorithm.

SA by mixing exploration features such as the random search and exploitation features like hill climbing usually gives quite good results. SA is a serious competitor of GAs. It is worth trying to compare the results obtained by each. Both are derived from analogy with natural system evolution and both deal with the same kind of optimization problem. GAs differ from SA in two main features which makes them more efficient. First, GAs use a population-based selection whereas SA only deals with one individual at each iteration. Hence GAs are expected to cover a much larger landscape of the search space at each iteration; however, SA iterations are much more simple, and so, often much faster. The great advantage of GA is its exceptional ability to be parallelized, whereas SA does not gain much of this. It is mainly due to the population scheme use by GA. Second, GAs use recombination operators, and are able to mix good characteristics from different solutions. The exploitation made by recombination operators are supposedly considered helpful to find optimal solutions of the problem. On the other hand, SA is still very simple to implement and gives good results. SAs have proved their efficiency over a large spectrum of difficult problems, like the optimal layout of printed circuit board or the famous traveling salesman problem.

15.3.5 Symbolic Artificial Intelligence

Most symbolic artificial intelligence (AI) systems are very static. Most of them can usually only solve one given specific problem, since their architecture was designed for whatever that specific problem was in the first place. Thus, if the given problem were somehow to be changed, these systems could have a hard time adapting to them, since the algorithm that would originally arrive to the solution may be either incorrect or less efficient. GAs were created to combat these problems. They are basically algorithms based on natural biological evolution. The architecture of systems that implement GAs is more able to adapt to a wide range of

problems. A GA functions by generating a large set of possible solutions to a given problem. It then evaluates each of those solutions, and decides on a "fitness level" (you may recall the phrase: "survival of the fittest") for each solution set. These solutions then breed new solutions. The parent solutions that were more "fit" are more likely to reproduce, while those that were less "fit" are more unlikely to do so. In essence, solutions are evolved over time. This way we evolve our search space scope to a point where you can find the solution. GAs can be incredibly efficient if programmed correctly.

15.4 Genetic Algorithm and Search Space

Evolutionary computing was introduced in the 1960s by I. Rechenberg in the work "Evolution Strategies." This idea was then developed by other researches. GAs were invented by John Holland and developed this idea in his book "Adaptation in Natural and Artificial Systems" in the year 1975. Holland proposed GA as a heuristic method based on "survival of the fittest." GA was discovered as a useful tool for search and optimization problems.

15.4.1 Search Space

Most often one is looking for the best solution in a specific set of solutions. The space of all feasible solutions (the set of solutions among which the desired solution resides) is called *search space* (also state space). Each and every point in the search space represents one possible solution. Therefore, each possible solution can be "marked" by its fitness value, depending on the problem definition. With GA one looks for the best solution among a number of possible solutions – represented by one point in the search space; GAs are used to search the search space for the best solution, e.g., minimum. The difficulties in this case are the local minima and the starting point of the search. Figure 15-6 gives an example of search space.

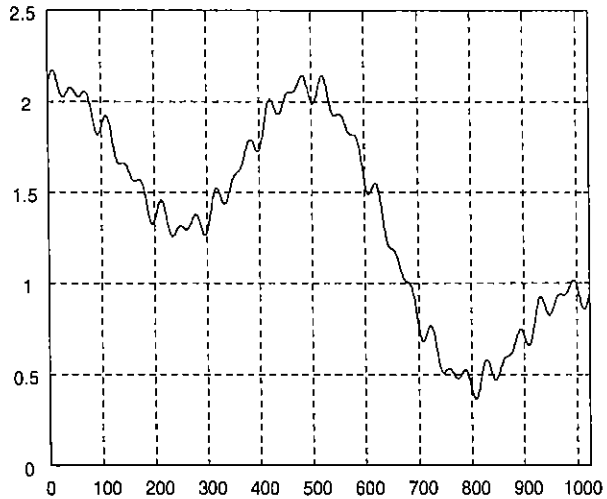


Figure 15-6 An example of search space.

15.4.2 Genetic Algorithms World

GA raises again a couple of important features. First, it is a *stochastic* algorithm; randomness has an essential role in GAs. Both selection and reproduction need random procedures. A second very important point is that GAs always consider a population of solutions. Keeping in memory more than a single solution at each iteration offers a lot of advantages. The algorithm can recombine different solutions to get better ones and so it can use the benefits of assortment. A population-based algorithm is also very amenable for parallelization. The *robustness* of the algorithm should also be mentioned as something essential for the algorithm's success. Robustness refers to the ability to perform consistently well on a broad range of problem types. There is no particular requirement on the problem before using GAs, so it can be applied to resolve any problem. All these features make GA a really powerful optimization tool.

With the success of GAs, other algorithms making use of the same principle of natural evolution have also emerged. Evolution strategy, genetic programming are some algorithms similar to these algorithms. The classification is not always clear between the different algorithms, thus to avoid any confusion, they are all gathered in what is called *Evolutionary Algorithms*.

The analogy with nature gives these algorithms something exciting and enjoyable. Their ability to deal successfully with a wide range of problem area, including those which are difficult for other methods to solve makes them quite powerful. However today, GAs are suffering from too much trendiness. GA is a new field, and parts of the theory still have to be properly established. We can find almost as many opinions on GAs as there are researchers in this field. In this document, we will generally find the most current point of view. But things evolve quickly in GAs too, and some comments might not be very accurate in few years.

It is also important to mention GA limits in this introduction. Like most stochastic methods, GAs are not guaranteed to find the global optimum solution to a problem; they are satisfied with finding "acceptably good" solutions to the problem. GAs are extremely general too, and so specific techniques for solving particular problems are likely to out-perform GAs in both speed and accuracy of the final result. GAs are something worth trying when everything else fails or when we know absolutely nothing of the search space. Nevertheless, even when such specialized techniques exist, it is often interesting to hybridize them with a GA in order to possibly gain some improvements. It is important always to keep an objective point of view; do not consider that GAs are a panacea for resolving all optimization problems. This warning is for those who might have the temptation to resolve anything with GA. The proverb says "If we have a hammer, all the problems look like a nails." GAs do work and give excellent results if they are applied properly on appropriate problems.

15.4.3 Evolution and Optimization

To depict the importance of evolution and optimization process, consider a species Basilosaurus that originated 45 million years ago. The Basilosaurus was a prototype of a whale (Figure 15-7). It was about 15 m long and

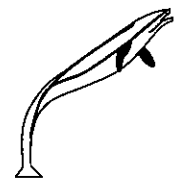


Figure 15-7 Basilosaurus.

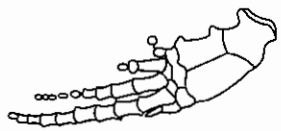


Figure 15-8 Tursiops flipper.

weighed approximately 5 tons. It still had a quasi-independent head and posterior paws, and moved using undulatory movements and hunted small preys. Its anterior members were reduced to small flippers with an elbow articulation. Movements in such a viscous element (water) are very hard and require big efforts. The anterior members of basilosaurus were not really adapted to swimming. To adapt them, a double phenomenon must occur: the shortening of the "arm" with the locking of the elbow articulation and the extension of the fingers constitute the base structure of the flipper (refer Figure 15-8).

The image shows that two fingers of the common dolphin are hypertrophied to the detriment of the rest of the member. The basilosaurus was a hunter; it had to be fast and precise. Through time, subjects appeared with longer fingers and short arms. They could move faster and more precisely than before, and therefore, live longer and have many descendants.

Meanwhile, other improvements occurred concerning the general aerodynamic like the integration of the head to the body, improvement of the profile, strengthening of the caudal fin, and so on, finally producing a subject perfectly adapted to the constraints of an aqueous environment. This process of adaptation and this morphological optimization is so perfect that nowadays the similarity between a shark, a dolphin or a submarine is striking. The first is a cartilaginous fish (Chondrichtyen) that originated in the Devonian period (~400 million years), long before the apparition of the first mammal. Darwinian mechanism hence generated an optimization process – *hydrodynamic optimization* – for fishes and others marine animals – *aerodynamic optimization* for pterodactyls, birds and bats. This observation is the basis of GAs.

15.4.4 Evolution and Genetic Algorithms

The basic idea is as follows: the genetic pool of a given population potentially contains the solution, or a better solution, to a given adaptive problem. This solution is not "active" because the genetic combination on which it relies is split among several subjects. Only the association of different genomes can lead to the solution. Simplistically speaking, we could by example consider that the shortening of the paw and the extension of the fingers of our basilosaurus are controlled by two "genes." No subject has such a genome, but during reproduction and crossover, new genetic combination occur and, finally, a subject can inherit a "good gene" from both parents his paw is now a flipper.

Holland method is especially effective because he not only considered the role of mutation (mutations improve very seldom the algorithms), but also utilized genetic recombination (crossover): these recombination, the crossover of partial solutions, greatly improve the capability of the algorithm to approach, and eventually find, the optimum.

Recombination or sexual reproduction is a key operator for natural evolution. Technically, it takes two genotypes and it produces a new genotype by mixing the gene found in the originals. In biology, the most common form of recombination is crossover: two chromosomes are cut at one point and the halves are spliced to create new chromosomes. The effect of recombination is very important because it allows characteristics from two different parents to be assorted. If the father and the mother possess different good qualities, we would expect that all the good qualities will be passed to the child. Thus the offspring, just by combining all

the good features from its parents, may surpass its ancestors. Many people believe that this mixing of genetic material via sexual reproduction is one of the most powerful features of GAs. As a quick parenthesis about sexual reproduction, GA representation usually does not differentiate male and female individuals (without any perversity). As in many livings species (e.g., snails) any individual can be either a male or a female. In fact, for almost all recombination operators, mother and father are interchangeable.

Mutation is the other way to get new genomes. Mutation consists in changing the value of genes. In natural evolution, mutation mostly engenders non-viable genomes. Actually mutation is not a very frequent operator in natural evolution. Nevertheless, in optimization, a few random changes can be a good way of exploring the search space quickly.

Through those low-level notions of genetic, we have seen how living beings store their characteristic information and how this information can be passed into their offspring. It very basic but it is more than enough to understand the GA theory.

Darwin was totally unaware of the biochemical basics of genetics. Now we know how the genetic inheritable information is coded in DNA, RNA, and proteins and that the coding principles are actually digital, much resembling the information storage in computers. Information processing is in many ways totally different, however. The magnificent phenomenon called the evolution of species can also give some insight into information processing methods and optimization, in particular. According to Darwinism, inherited variation is characterized by the following properties:

1. Variation must be copying because selection does not create directly anything, but presupposes a large population to work on.
2. Variation must be small-scaled in practice. Species do not appear suddenly.
3. Variation is undirected. This is also known as the blind watch maker paradigm.

While the natural sciences approach to evolution has for over a century been to analyze and study different aspects of evolution to find the underlying principles, the engineering sciences are happy to apply evolutionary principles, that have been heavily tested over billions of years, to attack the most complex technical problems, including protein folding.

15.5 Genetic Algorithm vs. Traditional Algorithms

The principle of GAs is simple: imitate genetics and natural selection by a computer program: The parameters of the problem are coded most naturally as a DNA – like linear data structure, a vector or a string. Sometimes, when the problem is naturally two or three dimensional, corresponding array structures are used.

A set, called *population*, of these problem-dependent parameter value vectors is processed by GA. To start, there is usually a totally random population, the values of different parameters generated by a random number generator. Typical population size is from few dozens to thousands. To do optimization we need a cost function or fitness function as it is usually called when GAs are used. By a fitness function we can select the best solution candidates from the population and delete the not so good specimens.

The nice thing when comparing GAs to other optimization methods is that the fitness function can be nearly anything that can be evaluated by a computer or even something that cannot! In the latter case it might be a human judgment that cannot be stated as a crisp program, like in the case of eye witness, where a human being selects from the alternatives generated by GA. So, there are not any definite mathematical restrictions on the properties of the fitness function. It may be discrete, multimodal, etc.

The main criteria used to classify optimization algorithms are as follows: continuous/discrete, constrained/unconstrained and sequential/parallel. There is a clear difference between discrete and continuous problems. Therefore, it is instructive to notice that continuous methods are sometimes used to solve inherently discrete problems and vice versa. Parallel algorithms are usually used to speed up processing. There are, however, some cases in which it is more efficient to run several processors in parallel rather than sequentially. These cases include among others those in which there is high probability of each individual search run to get stuck into a local extreme.

Irrespective of the above classification, optimization methods can be further classified into deterministic and non-deterministic methods. In addition, optimization algorithms can be classified as local or global. In terms of energy and entropy local search corresponds to entropy while global optimization depends essentially on the fitness, i.e., energy landscape.

GA differs from conventional optimization techniques in following ways:

1. GAs operate with coded versions of the problem parameters rather than parameters themselves, i.e., GA works with the coding of solution set and not with the solution itself.
2. Almost all conventional optimization techniques search from a single point, but GAs always operate on a whole population of points (strings), i.e., GA uses population of solutions rather than a single solution for searching. This plays a major role to the robustness of GAs. It improves the chance of reaching the global optimum and also helps in avoiding local stationary point.
3. GA uses fitness function for evaluation rather than derivatives. As a result, they can be applied to any kind of continuous or discrete optimization problem. The key point to be performed here is to identify and specify a meaningful decoding function.
4. GAs use probabilistic transition operates while conventional methods for continuous optimization apply deterministic transition operates, i.e., GAs do not use deterministic rules.

These are the major differences that exist between GA and conventional optimization techniques.

15.6 Basic Terminologies in Genetic Algorithm

The two distinct elements in the GA are individuals and populations. An individual is a single solution while the population is the set of individuals currently involved in the search process.

15.6.1 Individuals

An individual is a single solution. An individual groups together two forms of solutions as given below:

1. The chromosome which is the raw "genetic" information (genotype) that the GA deals.
2. The phenotype which is the expressive of the chromosome in the terms of the model.

A chromosome is subdivided into genes. A gene is the GA's representation of a single factor for a control factor. Each factor in the solution set corresponds to a gene in the chromosome. Figure 15-9 shows the representation of a genotype.

A chromosome should in some way contain information about the solution that it represents. The morphogenesis function associates each genotype with its phenotype. It simply means that each chromosome must define one unique solution, but it does not mean that each solution is encoded by exactly one chromosome. Indeed, the morphogenesis function is not necessarily bijective, and it is even sometimes impossible (especially with binary representation). Nevertheless, the morphogenesis function should at least be subjective. Indeed;

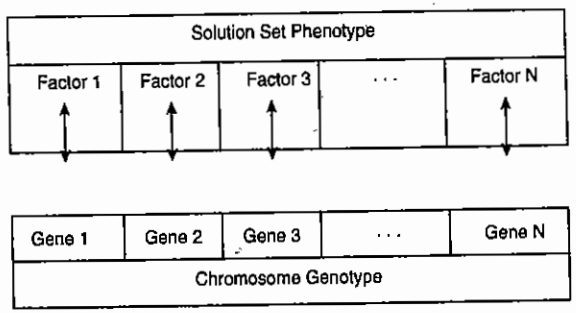


Figure 15-9 Representation of genotype and phenotype.

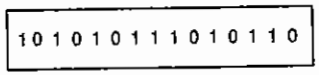


Figure 15-10 Representation of a chromosome.

all the candidate solutions of the problem must correspond to at least one possible chromosome, to be sure that the whole search space can be explored. When the morphogenesis function that associates each chromosome to one solution is not injective, i.e., different chromosomes can encode the same solution, the representation is said to be degenerated. A slight degeneracy is not so worrying, even if the space where the algorithm is looking for the optimal solution is inevitably enlarged. But a too important degeneracy could be a more serious problem. It can badly affect the behavior of the GA, mostly because if several chromosomes can represent the same phenotype, the meaning of each gene will obviously not correspond to a specific characteristic of the solution. It may add some kind of confusion in the search. Chromosomes encoded by bit strings are given in Figure 15-10.

15.6.2 Genes

Genes are the basic "instructions" for building a GA. A chromosome is a sequence of genes. Genes may describe a possible solution to a problem, without actually being the solution. A gene is a bit string of arbitrary lengths. The bit string is a binary representation of number of intervals from a lower bound. A gene is the GA's representation of a single factor value for a control factor, where control factor must have an upper bound and a lower bound. This range can be divided into the number of intervals that can be expressed by the gene's bit string. A bit string of length "n" can represent $(2^n - 1)$ intervals. The size of the interval would be (range)/$(2^n - 1)$.

The structure of each gene is defined in a record of phenotyping parameters. The phenotype parameters are instructions for mapping between genotype and phenotype. It can also be said as encoding a solution set into a chromosome and decoding a chromosome to a solution set. The mapping between genotype and phenotype is necessary to convert solution sets from the model into a form that the GA can work with, and for converting new individuals from the GA into a form that the model can evaluate. In a chromosome, the genes are represented as shown in Figure 15-11.

15.6.3 Fitness

The fitness of an individual in a GA is the value of an objective function for its phenotype. For calculating fitness, the chromosome has to be first decoded and the objective function has to be evaluated. The fitness

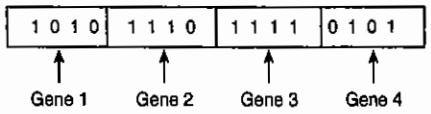


Figure 15-11 Representation of a gene.

not only indicates how good the solution is, but also corresponds to how close the chromosome is to the optimal one.

In the case of multicriterion optimization, the fitness function is definitely more difficult to determine. In multicriterion optimization problems, there is often a dilemma as how to determine if one solution is better than another. What should be done if a solution is better for one criterion but worse for another? But here, the trouble comes more from the definition of a "better" solution rather than from how to implement a GA to resolve it. If sometimes a fitness function obtained by a simple combination of the different criteria can give good result, it supposes that criterions can be combined in a consistent way. But, for more advanced problems, it may be useful to consider something like Pareto optimally or other ideas from multicriteria optimization theory.

15.6.4 Populations

A population is a collection of individuals. A population consists of a number of individuals being tested, the phenotype parameters defining the individuals and some information about the search space. The two important aspects of population used in GAs are:

1. The initial population generation.
2. The population size.

For each and every problem, the population size will depend on the complexity of the problem. It is often a random initialization of population. In the case of a binary coded chromosome this means that each bit is initialized to a random 0 or 1. However, there may be instances where the initialization of population is carried out with some known good solutions.

Ideally, the first population should have a gene pool as large as possible in order to be able to explore the whole search space. All the different possible alleles of each should be present in the population. To achieve this, the initial population is, in most of the cases, chosen randomly. Nevertheless, sometimes a kind of heuristic can be used to seed the initial population. Thus, the mean fitness of the population is already high and it may help the GA to find good solutions faster. But for doing this one should be sure that the gene pool is still large enough. Otherwise, if the population badly lacks diversity, the algorithm will just explore a small part of the search space and never find global optimal solutions.

The size of the population raises few problems too. The larger the population is, the easier it is to explore the search space. However, it has been established that the time required by a GA to converge is $O(n \log n)$ function evaluations where n is the population size. We say that the population has converged when all the individuals are very much alike and further improvement may only be possible by mutation. Goldberg has also shown that GA efficiency to reach global optimum instead of local ones is largely determined by the size of the population. To sum up, a large population is quite useful. However, it requires much more computational cost memory and time. Practically, a population size of around 100 individuals is quite frequent, but anyway this size can be changed according to the time and the memory disposed on the machine compared to the quality of the result to be reached.

Population	Chromosome 1	1 1 1 0 0 0 1 0
	Chromosome 2	0 1 1 1 1 0 1 1
	Chromosome 3	1 0 1 0 1 0 1 0
	Chromosome 4	1 1 0 0 1 1 0 0

Figure 15-12 Population.

Population being combination of various chromosomes is represented as in Figure 15-12. Thus the population in Figure 15-12 consists of four chromosomes.

15.7 Simple GA

GA handles a population of possible solutions. Each solution is represented through a chromosome, which is just an abstract representation. Coding all the possible solutions into a chromosome is the first part, but certainly not the most straightforward one of a GA. A set of reproduction operators has to be determined, too. Reproduction operators are applied directly on the chromosomes, and are used to perform mutations and recombinations over solutions of the problem. Appropriate representation and reproduction operators are the determining factors, as the behavior of the GA is extremely dependent on it. Frequently, it can be extremely difficult to find a representation that respects the structure of the search space and reproduction operators that are coherent and relevant according to the properties of the problems.

The simple form of GA is given by the following.

1. Start with a randomly generated population.
2. Calculate the fitness of each chromosome in the population.
3. Repeat the following steps until n offsprings have been created:
   - Select a pair of parent chromosomes from the current population.
   - With probability p_c crossover the pair at a randomly chosen point to form two offsprings.
   - Mutate the two offsprings at each locus with probability p_m.
4. Replace the current population with the new population.
5. Go to step 2.

Now we discuss each iteration of this process.

Generation: Selection: is supposed to be able to compare each individual in the population. Selection is done by using a fitness function. Each chromosome has an associated value corresponding to the fitness of the solution it represents. The fitness should correspond to an evaluation of how good the candidate solution is. The optimal solution is the one which maximizes the fitness function. GAs deal with the problems that maximize the fitness function. But, if the problem consists of minimizing a cost function, the adaptation is quite easy. Either the cost function can be transformed into a fitness function, for example by inverting it; or the selection can be adapted in such way that they consider individuals with low evaluation functions as better. Once the reproduction and the fitness function have been properly defined, a GA is evolved according to the same basic structure. It starts by generating an initial population of chromosomes. This first population must

offer a wide diversity of genetic materials. The gene pool should be as large as possible so that any solution of the search space can be engendered. Generally, the initial population is generated randomly. Then, the GA loops over an iteration process to make the population evolve. Each iteration consists of the following steps:

1. *Selection*: The first step consists in selecting individuals for reproduction. This selection is done randomly with a probability depending on the relative fitness of the individuals so that best ones are often chosen for reproduction rather than the poor ones.
2. *Reproduction*: In the second step, offspring are bred by selected individuals. For generating new chromosomes, the algorithm can use both recombination and mutation.
3. *Evaluation*: Then the fitness of the new chromosomes is evaluated.
4. *Replacement*: During the last step, individuals from the old population are killed and replaced by the new ones.

The algorithm is stopped when the population converges toward the optimal solution.

```
BEGIN /* genetic algorithm*/
  Generate initial population;
  Compute fitness of each individual;
  WHILE NOT finished DO LOOP
    BEGIN
        Select individuals from old generations
          For mating;
        Create offspring by applying
          recombination and/or mutation
          to the selected individuals;
        Compute fitness of the new individuals;
        Kill old individuals to make room for
          new chromosomes and insert
          offspring in the new generalization;
        IF Population has converged
          THEN finishes: = TRUE;
    END
END
```

Genetic algorithms are not too hard to program or understand because they are biological based. An example of a flowchart of a GA is shown in Figure 15-13.

15.8 General Genetic Algorithm

The general GA is as follows:

Step 1: *Create a random initial state*: An initial population is created from a random selection of solutions (which are analogous to chromosomes). This is unlike the situation for symbolic AI systems, where the initial state in a problem is already given.

Step 2: *Evaluate fitness*: A value for fitness is assigned to each solution (chromosome) depending on how close it actually is to solving the problem (thus arriving to the answer of the desired problem). (These "solutions" are not to be confused with "answers" to the problem; think of them as possible characteristics that the system would employ in order to reach the answer.)

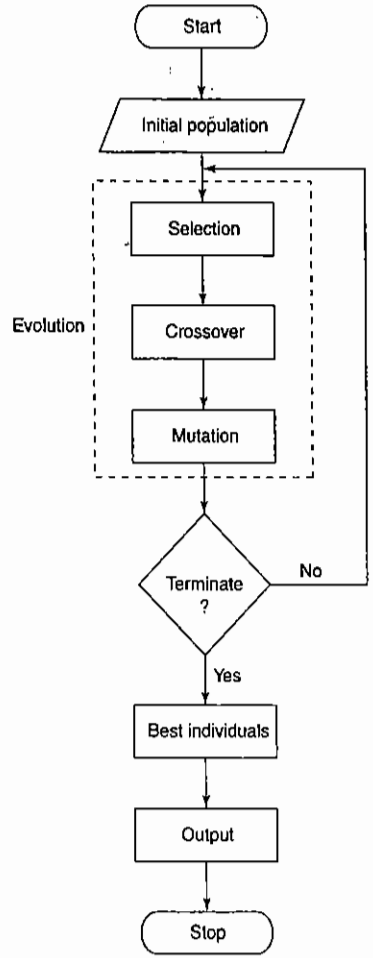


Figure 15-13 Flowchart for genetic algorithm.

Step 3: *Reproduce (and children mutate)*: Those chromosomes with a higher fitness value are more likely to reproduce offspring (which can mutate after reproduction). The offspring is a product of the father and mother, whose composition consists of a combination of genes from the two (this process is known as "crossingover").

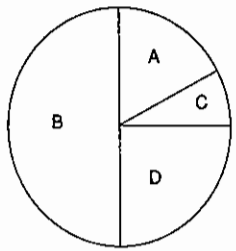
Step 4: *Next generation*: If the new generation contains a solution that produces an output that is close enough or equal to the desired answer then the problem has been solved. If this is not the case, then the new generation will go through the same process as their parents did. This will continue until a solution is reached.

Table 15-2 Fitness value for corresponding chromosomes (Example 15.1)

Chromosome	Fitness
A: 00000110	2
B: 11101110	6
C: 00100000	1
D: 00110100	3

Table 15-3 Fitness value for corresponding chromosomes

Chromosome	Fitness
A: 01101110	5
B: 00100000	1
C: 10110000	3
D: 01101110	5



Fitness-proportionate selection (Roulette wheel sampling)

Figure 15-14 Roulette wheel sampling for fitness-proportionate selection.

Example 15.1: Consider 8-bit chromosomes with the following properties:

1. Fitness function $f(x)$ = number of 1 bits in chromosome;
2. population size $N = 4$;
3. crossover probability $p_c = 0.7$;
4. mutation probability $p_m = 0.001$;

Average fitness of population = 12/4 = 3.0.

1. If B and C are selected, crossover is not performed.
2. If B is mutated, then

$$\text{B: } 11101110 \rightarrow \text{B}'\text{: } 01101110$$

3. If B and D are selected, crossover is performed.

$$\text{B: } 11101110 \text{ E: } 10110100 \rightarrow \text{D: } 00110100 \text{ F: } 01101110$$

4. If E is mutated, then

$$\text{E: } 10110100 \rightarrow \text{E}'\text{: } 10110000$$

Best-fit string from previous population is lost, but the average fitness of population is as given below:

$$\text{Average fitness of population } 14/4 = 3.5$$

Tables 15-2 and 15-3 show the fitness value for the corresponding chromosomes and Figure 15-14 shows the Roulette wheel selection for the fitness proportionate selection.

15.9 Operators in Genetic Algorithm

The basic operators that are to be discussed in this section include: encoding, selection, recombination and mutation operators. The operators with their various types are explained with necessary examples.

15.9.1 Encoding

Encoding is a process of representing individual genes. The process can be performed using bits, numbers, trees, arrays, lists or any other objects. The encoding depends mainly on solving the problem. For example, one can encode directly real or integer numbers.

15.9.1.1 Binary Encoding

The most common way of encoding is a binary string, which would be represented as in Figure 15-15.

Each chromosome encodes a binary (bit) string. Each bit in the string can represent some characteristics of the solution. Every bit string therefore is a solution but not necessarily the best solution. Another possibility is that the whole string can represent a number. The way bit strings can code differs from problem to problem.

Binary encoding gives many possible chromosomes with a smaller number of alleles. On the other hand, this encoding is not natural for many problems and sometimes corrections must be made after genetic operation is completed. Binary coded strings with 1s and 0s are mostly used. The length of the string depends on the accuracy. In such coding

1. Integers are represented exactly.
2. Finite number of real numbers can be represented.
3. Number of real numbers represented increases with string length.

15.9.1.2 Octal Encoding

This encoding uses string made up of octal numbers (0–7) (see Figure 15-16).

Chromosome 1	1 1 0 1 0 0 0 1 1 0 1 0
Chromosome 2	0 1 1 1 1 1 1 1 1 1 0 0

Figure 15-15 Binary encoding.

Chromosome 1	03467216
Chromosome 2	15723314

Figure 15-16 Octal encoding.

Chromosome 1	9CE7
Chromosome 2	3DBA

Figure 15-17 Hexadecimal encoding.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

Figure 15-18 Permutation encoding.

15.9.1.3 Hexadecimal Encoding

This encoding uses string made up of hexadecimal numbers (0–9, A–F) (see Figure 15-17).

15.9.1.4 Permutation Encoding (Real Number Coding)

Every chromosome is a string of numbers, represented in a sequence. Sometimes corrections have to be done after genetic operation is complete. In permuration encoding, every chromosome is a string of integer/real values, which represents number in a sequence.

Permutation encoding (Figure 15-18) is only useful for ordering problems. Even for this problem, some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e., have real sequence in it).

15.9.1.5 Value Encoding

Every chromosome is a string of values and the values can be anything connected to the problem. This encoding produces best results for some special problems. On the other hand, it is often necessary to develop new genetic operator's specific to the problem. Direct value encoding can be used in problems, where some complicated values, such as real numbers, are used. Use of binary encoding for this type of problems would be very difficult.

In value encoding (Figure 15-19), every chromosome is a string of some values. Values can be anything connected to problem, form numbers, real numbers or characters to some complicated objects. Value encoding is very good for some special problems. On the other hand, for this encoding it is often necessary to develop some new crossover and mutation specific for the problem.

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

Figure 15-19 Value encoding.

15.9.1.6 Tree Encoding

This encoding is mainly used for evolving program expressions for genetic programming. Every chromosome is a tree of some objects such as functions and commands of a programming language.

15.9.2 Selection

Selection is the process of choosing two parents from the population for crossing. After deciding on an encoding, the next step is to decide how to perform selection, i.e., how to choose individuals in the population that will create offspring for the next generation and how many offspring each will create. The purpose of selection is to emphasize fitter individuals in the population in hopes that their offspring have higher fitness. Chromosomes are selected from the initial population to be parents for reproduction. The problem is how to select these chromosomes. According to Darwin's theory of evolution the best ones survive to create new offspring. Figure 15-20 shows the basic selection process.

Selection is a method that randomly picks chromosomes out of the population according to their evaluation function. The higher the fitness function, the better chance that an individual will be selected. The selection pressure is defined as the degree to which the better individuals are favored. The higher the selection pressured, the more the better individuals are favored. This selection pressure drives the GA to improve the population fitness over successive generations.

The convergence rate of GA is largely determined by the magnitude of the selection pressure, with higher selection pressures resulting in higher convergence rates. GAs should be able to identify optimal or nearly optimal solutions under a wide range of selection scheme pressure. However, if the selection pressure is too low, the convergence rate will be slow, and the GA will take unnecessarily longer to find the optimal solution. If the selection pressure is too high, there is an increased change of the GA prematurely converging to an incorrect (sub-optimal) solution. In addition to providing selection pressure, selection schemes should also preserve population diversity, as this helps to avoid premature convergence.

Typically we can distinguish two types of selection scheme, proportionate-based selection and ordinal-based selection. Proportionate-based selection picks out individuals based upon their fitness values relative to the fitness of the other individuals in the population. Ordinal-based selection schemes select individuals not upon their raw fitness, but upon their rank within the population. This requires that the selection pressure is independent of the fitness distribution of the population, and is solely based upon the relative ordering (ranking) of the population.

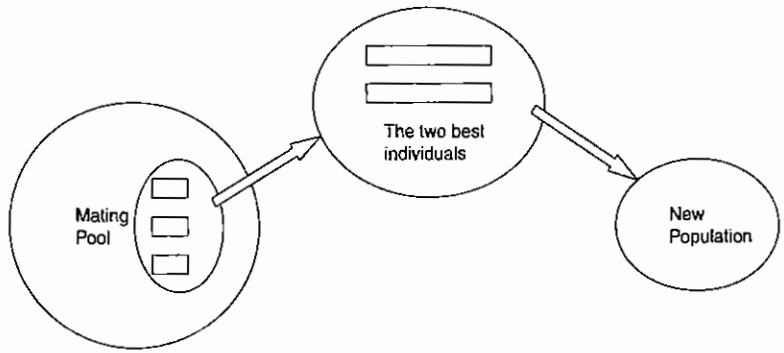


Figure 15-20 Selection.

It is also possible to use a scaling function to redistribute the fitness range of the population in order to adapt the selection pressure. For example, if all the solutions have their fitnesses in the range [999, 1000], the probability of selecting a better individual than any other using a proportionate-based method will not be important. If the fitness every individual is bringing to the range [0, 1] equitable, the probability of selecting good individual instead of bad one will be important.

Selection has to be balanced with variation from crossover and mutation. Too strong selection means sub-optimal highly fit individuals will take over the population, reducing the diversity needed for change and progress; too weak selection will result in too slow evolution. The various selection methods are discussed in the following subsections.

15.9.2.1 Roulette Wheel Selection

Roulette selection is one of the traditional GA selection techniques. The commonly used reproduction operator is the proportionate reproductive operator where a stting is selected from the mating pool with a probability proportional to the fitness. The principle of Roulette selection is a linear search through a Roulette wheel with the slots in the wheel weighted in proportion to the individual's fitness values. A target value is set, which is a random proportion of the sum of the fit nesses in the population. The population is stepped through until the target value is reached. This is only a moderately strong selection technique, since fit individuals are not guaranteed to be selected for, but somewhat have a greater chance. A fit individual will contribute more to the target value, but if it does not exceed it, the next chromosome in line has a chance, and it may be weak. It is essential that the population not be sorted by fitness, since this would dramatically bias the selection.

The Roulette process can also be explained as follows: The expected value of an individual is individual's fitness divided by the actual fitness of the population. Each individual is assigned a slice of the Roulette wheel, the size of the slice being proportional to the individual's fitness. The wheel is spun N times, where N is the number of individuals in the population. On each spin, the individual under the wheel's marker is selected to be in the pool of parents for the next generation. This method is implemented as follows:

1. Sum the total expected value of the individuals in the population. Let it be T.
2. Repeat N times:
   i. Choose a random integer "r" between 0 and T.
   ii. Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to "r." The individual whose expected value puts the sum over this limit is the one selected.

Roulette wheel selection is easier to implement but is noisy. The rate of evolution depends on the variance of fitness's in the population.

15.9.2.2 Random Selection

This technique randomly selects a parent from the population. In terms of disruption of genetic codes, random selection is a little more disruptive, on average, than Roulette wheel selection.

15.9.2.3 Rank Selection

The Roulette wheel will have a problem when the fitness values differ very much. If the best chromosome fitness is 90%, its circumference occupies 90% of Roulette wheel, and then other chromosomes have too few chances to be selected. Rank Selection ranks the population and every chromosome receives fitness from the ranking. The worst has fitness 1 and the best has fitness N. It results in slow convergence but prevents too

quick convergence. It also keeps up selection pressure when the fitness variance is low. It preserves diversity and hence leads to a successful search. In effect, potential parents are selected and a tournament is held to decide which of the individuals will be the parent. There are many ways this can be achieved and two suggestions are:

1. Select a pair of individuals at random. Generate a random number R between 0 and 1. If $R < r$ use the first individual as a parent. If the $R \geq r$ then use the second individual as the parent. This is repeated to select the second parent. The value of r is a parameter to this method.
2. Select two individuals at random. The individual with the highest evaluation becomes the parent. Repeat to find a second parent.

15.9.2.4 Tournament Selection

An ideal selection strategy should be such that it is able to adjust its selective pressure and population diversity so as to fine-tune GA search performance. Unlike, the Roulette wheel selection, the tournament selection strategy provides selective pressure by holding a tournament competition among Nu individuals.

The best individual from the tournament is the one with the highest fitness, who is the winner of Nu. Tournament competitions and the winner are then inserted into the mating pool. The tournament competition is repeated until the mating pool for generating new offspring is filled. The mating pool comprising the toutnament winner has higher average population fitness. The fitness difference provides the selection pressure, which drives GA to improve the fitness of the succeeding genes. This method is more efficient and leads to an optimal solution.

15.9.2.5 Boltzmann Selection

SA is a method of function minimization or maximization. This method simulates the process of slow cooling of molten metal to achieve the minimum function value in a minimization problem. Controlling a temperature-like parameter introduced with the concept of Boltzmann probability distribution simulates the cooling phenomenon.

In Boltzmann selection, a continuously varying temperature controls the rate of selection according to a preset schedule. The temperature starts out high, which means that the selection pressure is low. The temperature is gradually lowered, which gradually increases the selection pressure, thereby allowing the GA to narrow in more closely to the best part of the search space while maintaining the appropriate degree of diversity.

A logarithmically decreasing temperature is found useful for convergence without getting stuck to a local minima state. However, it takes time to cool down the system to the equilibrium state.

Let $f_{\max}$ be the fitness of the currently available best string. If the next string has fitness $f(X_i)$ such that $f(X_i) > f_{\max}$, then the new string is selected. Otherwise it is selected with Bolt/Mann probability

$$P = \exp[-\{f_{\max} - f(X_i)\}/T] \tag{15.10}$$

where $T = T_0(1-\alpha)^k$ and $k = (1 + 100 * g/G)$; g is the current generation number; G the maximum value of g. The value of α can be chosen from the range [0, 1] and that of T_0 from the range [5, 100]. The final state is reached when computation approaches zero value of T, i.e., the global solution is achieved at this point.

The probability that the best string is selected and introduced into the mating pool is very high. However, Elitism can be used to eliminate the chance of any undesired loss of information during the mutation stage. Moreover, the execution time is less.

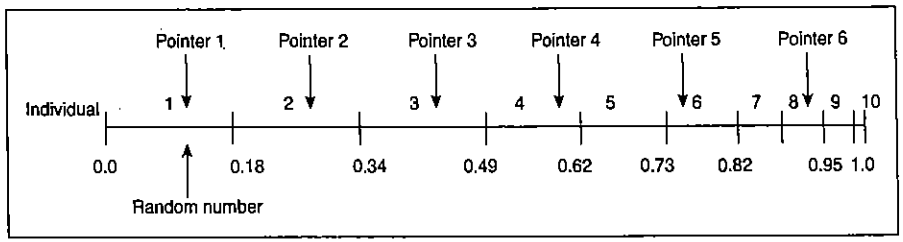


Figure 15-21 Stochastic universal sampling.

Elitism

The first best chromosome or the few best chromosomes are copied to the new population. The rest is done in a classical way. Such individuals can be lost if they are not selected to reproduce or if crossover or mutation destroys them. This significantly improves the GA's performance.

15.9.2.6 Stochastic Universal Sampling

Stochastic universal sampling provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in Roulette wheel selection. Here equally spaced pointers are placed over the line, as many as there are individuals to be selected. Consider *NPointer* the number of individuals to be selected, then the distance between the pointers are $1/NPointer$ and the position of the first pointer is given by a randomly generated number in the range $[0, 1/NPointer]$. For 6 individuals to be selected, the distance between the pointers is $1/6 = 0.167$. Figure 15-21 shows the selection for the above example.

Sample of 1 random number in the range [0, 0.167]: 0.1.
After selection the mating population consists of the individuals,

$$1, 2, 3, 4, 6, 8$$

Stochastic universal sampling ensures selection of offspring that is closer to what is deserved as compared to Roulette wheel selection.

15.9.3 Crossover (Recombination)

Crossover is the process of taking two parent solutions and producing from them a child. After the selection (reproduction) process, the population is enriched with better individuals. Reproduction makes clones of good strings but does not create new ones. Crossover operator is applied to the mating pool with the hope that it creates a better offspring.

Crossover is a recombination operator that proceeds in three steps:

1. The reproduction operator selects at random a pair of two individual strings for the mating.
2. A cross site is selected at random along the string length.
3. Finally, the position values are swapped between the two strings following the cross site.

That is the simplest way how to do that is to choose randomly some crossover point and copy everything before this point from the first parent and then copy everything after the crossover point from the other parent. The various crossover techniques are discussed in the following subsections.

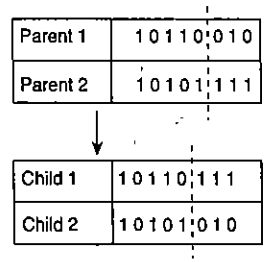


Figure 15-22 Single-point crossover.

15.9.3.1 Single-Point Crossover

The traditional genetic algorithm uses single-point crossover, where the two mating chromosomes are cut once at corresponding points and the sections after the cuts exchanged. Here, a cross site or crossover point is selected randomly along the length of the mated strings and bits next to the cross sites are exchanged. If appropriate site is chosen, better children can be obtained by combining good parents, else it severely hampers string quality.

Figure 15-22 illustrates single-point crossover and it can be observed that the bits next to the crossover point are exchanged to produce children. The crossover point can be chosen randomly.

15.9.3.2 Two-Point Crossover

Apart from single-point crossover, many different crossover algorithms have been devised, often involving more than one cut point. It should be noted that adding further crossover points reduces the performance of the GA. The problem with adding additional crossover points is that building blocks are more likely to be disrupted. However, an advantage of having more crossover points is that the problem space may be searched more thoroughly.

In two-point crossover, two crossover points are chosen and the contents between these points are exchanged between two mated parents.

In Figure 15-23 the dotted lines indicate the crossover points. Thus the contents between these points are exchanged between the parents to produce new children for mating in the next generation.

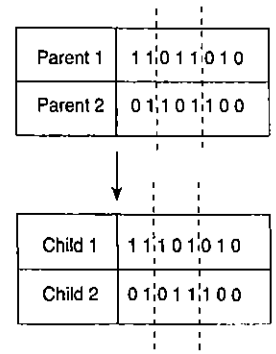


Figure 15-23 Two-point crossover.

Originally, GAs were using one-point crossover which cuts two chromosomes in one point and splices the two halves to create new ones. But with this one-point crossover, the head and the tail of one chromosome cannot be passed together to the offspring. If both the head and the tail of a chromosome contain good genetic information, none of the offspring obtained directly with one-point crossover will share the two good features. Using a two-point crossover one can avoid this drawback, and so it is generally considered better than one-point crossover. In fact, this problem can be generalized to each gene position in a chromosome. Genes that are close on a chromosome have more chance to be passed together to the offspring obtained through *N*-points crossover. It leads to an unwanted correlation between genes next to each other. Consequently, the efficiency of an *N*-point crossover will depend on the position of the genes within the chromosome. In a genetic representation, genes that encode dependent characteristics of the solution should be close together. To avoid all the problem of genes locus, a good thing is to use a uniform crossover as recombination operator.

15.9.3.3 Multipoint Crossover (N-Point Crossover)

There are two ways in this crossover. One is even number of cross sites and the other odd number of cross sites. In the case of even number of cross sites, the cross sites are selected randomly around a circle and information is exchanged. In the case of odd number of cross sites, a different cross point is always assumed at the string beginning.

15.9.3.4 Uniform Crossover

Uniform crossover is quite different from the *N*-point crossover. Each gene in the offspring is created by copying the corresponding gene from one or the other parent chosen according to a random generated binary crossover mask of the same length as the chromosomes. Where there is a 1 in the crossover mask, the gene is copied from the first parent, and where there is a 0 in the mask the gene is copied from the second parent. A new crossover mask is randomly generated for each pair of parents. Offspring, therefore, contain a mixture of genes from each parent. The number of effective crossing point is not fixed, but will average $L/2$ (where L is the chromosome length).

In Figure 15-24, new children are produced using uniform crossover approach. It can be noticed that while producing child 1, when there is a 1 in the mask, the gene is copied from parent 1 else it is copied from parent 2. On producing child 2, when there is a 1 in the mask, the gene is copied from parent 2, and when there is a 0 in the mask, the gene is copied from the parent 1.

15.9.3.5 Three-Parent Crossover

In this crossover technique, three parents are randomly chosen. Each bit of the first parent is compared with the bit of the second parent. If both are the same, the bit is taken for the offspring, otherwise the bit from the third parent is taken for the offspring. This concept is illustrated in Figure 15-25.

Parent 1	1 0 1 1 0 0 1 1
Parent 2	0 0 0 1 1 0 1 0
Mask	1 1 0 1 0 1 1 0
Child 1	1 0 0 1 1 0 1 0
Child 2	0 0 1 1 0 0 1 1

Figure 15-24 Uniform crossover.

Parent 1	1 1 0 1 0 0 0 1
Parent 2	0 1 1 0 1 0 0 1
Parent 3	0 1 1 0 1 1 0 0
Child	0 1 1 0 1 0 0 1

Figure 15-25 Three-parent crossover.

15.9.3.6 Crossover with Reduced Surrogate

The reduced surrogate operator constraints crossover to always produce new individuals wherever possible. This is implemented by restricting the location of crossover points such that crossover points only occur where gene values differ.

15.9.3.7 Shuffle Crossover

Shuffle crossover is related to uniform crossover. A single crossover position (as in single-point crossover) is selected. But before the variables are exchanged, they are randomly shuffled in both parents. After recombination, the variables in the offspring are unshuffled. This removes positional bias as the variables are randomly reassigned each time crossover is performed.

15.9.3.8 Precedence Preservative Crossover

Precedence preservative crossover (PPX) was independently developed for vehicle routing problems by Blanton and Wainwright (1993) and for scheduling problems by Bierwirth et al. (1996). The operator passes on precedence relations of operations given in two parental permutations to one offspring at the same rate, while no new precedence relations are introduced. PPX is illustrated below for a problem consisting of six operations A–F. The operator works as follows:

1. A vector of length Sigma, sub $i = 1$ to mi, representing the number of operations involved in the problem, is randomly filled with elements of the set {1, 2}.
2. This vector defines the order in which the operations are successively drawn from parent 1 and parent 2.
3. We can also consider the parent and offspring permutations as lists, for which the operations "append" and "delete" are defined.
4. First we start by initializing an empty offspring.
5. The lefrmost operation in one of the two parents is selected in accordance with the order of parents given in the vector.
6. After an operation is selected, it is deleted in both parents.
7. Finally the selected operation is appended to the offspring.
8. Step 7 is repeated until both parents are empty and the offspring contains all operations involved.

Note that PPX does not work in a uniform-crossover manner due to the "deletion–append" scheme used. Example is shown in Figure 15-26.

15.9.3.9 Ordered Crossover

Ordered two-point crossover is used when the problem is order based, for example in U-shaped assembly line balancing, etc. Given two parent chromosomes, two random crossover points are selected partitioning

Parent permutation 1	A B C D E F
Parent permutation 2	C A B F D E
Select parent no. (1/2)	1 2 1 1 2 2
Offspring permutation	A C B D F E

Figure 15-26 Precedence preservative crossover (PPX).

Parent 1: 4 2 | 1 3 | 6 5 Child 1: 4 2 | 3 1 | 6 5

Parent 2: 2 3 | 1 4 | 5 6 Child 2: 2 3 | 4 1 | 5 6

Figure 15-27 Ordered crossover.

them into a left, middle and right portions. The ordered two-point crossover behaves in the following way: child 1 inherits its left and right section from parent 1, and its middle section is determined by the genes in the middle section of parent 1 in the order in which the values appear in parent 2. A similar process is applied to determine child 2. This is shown in Figure 15-27.

15.9.3.10 Partially Matched Crossover

Partially matched crossover (PMX) can be applied usefully in the TSP. Indeed, TSP chromosomes are simply sequences of integers, where each integer represents a different city and the order represents the time at which a city is visited. Under this representation, known as permutation encoding, we are only interested in labels and not alleles. It may be viewed as a crossover of permutations that guarantees that all positions are found exactly once in each offspring, i.e., both offspring receive a full complement of genes, followed by the corresponding filling in of alleles from their parents. PMX proceeds as follows:

1. The two chromosomes are aligned.
2. Two crossing sites are selected uniformly at random along the strings, defining a matching section.
3. The matching section is used to effect a cross through position-by-position exchange operation.
4. Alleles are moved to their new positions in the offspring.

The following illustrates how PMX works.

Name 9 8 4 . 5 6 7 . 1 3 2 1 0	Allele 1 0 1 . 0 0 1 . 1 1 0 0
Name 8 7 1 . 2 3 1 0 . 9 5 4 6	Allele 1 1 1 . 0 1 1 . 1 1 0 1

Figure 15-28 Given strings.

Consider the two strings shown in Figure 15-28, where the dots mark the selected cross points. The matching section defines the position-wise exchanges that must take place in both parents to produce the offspring. The exchanges are read from the matching section of one chromosome to that of the other. In the example illustrate in Figure 15-28, the numbers that exchange places are 5 and 2, 6 and 3, and 7 and 10. The resulting offspring are as shown in Figure 15-29. PMX is dealt in detail in the next chapter.

Name 9 8 4 . 2 3 1 0 . 1 6 5 7	**Allele** 1 0 1 . 0 1 0 . 1 0 0 1
Name 8 1 0 1 . 5 6 7 . 9 2 4 3	**Allele** 1 1 1 . 1 1 1 . 1 0 0 1

Figure 15-29 Partially matched crossover.

15.9.3.11 Crossover Probability

The basic parameter in crossover technique is the crossover probability (P_c). Crossover probability is a parameter to describe how often crossover will be performed. If there is no crossover, offspring are exact copies of parents. If there is crossover, offspring are made from parts of both parents' chromosome. If crossover probability is 100%, then all offspring are made by crossover. If it is 0%, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!). Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.

15.9.4 Mutation

After crossover, the strings are subjected to mutation. Mutation prevents the algorithm to be trapped in a local minimum. Mutation plays the role of recovering the lost genetic materials as well as for randomly distributing genetic information. It is an insurance policy against the irreversible loss of genetic material. Mutation has been traditionally considered as a simple search operator. If crossover is supposed to exploit the current solution to find better ones, mutation is supposed to help for the exploration of the whole search space. Mutation is viewed as a background operator to maintain genetic diversity in the population. It introduces new genetic structures in the population by randomly modifying some of its building blocks. Mutation helps escape from local minima's trap and maintains diversity in the population. It also keeps the gene pool well stocked, thus ensuring ergodicity. A search space is said to be ergodic if there is a non-zero probability of generating any solution from any population state.

There are many different forms of mutation for the different kinds of representation. For binary representation, a simple mutation can consist in inverting the value of each gene with a small probability. The probability is usually taken about $1/L$, where L is the length of the chromosome. It is also possible to implement kind of hill climbing mutation operators that do mutation only if it improves the quality of the solution. Such an operator can accelerate the search; however, care should be taken, because it might also reduce the diversity in the population and make the algorithm converge toward some local optima. Mutation of a bit involves flipping a bit, changing 0 to 1 and vice-versa.

15.9.4.1 Flipping

Flipping of a bit involves changing 0 to 1 and 1 to 0 based on a mutation chromosome generated. Figure 15-30 explains mutation-flipping concept. A parent is considered and a mutation chromosome is randomly generated. For a 1 in mutation chromosome, the corresponding bit in parent chromosome is flipped (0 to 1 and 1 to 0) and child chromosome is produced. In the case illustrated in Figure 15-30, 1 occurs at 3 places of mutation chromosome, the corresponding bits in parent chromosome are flipped and the child is generated.

15.9.4.2 Interchanging

Two random positions of the string are chosen and the bits corresponding to those positions are interchanged (Figure 15.31).

Parent	1 0 1 1 0 1 0 1
Mutation chromosome	1 0 0 0 1 0 0 1
Child	0 0 1 1 1 1 0 0

Figure 15-30 Mutation flipping.

Parent	1 0 1 1 0 1 0 1
Child	1 1 1 1 0 0 0 1

Figure 15-31 Interchanging.

Parent	1 0 1 1 0 1 0 1
Child	1 0 1 1 0 1 1 0

Figure 15-32 Reversing.

15.9.4.3 Reversing

A random position is chosen and the bits next to that position are reversed and child chromosome is produced (Figure 15-32).

15.9.4.4 Mutation Probability

An important parameter in the mutation technique is the mutation probability (P_m). It decides how often parts of chromosome will be mutated. If there is no mutation, offspring are generated immediately after crossover (or directly copied) without any change. If mutation is performed, one or more parts of a chromosome are changed. If mutation probability is 100%, whole chromosome is changed; if it is 0%, nothing is changed. Mutation generally prevents the GA from falling into local extremes. Mutation should not occur very often, because then GA will in fact change to random search.

15.10 Stopping Condition for Genetic Algorithm Flow

In short, the various stopping condition are listed as follows:

1. *Maximum generations*: The GA stops when the specified number of generations has evolved.
2. *Elapsed time*: The genetic process will end when a specified time has elapsed.
   Note: If the maximum number of generation has been reached before the specified time has elapsed, the process will end.
3. *No change in fitness*: The genetic process will end if there is no change to the population's best fitness for a specified number of generations.
   Note: If the maximum number of generation has been reached before the specified number of generation with no changes has been reached, the process will end.

4. *Stall generations*: The algorithm stops if there is no improvement in the objective function for a sequence of consecutive generations of length "Stall generations."
5. *Stall time limit*: The algorithm stops if there is no improvement in the objective function during an interval of time in seconds equal to "Stall time limit."

The termination or convergence criterion finally brings the search to a halt. The following are the few methods of termination techniques.

15.10.1 Best Individual

A best individual convergence criterion stops the search once the minimum fitness in the population drops below the convergence value. This brings the search to a faster conclusion, guaranteeing at least one good solution.

15.10.2 Worst Individual

Worst individual terminates the search when the least fit individuals in the population have fitness less than the convergence criteria. This guarantees the entire population to be of minimum standard, although the best individual may not be significantly better than the worst. In this case, a stringent convergence value may never be met, in which case the search will terminate after the maximum has been exceeded.

15.10.3 Sum of Fitness

In this termination scheme, the search is considered to have satisfaction converged when the sum of the fitness in the entire population is less than or equal to the convergence value in the population record. This guarantees that virtually all individuals in the population will be within a particular fitness range, although it is better to pair this convergence criteria with weakest gene replacement, otherwise a few unfit individuals in the population will blow out the fitness sum. The population size has to be considered while setting the convergence value.

15.10.4 Median Fitness

Here at least half of the individuals will be better than or equal to the convergence value, which should give a good range of solutions to choose from.

15.11 Constraints in Genetic Algorithm

If the GA considered consists of only objective function and no information about the specifications of variable, then it is called *unconstrained optimization problem*. Consider, an unconstrained optimization problem of the form

$$\text{Minimize } f(x) = x^2 \tag{15.11}$$

and there is no information about "x" range. GA minimizes this function using its operators in random specifications.

In the case of constrained optimization problems, the information is provided for the variables under consideration. Constraints are classified as:

1. Equality relations.
2. Inequality relations.

A GA generates a sequence of parameters to be tested using the system under consideration, objective function (to be maximized or minimized) and the constraints. On running the system, the objective function is evaluated and constraints are checked to see if there are any violations. If there are no violations, the parameter set is assigned the fitness value corresponding to the objective function evaluation. When the constraints are violated, the solution is infeasible and thus has no fitness. Many practical problems are constrained and it is very difficult to find a feasible point that is best. As a result, one should get some information out of infeasible solutions, irrespective of their fitness ranking in relation to the degree of constraint violation. This is performed in penalty method.

Penalty method is one where a constrained optimization problem is transformed to an unconstrained optimization problem by associating a penalty or cost with all constraint violations. This penalty is included in the objective function evaluation.

Consider the original constrained problem in maximization form:

$$\text{Maximize } f(x)$$

$$\text{Subject to } g_i(x) \geq 0, \quad i = 1, 2, 3, \ldots, n \tag{15.12}$$

where x is a k-vector. Transforming this to unconstrained form:

$$\text{Maximize } f(x) + P\sum_{i=1}^{n} \Phi\,[g_i(x)] \tag{15.13}$$

where Φ is the penalty function and P is the penalty coefficient. There exist several alternatives for this penalty function. The penalty function can be squared for all violated constraints. In certain situations, the unconstrained solution converges to the constrained solution as the penalty coefficient p tends to infinity.

15.12 Problem Solving Using Genetic Algorithm

15.12.1 Maximizing a Function

Consider the problem of maximizing the function,

$$f(x) = x^2 \tag{15.14}$$

where x is permitted to vary between 0 and 31. The steps involved in solving this problem are as follows:

Step 1: For using GA approach, one must first code the decision variable "x" into a finite length string. Using a five bit (binary integer) unsigned integer, numbers between 0(00000) and 31(11111) can be obtained.

The objective function here is $f(x) = x^2$ which is to be maximized. A single generation of a GA is performed here with encoding, selection, crossover and mutation. To start with, select initial population at random. Here initial population of size 4 is chosen, but any number of populations can be selected based on the requirement and application. Table 15-4 shows an initial population randomly selected.

Table 15-4 Selection

String no.	Initial population (randomly selected)	x value	Fitness $f(x)= x^2$	Prob$_i$	Percentage probability (%)	Expected count	Actual count
1	0 1 1 0 0	12	144	0.1247	12.47	0.4987	1
2	1 1 0 0 1	25	625	0.5411	54.11	2.1645	2
3	0 0 1 0 1	5	25	0.0216	2.16	0.0866	0
4	1 0 0 1 1	19	361	0.3126	31.26	1.2502	1
Sum			1155	1.0000	100	4.0000	4
Average			288.75	0.2500	25	1.0000	1
Maximum			625	0.5411	54.11	2.1645	2

Step 2: Obtain the decoded x values for the initial population generated. Consider string 1,

$$\begin{aligned} 01100 &= 0 * 2^4 + 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 0 * 2^0 \\ &= 0 + 8 + 4 + 0 + 0 \\ &= 12 \end{aligned}$$

Thus for all the four strings the decoded values are obtained.

Step 3: Calculate the fitness or objective function. This is obtained by simply squaring the "x" value, since the given function is $f(x) = x^2$. When $x = 12$, the fitness value is

$$f(x) = x^2 = (12)^2 = 144$$

$$\text{For } x = 25, \quad f(x) = x^2 = (25)^2 = 625$$

and so on, until the entire population is computed.

Step 4: Compute the probability of selection,

$$\text{Prob}_i = \frac{f(x)_i}{\sum_{i=1}^{n} f(x)_i} \tag{15.15}$$

where n is the number of populations; $f(x)$ is the fitness value corresponding to a particular individual in the population;

$\Sigma f(x)$ is the summation of all the fitness value of the entire population.

Considering string 1,

$$\text{Fitness } f(x) = 144$$

$$\Sigma f(x) = 1155$$

The probability that string 1 occurs is given by

$$P_1 = 144/1155 = 0.1247$$

The percentage probability is obtained as

$$0.1247 * 100 = 12.47\%$$

The same operation is done for all the strings. It should be noted that summation of probability select is 1.

Step 5: The next step is to calculate the expected count, which is calculated as

$$\text{Expected count} = \frac{f(x)_i}{[\text{Avg}\, f(x)]_i} \tag{15.16}$$

where

$$(\text{Avg}\, f(x))_i = \left[\frac{\sum_{i=1}^{n} f(x)_i}{n}\right]$$

For string 1,

$$\text{Expected count} = \text{Fitness/Average} = 144/288.75 = 0.4987$$

We then compute the expected count for the entire population. The expected count gives an idea of which population can be selected for further processing in the mating pool.

Step 6: Now the actual count is to be obtained to select the individuals who would participate in the crossover cycle using Roulette wheel selection. The Roulette wheel is formed as shown Figure 15-33.

The entire Roulette wheel covers 100% and the probabilities of selection as calculated in step 4 for the entire populations are used as indicators to fit into the Roulette wheel. Now the wheel may be spun and the number of occurrences of population is noted to get actual count.

1. String 1 occupies 12.47%, so there is a chance for it to occur at least once. Hence its actual count may be 1.
2. With string 2 occupying 54.11% of the Roulette wheel, it has a fair chance of being selected twice. Thus its actual count can be considered as 2.
3. On the other hand, string 3 has the least probability percentage of 2.16%, so their occurrence for next cycle is very poor. As a result, it actual count is 0.

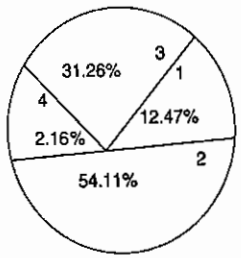


Figure 15-33 Selection using Roulette wheel.

Table 15-5 Crossover

String no.	Mating Pool	Crossover point	Offspring after crossover	x value	Fitness value $f(x)= x^2$
1	0 1 1 0 0	4	0 1 1 0 1	13	169
2	1 1 0 0 1	4	1 1 0 0 0	24	576
3	1 1 0 0 1	2	1 1 0 1 1	27	729
4	1 0 0 1 1	2	1 0 0 0 1	17	289
Sum					1763
Average					440.75
Maximum					729

4. String 4 with 31.26% has at least one chance for occurring while Roulette wheel is spun, thus its actual count is 1.

The above values of actual count are tabulated as shown is Table 15-5.

Step 7: Now, write the mating pool based upon the actual count as shown in Table 15-5.

The actual count of string no. 1 is 1, hence it occurs once in the mating pool. The actual count of string no. 2 is 2, hence it occurs twice in the mating pool. Since the actual count of string no. 3 is 0, it does not occur in the mating pool. Similarly, the actual count of string no. 4 being 1, it occurs once in the mating pool. Based on this, the mating pool is formed.

Step 8: Crossover operation is performed to produce new offspring (children). The crossover point is specified and based on the crossover point, single-point crossover is performed and new offspring is produced. The parents are

Parent 1	0 1 1 0 0
Parent 2	1 1 0 0 1

The offspring is produced as

Offspring 1	0 1 1 0 1
Offspring 2	1 1 0 0 0

In a similar manner, crossover is performed for the next strings.

Step 9: After crossover operations, new offspring are produced and "x" values are decoded and fitness is calculated.

Step 10: In this step, mutation operation is performed to produce new offspring after crossover operation. As discussed in Section 15.9.4.1 mutation-flipping operation is performed and new offspring are produced. Table 15-6 shows the new offspring after mutation. Once the offspring are obtained after mutation, they are decoded to x value and the fitness values are computed.

This completes one generation. The mutation is performed on a bit-bit by basis. The crossover probability and mutation probability were assumed to be 1.0 and 0.001, respectively. Once selection, crossover and

Table 15-6 Mutation

String no.	Offspring after crossover	Mutation chromosomes for flipping	Offspring after mutation	x value	Fitness $f(x)=x^2$
1	0 1 1 0 1	1 0 0 0 0	1 1 1 0 1	29	841
2	1 1 0 0 0	0 0 0 0 0	1 1 0 0 0	24	576
3	1 1 0 1 1	0 0 0 0 0	1 1 0 1 1	27	729
4	1 0 0 0 1	0 0 1 0 0	1 0 1 0 0	20	400
Sum					2546
Average					636.5
Maximum					841

mutation are performed, the new population is now ready to be tested. This is performed by decoding the new strings created by the simple GA after mutation and calculates the fitness function values from the x values thus decoded. The results for successive cycles of simulation are shown in Tables 15-4 and 15-6.

From the tables, it can be observed how GAs combine high-performance notions to achieve better performance. In the tables, it can be noted how maximal and average performances have improved in the new population. The population average fitness has improved from 288.75 to 636.5 in one generation. The maximum fitness has increased from 625 to 841 during the same period. Although random processes make this best solution, its improvement can also be seen successively. The best string of the initial population (1 1 0 0 1) receives two chances for its existence because of its high, above-average performance. When this combines at random with the next highest string (1 0 0 1 1) and is crossed at crossover point 2 (as shown in Table 15-5), one of the resulting strings (1 1 0 1 1) proves to be a very best solution indeed. Thus after mutation at random, a new offspring (1 1 1 0 1) is produced which is an excellent choice.

This example has shown one generation of a simple GA.

15.13 The Schema Theorem

In this section, we will formulate and prove the fundamental result on the behavior of GAs – the so-called Schema Theorem. Although being completely incomparable with convergence results for conventional optimization methods, it still provides valuable insight into the intrinsic principles of GAs. Assume a GA with proportional selection and an arbitrary but fixed fitness function f. Let us make the following notations:

1. The number of individuals which fulfill H at time step t are denoted as

$$r_{H,t} = |B_t \cap H|$$

2. The expression $\bar{f}(t)$ refers to the observed average fitness at time t:

$$\bar{f}(t) = \frac{1}{m}\sum_{i=1}^{m} f(b_{i,t})$$

3. The term $\bar{f}(H,t)$ stands for the observed average fitness of schema H in time step t:

$$\bar{f}(H,t) = \frac{1}{r_{H,t}} \sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})$$

Theorem (Schema Theorem – Holland 1975). Assuming we consider a simple GA, the following inequality holds for every schema H:

$$E[r_{H,t+1}] \geq r_{H,t} \frac{\bar{f}(H,t)}{\bar{f}(t)} \left(1 - p_C \frac{\delta(H)}{n-1}\right) (1-p_M)^{O(H)}$$

Proof. The probability that we select an individual fulfilling H is

$$\frac{\sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})}{\sum_{i=1}^{m} f(b_{i,t})}$$

This probability does not change throughout the execution of the selection loop. Moreover, each of the m individuals is selected independent of the others. Hence, the number of selected individuals, which fulfill H, is binomially distributed with sample amount m and the probability. We obtain, therefore, that the expected number of selected individuals fulfilling H is

$$m \frac{\sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})}{\sum_{i=1}^{m} f(b_{i,t})} = m \frac{r_{H,t}}{r_{H,t}} \frac{\sum_{i \in \{j | b_{j,t} \in H\}} f(b_{i,t})}{\sum_{i=1}^{m} f(b_{i,t})} = r_{H,t} \frac{\sum_{i \in \{j | b_{i,t} \in H\}} f(b_{i,t})/r_{H,t}}{\sum_{i=1}^{m} f(b_{i,t})/m} = r_{H,t} \frac{\bar{f}(H,t)}{\bar{f}(t)}$$

If two individuals are crossed, which both fulfill H, the two offsprings again fulfill H. The number of strings fulfilling H can only decrease if one string, which fulfills H, is crossed with a string which does not fulfill H, but, obviously, only if the cross site is chosen somewhere in between the specifications of H. The probability that the cross site is chosen within the defining length of H is

$$\frac{\delta(H)}{n-1}$$

Hence the survival probability p_S of H, i.e., the probability that a string fulfilling H produces an offspring also fulfilling H, can be estimated as follows (crossover is only done with probability p_C):

$$p_S \geq 1 - p_C \frac{\delta(H)}{n-1}$$

Selection and crossover are carried out independently, so we may compute the expected number of strings fulfilling H after crossover simply as

$$\frac{\bar{f}(H,t)}{\bar{f}(t)} r_{H,t}\, p_S \geq \frac{\bar{f}(H,t)}{\bar{f}(t)} r_{H,t} \left(1 - p_C \frac{\delta(H)}{n-1}\right)$$

After crossover, the number of strings fulfilling H can only decrease if a string fulfilling H is altered by mutation at a specification of H. The probability that all specifications of H remain untouched by mutation is obviously

$$(1-p_M)^{O(H)}$$

The arguments in the proof of the Schema Theorem can be applied analogously to many other crossover and mutation operations.

15.13.1 The Optimal Allocation of Trials

The Schema Theorem has provided the insight that building blocks receive exponentially increasing trials in future generations. The question remains, however, why this could be a good strategy. This leads to an important and well-analyzed problem from statistical decision theory – the two-armed bandit problem and its generalization, the k-armed bandit problem. Although this seems like a detour from our main concern, we shall soon understand the connection to GAs.

Suppose we have a gambling machine with two slots for coins and two arms. The gambler can deposit the coin either into the left or the right slot. After pulling the corresponding arm, either a reward is given or the coin is lost. For mathematical simplicity, we just work with outcomes, i.e., the difference between the reward (which can be zero) and the value of the coin. Let us assume that the lefr arm produces an outcome with mean value μ_1 and a variance σ_1^2 while the right arm produces an outcome with mean value μ_2 and variance σ_2^2. Without loss of generality, although the gambler does not know this, assume that $\mu_1 \geq \mu_2$.

Now the question arises which arm should be played. Since we do not know beforehand which arm is associated with the higher outcome, we are faced with an interesting dilemma. Not only must we make a sequence of decisions about which arm to play, we have to collect, at the same time, information about which is the better arm. This trade-off between exploration of knowledge and its exploitation is the key issue in this problem and, as turns out later, in GAs, too.

A simple approach to this problem is to separate exploration from exploitation. More specifically, we could perform a single experiment at the beginning and thereafter make an irreversible decision that depends on the results of the experiment. Suppose we have N coins. If we first allocate an equal number n (where $2n \leq N$) of trials to both arms, we could allocate the remaining $N - 2n$ trials to the observed better arm. Assuming we know all involved parameters, the expected loss is given as

$$L(N, n) = (\mu_1 - \mu_2)\{(N - n)q(n) + n[1 - q(n)]\}$$

where $q(n)$ is the probability that the worst arm is the observed best arm after $2n$ experimental trials. The underlying idea is obvious: In case that we observe that the worse arm is the best, which happens with probability $q(n)$, the total number of trials allocated to the right arm is $N - n$. The loss is, therefore, $(\mu_1 - \mu_2)(N - n)$. In the reverse case where we actually observe that the best arm is the best, which happens with probability $1 - q(n)$, the loss is only what we get less because we played the worse arm n times, i.e., $(\mu_1 - \mu_2)n$. Taking the central limit theorem into account, we can approximate $q(n)$ with the tail of a normal distribution:

$$q(n) \approx \frac{1}{\sqrt{2\pi}} \frac{e^{-c^2/2}}{c}$$

where

$$c = \frac{\mu_1 - \mu_2}{\sqrt{\sigma_1^2 + \sigma_2^2}} \sqrt{n}$$

Now we have to specify a reasonable experiment size n. Obviously, if we choose $n = 1$, the obtained information is potentially unreliable. If we choose, however, $n = N/2$ there are no trials left to make use of the information gained through the experimental phase. What we see is again the tradeoff between exploitation with almost no exploration ($n = 1$) and exploration without exploitation ($n = N/2$). It does not take a Nobel prize winner to see that the optimal way is somewhere in the middle. Holland has studied this problem in detail. He came to the conclusion that the optimal strategy is given by the following equation:

$$n^4 \approx b^2 \ln\left(\frac{N^2}{8\pi\, b^4 \ln N^2}\right)$$

where

$$b = \frac{\sigma_1}{\mu_1 - \mu_2}$$

Making a few transformations, we obtain that

$$N - n^4 \approx \sqrt{8\pi\, b^4 \ln N^2}\, e^{n^4/2b^2}$$

That is, the optimal strategy is to allocate slightly more than an exponentially increasing number of trials to the observed best arm. Although no gambler is able to apply this strategy in practice, because it requires knowledge of the mean values μ_1 and μ_2, we still have found an important bound of performance a decision strategy should try to approach.

A GA, although the direct connection is not yet fully clear, actually comes close to this ideal, giving at least an exponentially increasing number of trials to the observed best building blocks. However, one may still wonder how the two-armed bandit problem and GAs are related. Let us consider an arbitrary string position. Then there are two schemata of order one which have their only specification in this position. According to the Schema Theorem, the GA implicitly decides between these two schemata, where only incomplete data are available (observed average fitness values). In this sense, a GA solves a lot of two-armed problems in parallel.

The Schema Theorem, however, is not restricted to schemata of order one. Looking at competing schemata (different schemata which are specified in the same positions), we observe that a GA is solving an enormous number of k-armed bandit problems in parallel. The k-armed bandit problem, although much more complicated, is solved in an analogous way – the observed better alternatives should receive an exponentially increasing number of trials. This is exactly what a GA does!

15.13.2 Implicit Parallelism

So far we have discovered two distinct, seemingly conflicting views of genetic algorithms:

1. The algorithmic view that GAs operate on strings;
2. the schema-based interpretation.

So, we may ask what a GA really processes, strings or schemata? The answer is surprising: Both. Nowadays, the common interpretation is that a GA processes an enormous amount of schemata implicitly. This is accomplished by exploiting the currently available, incomplete information about these schemata continuously, while trying to explore more information about them and other, possibly better schemata.

This remarkable property is commonly called the implicit parallelism of GAs. A simple GA has only m structures in one time step, without any memory or bookkeeping about the previous generations. We will now try to get a feeling how many schemata a GA actually processes.

Obviously, there are 3^n schemata of length n. A single binary string fulfills n schema of order 1, $\binom{n}{2}$ schemata of order 2, in general, $\binom{n}{k}$ schemata of order k. Hence, a string fulfills

$$\sum_{k=1}^{n} \binom{n}{k} = 2^n$$

Theorem. Consider a randomly generated start population of a simple GA and let $\varepsilon \in (0, 1)$ be a fixed error bound. Then schemata of length

$$l_s < \varepsilon (n - 1) + 1$$

have a probability of at least $(1-\varepsilon)$ to survive one-point crossover (compare with the proof of the Schema Theorem). If the population size is chosen as $m = 2^l/2$, the number of schemata, which survive for the next generation, is of order $O(m^3)$.

15.14 Classification of Genetic Algorithm

There exist wide variety of GAs including simple and general GAs discussed in Sections 15.4 and 15.5, respectively. Some other variants of GA are discussed below.

15.14.1 Messy Genetic Algorithms

In a "classical" GA, the genes are encoded in a fixed order. The meaning of a single gene is determined by its position inside the string. We have seen in the previous chapter that a GA is likely to converge well if the optimization task can be divided into several short building blocks. What, however, happens if the coding is chosen such that couplings occur between distant genes? Of course, one-point crossover tends to disadvantage long schemata (even if they have low order) over short ones.

Messy GAs try to overcome this difficulty by using a variable-length, position-independent coding. The key idea is to append an index to each gene which allows identifying its position. A gene, therefore, is no longer represented as a single allele value and a fixed position, but as a pair of an index and an allele. Figure 15-34(A) shows how this "messy" coding works for a string of length 6.

Since with the help of the index we can identify the genes uniquely, genes may be swapped arbitrarily without changing the meaning of the string. With appropriate genetic operations, which also change the order of the pairs, the GA could possibly group coupled genes together automatically.

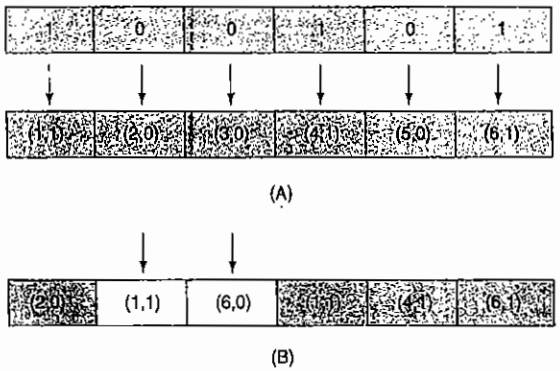


Figure 15-34 (A) Messy coding and (B) positional preference: Genes with indices 1 and 6 occur twice, the first occurrences are used.

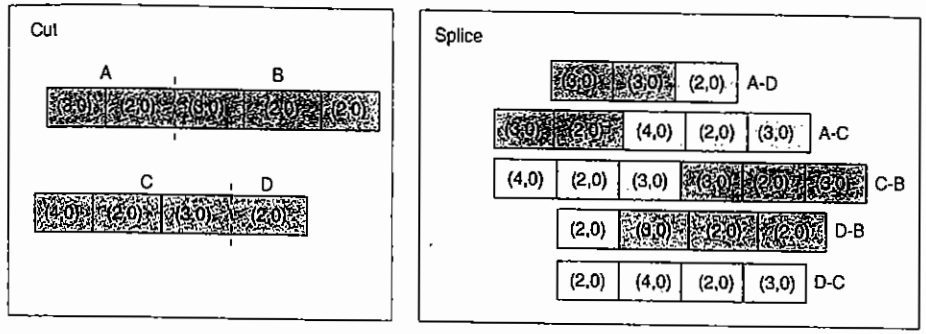


Figure 15-35 The cut and splice operation.

Owing to the free arrangement of genes and the variable length of the encoding, we can, however, run into problems, which do not occur, in a simple GA. First of all, it can happen that there are two entries in a string, which correspond to the same index but have conflicting alleles. The most obvious way to overcome this "over-specification" is positional preference – the first entry, which refers to a gene, is taken. Figure 15-34(B) shows an example. The reader may have observed that the genes with indices 3 and 5 do not occur at all in the example in Figure 15-34(B). This problem of "under specification" is more complicated and its solution is not as obvious as for over-specification. Of course, a lot of variants are reasonable.

One approach could be to check all possible combinations and to take the best one (for k missing genes, there are 2^k combinations). With the objective to reduce this effort, Goldberg et al. have suggested to use so-called competitive templates for finding specifications for k missing genes. It is nothing else than applying a local hill climbing method with random initial value to the k missing genes.

While messy GAs usually work with the same mutation operator as simple GAs (every allele is altered with a low probability pM), the crossover operator is replaced by a more general cut and splice operator which also allows to mate parents with different lengths. The basic idea is to choose cut sites for both parents independently and to splice the four fragments. Figure 15-35 shows an example.

15.14.2 Adaptive Genetic Algorithms

Adaptive GAs are those whose parameters, such as the population size, the crossing over probability, or the mutation probability, are varied while the GA is running. A simple variant could be the following: The mutation rate is changed according to changes in the population – the longer the population does not improve, the higher the mutation rate is chosen. Vice versa, it is decreased again as soon as an improvement of the population occurs.

15.14.2.1 Adaptive Probabilities of Crossover and Mutation

It is essential to have two characteristics in GAs for optimizing multimodal functions. The first characteristic is the capacity to converge to an optimum (local or global) after locating the region containing the optimum. The second characteristic is the capacity to explore new regions of the solution space in search of the global optimum. The balance between these characteristics of the GA is dictated by the values of p_m and p_c, and the type of crossover employed. Increasing values of p_m and p_c promote exploration at the expense of exploitation. Moderately large values of p_c (in the range 0.5–1.0) and small values of p_m (in the range 0.001–0.05) are commonly employed in GA practice. In this approach, we aim at achieving this tradeoff between exploration

and exploitation in a different manner, by varying p_c and p_m adaptively in response to the fitness values of the solutions; p_c and p_m are increased when the population tends to get stuck at a local optimum and are decreased when the population is scattered in the solution space.

15.14.2.2 Design of Adaptive p_c and p_m

To vary p_c and p_m adaptively for preventing premature convergence of the GA to a local optimum, it is essential to identify whether the GA is converging to an optimum. One possible way of detecting is to observe average fitness value $\bar{f}$ of the population in relation to the maximum fitness value f_{max} of the population. The value $f_{max} - \bar{f}$ is likely to be less for a population that has converged to an optimum solution than that for a population scattered in the solution space. We have observed the above property in all our experiments with GAs, and Figure 15-36 illustrates the property for a typical case. In Figure 15-36 we notice that $f_{max} - \bar{f}$ decreases when the GA converges to a local optimum with a fitness value of 0.5. (The globally optimal solution has a fitness value of 1.0.) We use the difference in the average and maximum fitness value, $f_{max} - \bar{f}$, as a yardstick for detecting the convergence of the GA. The values of p_c and p_m are varied depending on the value of $f_{max} - \bar{f}$. Since p_c and p_m have to be increased when the GA converges to a local optimum, i.e., when $f_{max} - \bar{f}$ decreases, p_c and p_m will have to be varied inversely with $f_{max} - \bar{f}$. The expressions that we have chosen for p_c and p_m are of the form

$$p_c = k_1/(f_{max} - \bar{f})$$
$$p_m = k_2/(f_{max} - \bar{f})$$

It has to be observed in the above expressions that p_c and p_m do not depend on the fitness value of any particular solution, and have the same values for all the solution of the population. Consequently, solutions with high fitness values as well as solutions with low fitness values are subjected to the same levels of mutation and crossover. When a population converges to a globally optimal solution (or even a locally optimal solution), p_c and p_m increase and may cause the disruption of the near-optimal solutions. The population may never

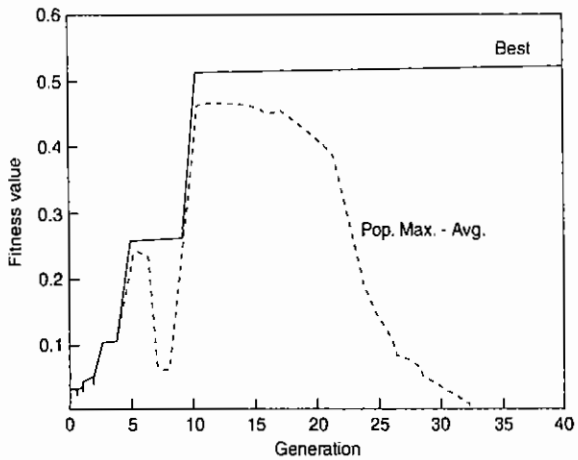


Figure 15-36 Variation of $f_{max} - \bar{f}$ and f_{best} (best fitness).

converge to the global optimum. Though we may prevent the GA from getting stuck at a local optimum, the performance of the GA (in terms of the generations required for convergence) will certainly deteriorate.

To overcome the above-stated problem, we need to preserve "good" solutions of the population. This can be achieved by having lower values of p_c and p_m for high fitness solutions and higher values of p_c and p_m for low fitness solutions. While the high fitness solutions aid in the convergence of the GA, the low fitness solutions prevent the GA from getting stuck at a local optimum. The value of p_m should depend not only on $f_{max} - \bar{f}$ but also on the fitness value f of the solution. Similarly, p_c should depend on the fitness values of both the parent solutions. The closer f is to f_{max} the smaller p_m should be, i.e., p_m should vary directly as $f_{max} - f$. Similarly, p_c should vary directly as $f_{max} - f'$, where f' is the larger of the fitness value of the solutions to be crossed. The expressions for p_c and p_m now take the forms

$$p_c = k_1[(f_{max} - f')/(f_{max} - \bar{f})], \; k_1 \leq 1.0$$
$$p_m = k_2[(f_{max} - f)/(f_{max} - \bar{f})], \; k_2 \leq 1.0$$

(Here k_1 and k_2 have to be less than 1.0 to constrain p_c and p_m to the range 0.0–1.0.)

Note that p_c and p_m are zero for the solution with the maximum fitness. Also $p_c = k_1$ for a solution with $f' = \bar{f}$, and $p_m = k_2$ for a solution with $f = \bar{f}$. For solution with subaverage fitness values, i.e., $f < \bar{f}$, p_c and p_m might assume values larger than 1.0. To prevent the overshooting of p_c and p_m beyond 1.0, we also have the following constraints:

$$p_c = k_3, \; f' \leq \bar{f}$$
$$p_m = k_4, \; f \leq \bar{f}$$

where $k_3, k_4 \leq 1.0$.

15.14.2.3 Practical Considerations and Choice of Values for k_1, k_2, k_3 and k_4

In the previous subsection, we saw that for a solution with the maximum fitness value p_c and p_m are both zero. The best solution in a population is transferred undisrupted into the next generation. Together with the selection mechanism, this may lead to an exponential growth of the solution in the population and may cause premature convergence. To overcome the above-stated problem, we introduce a default mutation rate (of 0.005) for every solution in the Adaptive Genetic Algorithm (AGA).

We now discuss the choice of values for k_1, k_2, k_3 and k_4. For convenience, the expressions for p_c and p_m are given as

$$p_c = k_1(f_{max} - f')/(f_{max} - \bar{f}), \; f \geq \bar{f}$$
$$p_c = k_3, \; f < \bar{f}$$
$$p_m = k_2(f_{max} - f')/(f_{max} - \bar{f}), \; f \geq \bar{f}$$
$$p_m = k_4, \; f < \bar{f}$$

where $k_1, k_2, k_3, k_4 \leq 1.0$.

It has been well established in GA literature that moderately large values of p_c ($0.5 < p_c < 1.0$) and small values of p_m ($0.001 < p_m < 0.05$) are essential for the successful working of GAs. The moderately large values of p_c promote the extensive recombination of schemata, while small values of p_m are necessary to prevent the disruption of the solutions. These guidelines, however, are useful and relevant when the values of p_c and p_m do not vary.

One of the goals of the approach is to prevent the GA from getting stuck at a local optimum. To achieve this goal, we employ solutions with sub-average fitnesses to search the search space for the region containing the global optimum. Such solutions need to be completely disrupted, and for this purpose we use a value of 0.5 for k_4. Since solutions with a fitness value of $\bar{f}$ should also be disrupted completely, we assign a value of 0.5 to k_2 as well.

Based on similar reasoning, we assign k_1 and k_3 a value of 1.0. This ensures that all solutions with a fitness value less than or equal to $\bar{f}$ compulsorily undergo crossover. The probability of crossover decreases as the fitness value (maximum of the fitness values of the parent solutions) tends to f_{max} and is 0.0 for solutions with a fitness value equal to f_{max}.

15.14.3 Hybrid Genetic Algorithms

As they use the fitness function only in the selection step, GAs are blind optimizers which do not use any auxiliary information such as derivatives or other specific knowledge about the special structure of the objective function. If there is such knowledge, however, it is unwise and inefficient not to make use of it. Several investigations have shown that a lot of synergism lies in the combination of genetic algorithms and conventional methods.

The basic idea is to divide the optimization task into two complementary parts. The GA does the coarse, global optimization while local refinement is done by the conventional method (e.g. gradient-based, hill climbing, greedy algorithm, simulated annealing, etc.). A number of variants are reasonable:

1. The GA performs coarse search first. After the GA is completed, local refinement is done.
2. The local method is integrated in the GA. For instance, every K generations, the population is doped with a locally optimal individual.
3. Both methods run in parallel: All individuals are continuously used as initial values for the local method. The locally optimized individuals are re-implanted into the current generation.

In this section a novel optimization approach is used that switches between global and local search methods based on the local topography of the design space. The global and local optimizers work in concert to efficiently locate quality design points better than either could alone. To determine when it is appropriate to execute a local search, some characteristics about the local area of the design space need to be determined. One good source of information is contained in the population of designs in the GA. By calculating the relative homogeneity of the population we can get a good idea of whether there are multiple local optima located within this local region of the design space.

To quantify the relative homogeneity of the population in each subspace, the coefficient of variance of the objective function and design variables is calculated. The coefficient of variance is a normalized measure of variation, and unlike the actual variance, is independent of the magnitude of the mean of the population. A high coefficient of variance could be an indication that there are multiple local optima present. Very low values could indicate that the GA has converged to a small area in the design space, warranting the use of a local search algorithm to find the best design within this region.

By calculating the coefficient of variance of the both the design variables and the objective function as the optimization progresses, it can also be used as a criterion to switch from the global to the local optimizer. As the variance of the objective values and design variables of the population increases, it may indicate that the optimizer is exploring new areas of the design space or hill climbing. If the variance is decreasing, the optimizer may be converging toward local minima and the optimization process could be made more efficient by switching to a local search algorithm.

The second method, regression analysis, used in this section helps us determine when to switch between the global and local optimizer. The design data present in the current population of the GA can be used to provide information as to the local topography of the design space by attempting to fit models of various order to it.

The use of regression analysis to augment optimization algorithms is not new. In problems in which the objective function or constraints are computationally expensive, approximations to the design space are created by sampling the design space and then using regression or other methods to create a simple mathematical model that closely approximates the actual design space, which may be highly nonlinear. The design space can then be explored to find regions of good designs or optimized to improve the performance of the system using the predictive surrogate approximation models instead of the computationally expensive analysis code, resulting in large computational savings. The most common regression models are linear and quadratic polynomials created by performing ordinary least squares regression on a set of analysis data.

To make clear the use of regression analysis in this way, consider Figure 15-37, which represents a complex design space. Our goal is to minimize this function, and as a first step the GA is run. Suppose that after a certain number of generations the population consists of the sampled points shown in the figure. Since the population of the GA is spread throughout the design space, having yet to converge into one of the local minima, it seems logical to continue the GA for additional generations. Ideally, before the local optimizer is run it would be beneficial to have some confidence that its starting point is somewhere within the mode that contains the optimum. Fitting a second-order response surface to the data and noting the large error (the R2 value is 0.13), there is a clear indication that the GA is currently exploring multiple modes in the design space.

In Figure 15-38, the same design space is shown but after the GA has begun to converge into the part of the design space containing the optimal design. Once again a second-order approximation is fit to GA's population. The dotted line connects the points predicted by the response surface. Note how much smaller the error is in the approximation (the R2 is 0.96), which is a good indication that the GA is currently exploring a single mode within the design space. At this point, the local optimizer can be made to quickly converge to the best solution within this area of the design space, thereby avoiding the slow convergence properties of the GA.

After each generation of the global optimizer, the values of the coefficient of determination and the coefficient of variance of the entire population are compared with the designer specified threshold levels.

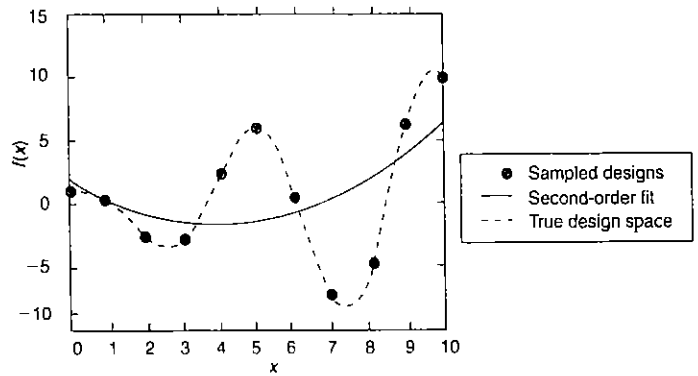


Figure 15-37 Approximating multiple modes with a second-order model.

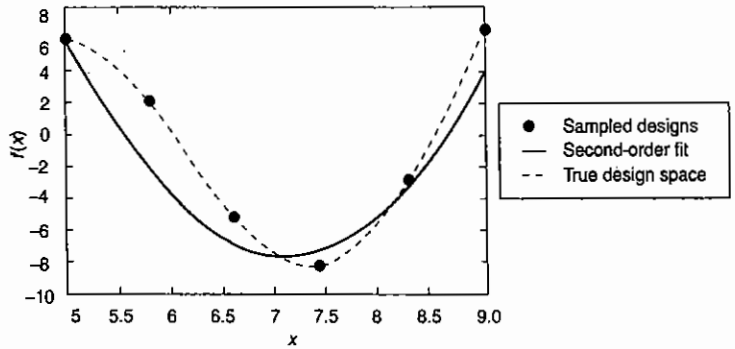


Figure 15-38 Approximating a single mode with a second-order model.

The first threshold simply states that if coefficient of determination of the population exceeds a designer set value when a second-order regression analysis is performed on the design data in the current GA population, then a local search is started from the current 'best design' in the population. The second threshold is based on the value of the coefficient of variance of the entire population. This threshold is also set by the designer and can range upwards from 0%. If it increases at a rate greater than the threshold level then a local search is executed from the best point in the population.

The flowchart in Figure 15-39 illustrates the stages in the algorithm. The algorithm can switch repeatedly between the global search (Stage 1) and the local search (Stage 2) during execution. In Stage 1, the global search is initialized and then monitored. This is also where the regression and statistical analysis occurs.

In Stage 2 the local search is executed when the threshold levels are exceeded, and then this solution is passed back and integrated into the global search. The algorithm stops when convergence is achieved for the global optimization algorithm.

15.14.4 Parallel Genetic Algorithm

GAs are powerful search techniques that are used successfully to solve problems in many different disciplines. Parallel GAs (PGAs) are particularly easy to implement and promise substantial gains in performance. As such, there has been extensive research in this field. The section describes some of the most significant problems in modeling and designing multi-population PGAs and presents some recent advancements.

One of the major aspects of GA is their ability to be parallelized. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process. Except in the selection phase, during which there is competition between individuals, the only intetactions between members of the population occur during the reproduction phase, and usually, no more than two individuals are necessary to engender a new child. Otherwise, any other operations of the evolution, in particular the evaluation of each member of the population, can be done separately. So, nearly all the operations in a genetic algorithm are implicitly parallel.

PGAs simply consist in distributing the task of a basic GA on different processors. As those tasks are implicitly parallel, little time will be spent on communication; and thus, the algorithm is expected to run much faster or to find more accurate results.

It has been established that GA's efficiency to find optimal solution is largely determined by the population size. With a larger population size, the genetic diversity increases, and so the algorithm is more likely to find

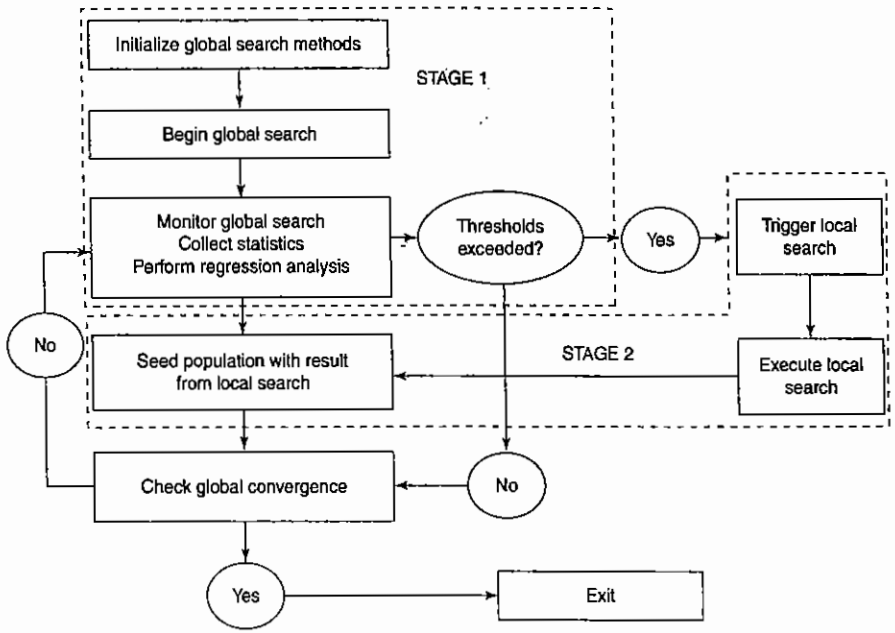


Figure 15-39 Steps in two-stage hybrid optimization approach.

a global optimum! A large population requires more memory to be stored; it has also been proved that it takes a longer time to converge. If n is the population size, the convergence is expected after $n \log(n)$ function evaluations.

The use of today's new parallel computers not only provides more storage space but also allows the use of several processors to produce and evaluate more solutions in a smaller amount of time. By parallelizing the algorithm, it is possible to increase population size, reduce the computational cost, and so improve the performance of the GA.

Probably the first attempt to map GAs to existing parallel computer architectures was made in 1981 by John Grefenstette. But obviously today, with the emergence of new high-performance computing (HPC), PGA is really a flourishing area. Researchers try to improve performance of GAs. The stake is to show that GAs are one of the best optimization methods to be used with HPC.

15.14.4.1 Global Parallelization

The first attempt to parallelize GAs simply consists of global parallelization. This approach tries to explicitly parallelize the implicit parallel tasks of the "sequential" GA. The nature of the problems remains unchanged. The algorithm still manipulates a single population where each individual can mate with any other, but the breeding of new children and/or their evaluation are now made in parallel. The basic idea is that different processors can create new individuals and compute their fitness in parallel almost without any communication among each other.

To start with, doing the evaluation of the population in parallel is something really simple to implement. Each processor is assigned a subset of individuals to be evaluated. For example, on a shared memory computer,

individuals could be stored in shared memory, so that each processor can read the chromosomes assigned and can write back the result of the fitness computation. This method only supposes that the GA works with a generational update of the population. Of course, some synchronization is needed between generations.

Generally, most of the computational time in a GA is spent calling the evaluation function. The time spent in manipulating the chromosomes during the selection or recombination phase is usually negligible. By assigning to each processor a subset of individuals to evaluate, a speed-up proportional to the number of processors can be expected if there is a good load balancing between them. However, load balancing should not be a problem as generally the time spent for the evolution of an individual does not really depend on the individual. A simple dynamic scheduling algorithm is usually enough to share the population between each processor equally.

On a distributed memory computer, we can store the population in one "master" processor responsible for sending the individuals to the other processors, i.e., "slaves." The master processor is also responsible for collecting the result of the evaluation. A drawback of this distributed memory implementation is that a bottleneck may occur when slaves are idle while only the master is working. But a simple and good use of the master processor can improve the load balancing by distributing individuals dynamically to the slave processors when they finish their jobs.

A further step could consist in applying the genetic operators in parallel. In fact, the interaction inside the population only occurs during selection. The breeding, involving only two individuals to generate the offspring, could easily be done simultaneously over $n/2$ pairs of individuals. But it is not that clear if it worth doing so. Crossover is usually very simple and not so time-consuming; the point is not that too much time will be lost during the communication, but that the time gain in the algorithm will be almost nothing compared to the effort produced to change the code.

This kind of global parallelization simply shows how easy it can be to transpose any GA onto a parallel machine and how a speed-up sublinear to the number of processors may be expected.

15.14.4.2 Classification of Parallel GAs

The basic idea behind most parallel programs is to divide a task into chunks and to solve the chunks simultaneously using multiple processors. This divide-and-conquer approach can be applied to GAs in many different ways, and the literature contains many examples of successful parallel implementations. Some parallelization methods use a single population, while others divide the population into several relatively isolated subpopulations. Some methods can exploit massively parallel computer architectures, while others are better suited to multicomputers with fewer and more powerful processing elements.

There are three main types of PGAs:

1. global single-population master-slave GAs,
2. single-population fine-grained,
3. multiple-population coarse-grained GAs.

In a master-slave GA there is a single panmictic population (just as in a simple GA), but the evaluation of fitness is distributed among several processors (see Figure 15-40). Since in this type of PGA, selection and crossover consider the entire population it is also known as global PGA. Fine-grained PGAs are suited for massively parallel computers and consist of one spatially structured population. Selection and mating are restricted to a small neighborhood, but neighborhoods overlap permitting some interaction among all the individuals (see Figure 15-41 for a schematic of this class of GAs). The ideal case is to have only one individual for every processing element available.

Multiple-population (or multiple-deme) GAs are more sophisticated, as they consist in several subpopulations which exchange individuals occasionally (Figure 15-42 has a schematic). This exchange of individuals

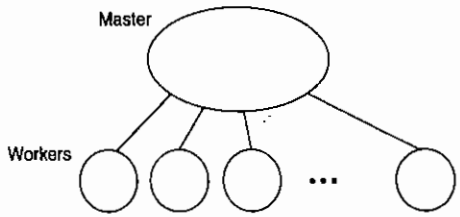


Figure 15-40 A schematic of a master-slave PGA. The master stores the population, executes GA operations and distributes individuals to the slaves. The slaves only evaluate the fitness of the individuals.

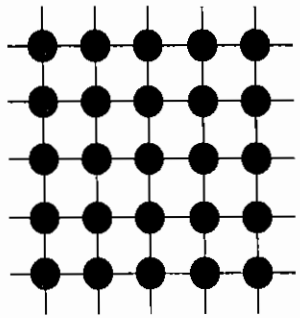


Figure 15-41 A schematic of a fine-grained PGA. This class of PGAs has one spatially distributed population, and it can be implemented very efficiently on massively parallel computers.

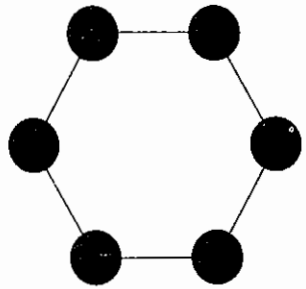


Figure 15-42 A schematic of a multiple-population PGA. Each process is a simple GA, and there is (infrequent) communication between the populations.

is called migration and, as we shall see in later sections, it is controlled by several parameters. Multiple-deme GAs are very popular, but also are the class of PGAs which is most difficult to understand, because the effects of migration are not fully understood. Multiple-deme PGAs introduce fundamental changes in the operation of the GA and have a different behavior than simple GAs.

Multiple-deme PGAs are known with different names. Sometimes they are known as "distributed" GAs, because they are usually implemented on distributed memory MIMD computers. Since the computation to communication ratio is usually high, they are occasionally called coarse-grained GAs. Finally, multiple-deme

GAs resemble the "island model" in Population Genetics which considers relatively isolated demes, so the PGAs are also known as "island" PGAs. Since the size of the demes is smaller than the population used by a serial GA, we would expect that the PGA converges faster. However, when we compare the performance of the serial and the parallel algorithms, we must also consider the quality of the solutions found in each case. Therefore, while it is true that smaller demes converge faster, it is also true that the quality of the solution might be poorer.

It is important to emphasize that while the master-slave parallelization method does not affect the behavior of the algorithm, the last two methods change the way the GA works. For example, in master-slave PGAs, selection takes into account all the population, but in the other two PGAs, selection only considers a subset of individuals. Also, in the master-slave any two individuals in the population can mate (i.e., there is random mating), but in the other methods mating is restricted to a subset of individuals.

The final method to parallelize GAs combines multiple demes with master-slave or fine-grained GAs. We call this class of algorithms *hierarchical PGAs*, because at a higher level they are multiple-deme algorithms with single–population PGAs (either master-slave or fine-grained) at the lower level. A hierarchical PGA combines the benefits of its components, and it promises better performance than any of them alone.

Master-slave parallelization: This section reviews the master-slave (or global) parallelization method. The algorithm uses a single population and the evaluation of the individuals and/or the application of genetic operators are done in parallel. As in the serial GA, each individual may compete and mate with any other (thus selection and mating are global). Global PGAs are usually implemented as master-slave programs, where the master stores the population and the slaves evaluate the fitness.

The most common operation that is parallelized is the evaluation of the individuals, because the fitness of an individual is independent from the rest of the population, and there is no need to communicate during this phase. The evaluation of individuals is parallelized by assigning a fraction of the population to each of the processors available. Communication occurs only as each slave receives its subset of individuals to evaluate and when the slaves return the fitness values. If the algorithm stops and waits to receive the fitness values for all the population before proceeding into the next generation, then the algorithm is synchronous. A synchronous master-slave GA has exactly the same properties as a simple GA, with speed being the only difference. However, it is also possible to implement an asynchronous master-slave GA where the algorithm does not stop to wait for any slow processors, but it does not work exactly like a simple GA. Most global PGA implementations are synchronous and the rest of the paper assumes that global PGAs carry out exactly the same search of simple GAs.

The global parallelization model does not assume anything about the underlying computer architecture, and it can be implemented efficiently on shared-memory and distributed-memory computers. On a shared-memory multiprocessor, the population could be stored in shared memory and each processor can read the individuals assigned to it and write the evaluation results back without any conflicts.

On a distributed-memory computer, the population can be stored in one processor. This "master" processor would be responsible for explicitly sending the individuals to the other processors (the "slaves") for evaluation, collecting the results and applying the genetic operators to produce the next generation. The number of individuals assigned to any processor may be constant, but in some cases (like in a multiuser environment where the utilization of processors is variable) it may be necessary to balance the computational load among the processors by using a dynamic scheduling algorithm (e.g., guided self-scheduling).

Multiple-deme parallel GAs: The important characteristics of multiple-deme PGAs are the use of a few relatively large subpopulations and migration. Multiple-deme GAs are the most popular parallel method, and many papers have been written describing innumerable aspects and details of their implementation.

Probably the first systematic study of PGAs with multiple populations was Grosso's dissertation. His objective was to simulate the interaction of several parallel subcomponents of an evolving population. Grosso simulated diploid individuals (so there were two subcomponents for each "gene"), and the population was divided into five demes. Each deme exchanged individuals with all the others with a fixed migration rate.

With controlled experiments, Grosso found that the improvement of the average population fitness was faster in the smaller demes than in a single large panmictic population. This confirms a long-held principle in Population Genetics: favorable traits spread faster when the demes are small than when the demes are large. However, he also observed that when the demes were isolated, the rapid rise in fitness stopped at a lower fitness value than with the large population. In other words, the quality of the solution found after convergence was worse in the isolated case than in the single population.

With a low migration rate, the demes still behaved independently and explored different regions of the search space. The migrants did not have a significant effect on the receiving deme and the quality of the solutions was similar to the case where the demes were isolated. However, at intermediate migration rates the divided population found solutions similar to those found in the panmictic population. These observations indicate that there is a critical migration rate below which the performance of the algorithm is obstructed by the isolation of the demes, and above which the partitioned population finds solutions of the same quality as the panmictic population.

It is interesting that such important observations were made so long ago, at the same time that other systematic studies of PGAs were underway. For example, Tanese proposed a PGA with the demes connected on a four-dimensional hypercube topology. In Tanese's algorithm, migration occurred at fixed intervals between processors along one dimension of the hypercube. The migrants were chosen probabilistically from the best individuals in the subpopulation, and they replaced the worst individuals in the receiving deme. Tanese carried out three sets of experiments. In the first, the interval between migrations was set to five generations, and the number of processors varied. In tests with two migration rates and varying the number of processors, the PGA found results of the same quality as the serial GA. However, it is difficult to see from the experimental results if the PGA found the solutions sooner than the serial GA, because the range of the times is too large. In the second set of experiments, Tanese varied the mutation and crossover rates in each deme, attempting to find parameter values to balance exploration and exploitation. The third set of experiments studied the effect of the exchange frequency on the search, and the results showed that migrating too frequently or too infrequently degraded the performance of the algorithm.

The multideme PGAs are popular due to the following several reasons:

1. Multiple-deme GAs seem like a simple extension of the serial GA. The recipe is simple: take a few conventional (serial) GAs, run each of them on a node of a parallel computer, and at some predetermined times exchange a few individuals.
2. There is relatively little extra effort needed to convert a serial GA into a multiple-deme GA. Most of the program of the serial GA remains the same and only a few subroutines need to be added to implement migration.
3. Coarse-grain parallel computers are easily available, and even when they are not, it is easy to simulate one with a network of workstations or even on a single processor using free software (like MPI or PVM).

There are a few important issues noted from the above sections. For example, PGAs are very promising in terms of the gains in performance. Also, PGAs are more complex than their serial counterparts. In particular, the migration of individuals from one deme to another is controlled by several parameters like (a) the topology that defines the connections between the subpopulations, (b) a migration rate that controls how many individuals migrate and (c) a migration interval that affects the frequency of migrations. In the late 1980s and early 1990s,

the research on PGAs began to explore alternatives to make PGAs faster and to understand better how they worked.

Around this time the first theoretical studies on PGAs began to appear and the empirical research attempted to identify favorable parameters. This section reviews some of that early theoretical work and experimental studies on migration and topologies. Also in this period, more researchers began to use multiple-population GAs to solve application problems, and this section ends with a brief review of their work.

One of the directions in which the field matured is that PGAs began to be tested with very large and difficult test functions.

Fine-grained PGAs: The development of massively parallel computers triggers a new approach of PGAs. To take advantage of new architectures with even a greater number of processors and less communication costs, fine-grained PGAs have been developed. The population is now partitioned into a large number of very small subpopulations. The limit (and may be ideal) case is to have just one individual for every processing element available.

Basically, the population is mapped onto a connected processor graph, usually, one individual on each processor. (But it works also more than one individual on each processor. In this case, it is preferable to choose a multiple of the number of processors for the population size.) Mating is only possible between neighboring individual, i.e, individuals stored on neighboring processors. The selection is also done in a neighborhood of each individual and so depends only on local information. A motivation behind local selection is biological. In nature there is no global selection, instead natural selection is a local phenomenon, taking place in an individual's local environment.

If we want to compare this model to the island model, each neighborhood can be considered as a different deme. But here, the demes overlap providing a way to disseminate good solutions across the entire population. Thus, the topology does not need to explicitly define migration roads and migration rate.

It is common to place the population on a two-dimensional or three-dimensional torus grid because in many massively parallel computers the processing elements are connected using this topology. Consequently each individual has four neighbors. Experimentally, it seems that good results can be obtained using a topology with a medium diameter and neighborhoods not too large. Like the coarse-grained models, it worth trying to simulate this model even on a single processor to improve the results. Indeed, when the population is stored in a grid like this, after few generations, different optima could appear in different places on the grid.

To sum up, with parallelization of GA, all the different models proposed and all the new models we can imagine by mixing those ones, can demonstrate how well GA are adapted to parallel computation. In fact, the too many implementations reported in the literature may even be confusing. We really need to understand what truly affects the performance of PGAs.

Fine-grained PGAs have only one population, but have a spatial structure that limits the interactions between individuals. An individual can only compete and mate with its neighbors; but since the neighborhoods overlap good solutions may disseminate across the entire population.

Robertson parallelized the GA of a classifier system on a Connection Machine 1. He parallelized the selection of parents, the selection of classifiers to replace, mating, and crossover. The execution time of his implementation was independent of the number of classifiers (up to 16K, the number of processing elements in the CM-1).

Hierarchical parallel algorithms: A few researchers have tried to combine two of the methods to parallelize GAs, producing hierarchical PGAs. Some of these new hybrid algorithms add a new degree of complexity to the already complicated scene of PGAs, but other hybrids manage to keep the same complexity as one of their components. When two methods of parallelizing GAs are combined they form a hierarchy. At the upper level most of the hybrid PGAs are multiple-population algorithms.

Some hybrids have a fine-grained GA at the lower level (see Figure 15-43). For example Gruau invented a "mixed" PGA. In his algorithm, the population of each deme was placed on a two-dimensional grid, and the demes themselves were connected as a two-dimensional torus. Migration between demes occurred at regular intervals, and good results were reported for a novel neural network design and training application.

Another type of hierarchical PGA uses a master-slave on each of the demes of a multi-population GA (see Figure 15-44). Migration occurs between demes, and the evaluation of the individuals is handled in parallel. This approach does not introduce new analytic problems, and it can be useful when working with complex applications with objective functions that need a considerable amount of computation time. Bianchini and

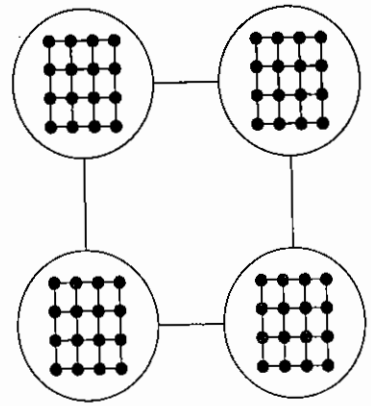


Figure 15-43 Hierarchical GA combines a multiple-deme GA (at the upper level) and a fine-grained GA (at the lower level).

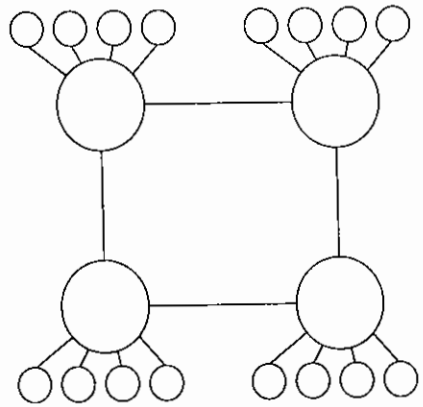


Figure 15-44 A schematic of a hierarchical PGA. At the upper level this hybrid is a multi-deme PGA where each node is a master-slave GA.

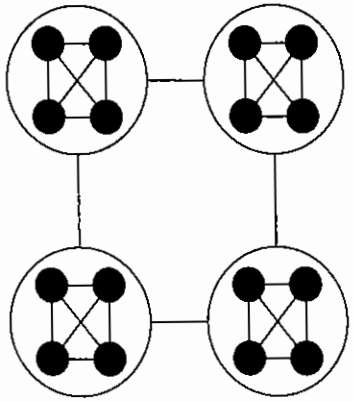


Figure 15-45 This hybrid uses multiple-deme GAs at both the upper and the lower levels. At the lower level the migration rate is faster and the communications topology is much denser than at the upper level.

Brown presented an example of this method of hybridizing PGAs, and showed that it can find a solution of the same quality as of a master-slave PGA or a multiple-deme GA in less time.

Interestingly, a very similar concept was invented by Goldberg in the context of an object-oriented implementation of a "community model" PGA. In each "community" there are multiple houses where parents reproduce and the offsprings are evaluated. Also, there are multiple communities and it is possible that individuals migrate to other places.

A third method of hybridizing PGAs is to use multiple-deme GAs at both the upper and the lower levels (see Figure 15-45). The idea is to force panmictic mixing at the lower level by using a high migration rate and a dense topology, while a low migration rate is used at the high level. The complexity of this hybrid would be equivalent to a multiple-population GA if we consider the groups of panmictic subpopulations as a single deme. This method has not been implemented yet. Hierarchical implementations can reduce the execution time more than any of their components alone.

15.14.4.3 Coarse-Grained PGAs – The Island Model

The second class of PGA is once again inspired by nature. The population is now divided into a few subpopulations or demes, and each of these relatively large demes evolves separately on different processors. Exchange between subpopulations is possible via a migration operator. The term *island model* is easily understandable; the GA behave as if the world was constituted of islands where populations evolve isolated from each other. On each island the population is free to converge toward different optima. The migration operator allows "metissage" of the different subpopulations and is supposed to mix good features that emerge locally in the different demes.

We can notice that this time the nature of the algorithm changes. An individual can no longer breed with any other from the entire population, but only with individuals of the same island. Amazingly, even if this algorithm has been developed to be used on several processors, it is worth simulating it sequentially on one processor. It has been shown on a few problems that better results can be achieved using this model. This algorithm is able to give different sub-optimal solutions, and in many problems, it is an advantage if we need to determine a kind of landscape in the search space to know where the good solutions are located. Another great advantage of the island model is that the population in each island can evolve with different

rules. That can be used for multicriterion optimization. On each island, selection can be made according to different fitness functions, representing different criterions. For example it can be useful to have as many islands as criteria, plus another central island where selection is done with a multicriterion fitness function. The migration operator allows individuals to move between islands, and therefore, to mix criteria.

In literature this model is sometimes also referred as the coarse-grained PGA. (In parallelism, grain size refers to the ratio of time spent in computation and time spent in communication; when the ratio is high the processing is called coarse-grained). Sometimes, we can also find the term "distributed" GA, since they are usually implemented on distributed memory machines (MIMD Computers).

Technically there are three important features in the coarse-grained PGA: the topology that defines connections between subpopulations, migration rate that controls how many individuals migrate, migtation intervals that affect how often the migration occurs. Even if a lot of work has been done to find optimal topology and migration parameters, here, intuition is still used more often than analysis with quite good results.

Many topologies can be defined to connect the demes, but the most common models are the island model and the stepping-stones model. In the basic island model, migration can occur between any subpopulations, whereas in the stepping stone demes are disposed on a ring and migration is restricted to neighbouring demes. Works have shown that the topology of the space is not so important as long as it has high connectivity and small diameter to ensure adequate mixing as time proceeds.

Choosing the right time for migration and which individuals should migrate appears to be more complicated. Quite a lot of work is done on this subject, and problems come from the following dilemmas. We can observe that species are converging quickly in small isolated populations. Nevertheless, migrations should occur after a time long enough for allowing the development of goods characteristics in each subpopulation. It also appears that, immigration is a trigger for evolutionary changes. If migration occurs after each new genetation, the algorithm is more or less equivalent to a sequential GA with a larger population. In practice, migration occurs either after a fixed number of iterations in each deme or at uniform periods of time. Migrants are usually selected randomly from the best individuals in the population and they replace the worst in the receiving deme. In fact, intuition is still mainly used to fix migration rate and migration intervals; there is absolutely nothing rigid, each personal cooking recipe may give good results.

15.14.5 Independent Sampling Genetic Algorithm (ISGA)

In the independent sampling phase, we design a core scheme, named the "Building Block Detecting Strategy" (BBDS), to extract relevant building block information of a fitness landscape. In this way, an individual is able to sequentially construct more highly fit partial solutions. For Royal Road $R1$, the global optimum can be attained easily. For other more complicated fitness landscapes, we allow a number of individuals to adopt the BBDS and independently evolve in parallel so that each schema region can be given samples independently. During this phase, the population is expected to be seeded with promising genetic material. Then follows the breeding phase, in which individuals are paired for breeding based on two mate-selection schemes (Huang, 2001): individuals being assigned mates by natural selection only and individuals being allowed to actively choose their mates. In the latter case, individuals are able to distinguish candidate mates that have the same fitness yet have different string structures, which may lead to quite different performance after crossover. This is not achievable by natural selection alone since it assigns individuals of the same fitness the same probability for being mates, without explicitly taking into account string structures. In short, in the breeding phase individuals manage to construct even more promising schemata through the recombination of highly fit building blocks found in the first phase. Owing to the characteristic of independent sampling of building blocks that distinguishes the proposed GAs from conventional GAs, we name this type of GA *independent sampling genetic algorithms* (ISGAs).

15.14.5.1 Comparison of ISGA with PGA

The independent sampling phase of ISGAs is similar to the fine-grained PGAs in the sense that each individual evolves autonomously, although ISGAs do not adopt the population structure. An initial population is randomly generated. Then in every cycle each individual does local hill climbing, and creates the next population by mating with a partner in its neighborhood and replacing parents if offsprings are better. By contrast, ISGAs partition the genetic processing into two phases: the independent sampling phase and the breeding phase as described in the preceding section. Third, the approach employed by each individual for improvement in ISGAs is different from that of the PGAs. During the independent sampling phase of ISGAs, in each cycle, through the BBDS, each individual attempts to extract relevant information of potential building blocks whenever its fitness increases. Then, based on the schema information accumulated, individuals continue to construct more complicated building blocks. However, the individuals of fine-grained PGAs adopt a local hill climbing algorithm that does not manage to extract relevant information of potential schemata.

The motivation of the two phased ISGAs was partially from the messy genetic algorithms (mGAs). The two stages employed in the mGAs are "primordial phase" and "juxtapositional phase," in which the mGAs first emphasize candidate building blocks based on the guess at the order k of small schemata, then juxtaposing them to build up global optima in the second phase by "cut" and "splice" operators. However, in the first phase, the mGAs still adopt centralized selection to emphasize some candidate schemata; this in turn results in the loss of samples of other potentially promising schemata. By contrast, ISGAs manage to postpone the emphasis of candidate building blocks to the latter stage, and highlight the feature of independent sampling of building blocks to suppress hitchhiking in the first phase. As a result, population is more diverse and implicit parallelism can be fulfilled to a larger degree. Thereafter, during the second phase, ISGAs implement population breeding through two mate-selection schemes as discussed in the preceding section. In the following subsections, we present the key components of ISGAs in detail and show the comparisons between the experimental results of the ISGAs and those of several other GAs on two benchmark test functions.

15.14.5.2 Components of ISGAs

ISGAs are divided into two phases: the independent sampling phase and the breeding phase. We describe them as follows.

Independent sampling phase: To implement independent sampling of various building blocks, a number of strings are allowed to evolve in parallel and each individual searches for a possible evolutionary path entirely independent of others.

In this section, we develop a new searching strategy, BBDS, for each individual to evolve based on the accumulated knowledge for potentially useful building blocks. The idea is to allow each individual to probe valuable information concerning beneficial schemata through testing its fitness increase since each time a fitness increase of a string could come from the presence of useful building blocks on it. In short, by systematically testing each bit to examine whether this bit is associated with the fitness increase during each cycle, a cluster of bits constituting potentially beneficial schemata will be uncovered. Iterating this process guarantees the formation of longer and longer candidate building blocks.

The operation of BBDS on a string can be described as follows:

1. Genetate an empry set for collecting genes of candidate schemata and create an initial string with uniform probability for each bit until its fitness exceeds 0. (Record the current fitness as *Fit*.)
2. Except the genes of candidate schemata collected, from left to right, successively all the other bits, one at a time, evaluate the resulting string. If the resulting fitness is less than *Fit*, record this bit's position and original value as a gene of candidate schemata.

3. Except the genes recorded. Randomly generate all the other bits of the string until the resulting string's fitness exceeds *Fit*. Replace *Fit* by the new fitness.
4. Go to steps 2 and 3 until some end criterion. The idea of this strategy is that the cooperation of certain genes (bits) makes for good fitness.

Once these genes come in sight simultaneously, they contribute a fitness increase to the string containing them; thus any loss of one of these genes leads to the fitness decrease of the string. This is essentially what step 2 does and after this step we should be able to collect a set of genes of candidate schemata. Then at step 3, we keep the collected genes of candidate schemata fixed and randomly generate other bits, awaiting other building blocks to appear and bring forth another fitness in crease.

However, step 2 in this strategy only emphasizes the fitness drop due to a particular bit. It ignores the possibility that the same bit leads to a new fitness rise because many loci could interact in an extremely nonlinear fashion. To take this into account, the second version of BBDS is introduced through the change in step 2 as follows.

Step 2: Except the genes of candidate schemata collected, ftom left to right, successively all the other bits, one at a time, evaluate the resulting string. If the resulting fitness is less than *Fit*, record this bit's position and original value as a gene of candidate schemata. If the resulting fitness exceeds *Fit*, substitute this bit's 'new' value for the old value, replace *Fit* by this new fitness, record this bit's position and 'new' value as a gene of candidate schemata, and re-execute this step.

Because this version of BBDS takes into consideration the fitness increase resulted from that particular bit, it is expected to take less time for detecting. Other versions of BBDS are of course possible. For example, in step 2, if the same bit results in a fitness increase, it can be recorded as a gene of candidate schemata, and the procedure continues to test the residual bits yet without completely traveling back to the first bit to reexamine each bit. However, the empirical results obtained thus far indicate that the performance of this alternative is quite similar to that of the second version. More experimental results are needed to distinguish the difference between them.

The overall implementation of the independent sampling phase of ISGAs is through the proposed BBDS to get autonomous evolution of each string until all individuals in the population have reached some end criterion.

Breeding phase: After the independent sampling phase, individuals independently build up their own evolutionary avenues by various building blocks. Hence the population is expected to contain diverse beneficial schemata and premature convergence is alleviated to some degree. However, factors such as deception and incompatible schemata (i.e., two schemata have different bit values at common defining positions) still could lead individuals to arrive at suboptimal regions of a fitness landscape. Since building blocks for some strings to leave suboptimal regions may be embedded in other strings, the search for proper mating partners and then exploiting the building blocks on them are critical for overwhelming the difficulty of strings being trapped in undesired regions. In Huang (2001) the importance of mate selection has been investigated and the results showed that the GAs is able to improve their performance when the individuals are allowed to select mates to a larger degree.

In this section, we adopt two mate-selection schemes analyzed in Huang (2001) to breed the population: individuals being assigned mates by natural selection only and individuals being allowed to actively choose their mates. Since natural selection assigns strings of the same fitness the same probability for being parents, individuals of identical fitness yet distinct string structures are treated equally. This may result in significant loss of performance improvement after crossover.

We adopt the tournament selection scheme (Mitchell, 1996) as the role of natural selection and the mechanism for choosing mates in the breeding phase is as follows:

During each mating event, a binary tournament selection with probability 1.0 is performed to select the first individual out of the two fittest randomly sampled individuals according to the following schemes:

1. Run the binary tournament selection again to choose the partner.
2. Run another two times of the binary tournament selection to choose two highly fit candidate partners; then the one more dissimilar to the first individual is selected for mating.

The implementation of the breeding phase is through iterating each breeding cycle which consists of (a) two parents obtained on the basis of the mate-selection schemes above. (b) Two-point crossover operator (crossover rate 1.0) is applied to these parents. (c) Both parents are replaced with both offsprings if any of the two offsprings is better than them. Then steps (a), (b) and (c) are repeated until the population size is reached and this is a breeding cycle.

15.14.6 Real-Coded Genetic Algorithms

The variant of GAs for real-valued optimization that is closest to the original GA are so-called real-coded GAs. Let us assume that we are dealing with a free N-dimensional real-valued optimization problem, which means $X = R^N$ without constraints. In a real-coded GA, an individual is then represented as an N-dimensional vector of real numbers:

$$b = (x_1, \ldots, x_N)$$

As selection does not involve the particular coding, no adaptation needs to be made – all selection schemes discussed so far are applicable without any restriction. What has to be adapted to this special structure are the genetic operations crossover and mutation.

15.14.6.1 Crossover Operators for Real-Coded GAs

So far, the following crossover schemes are most common for real-coded GAs:

Flat crossover: Given two parents $b^1 = (x_1^1, \ldots, x_N^1)$ and $b^2 = (x_1^2, \ldots, x_N^2)$, a vector of random values from the unit interval $(\lambda_1, \ldots, \lambda_N)$ is chosen and the offspring $b = (x_1', \ldots, x_N')$ is computed as a vector of linear combinations in the following way (for all $i = 1, \ldots, N$):

$$x_i' = \lambda_i \cdot x_i^1 + (1 - \lambda_i) \cdot x_i^2$$

BLX-α crossover is an extension of flat crossover, which allows an offspring allele to be also located outside the interval

$$[\min(x_i^1, x_i^2), \max(x_i^1, x_i^2)]$$

In BLX-α crossover, each offspring allele is chosen as a uniformly distributed random value from the interval

$$[\min(x_i^1, x_i^2) - I \cdot \alpha\,, \max(x_i^1, x_i^2) + I \cdot \alpha\,]$$

where $I = \max(x_i^1, x_i^2) - \min(x_i^1, x_i^2)$. The parameter α has to be chosen in advance. For $\alpha = 0$, BLX-α crossover becomes identical to flat crossover.

Simple crossover is nothing else but classical one-point crossover for real vectors, i.e., a crossover site $k \in 2\{1, \ldots, N-1\}$ is chosen and two offspring are created in the following way:

$$b^1 = (x_1^1, \ldots, x_k^1, x_{k+1}^2, \ldots, x_N^2)$$
$$b^N = (x_1^2, \ldots, x_k^2, x_{k+1}^1, \ldots, x_N^1)$$

Discrete crossover is analogous to classical uniform crossover for real vectors. An offspring b of the two parents b^1 and b^2 is composed from alleles, which are randomly chosen either as x_i^1 or x_i^2.

15.14.6.2 Mutation Operators for Real-Coded GAs

The following mutation operators are most common for real-coded GAs:

1. *Random mutation*: For a randomly chosen gene i of an individual $b = (x_1, \ldots, x_N)$, the allele x_i is replaced by a randomly chosen value from a predefined interval $|a_i, b_i|$.
2. *Nonuniform mutation*: In nonuniform mutation, the possible impact of mutation decreases with the number of generations. Assume that t_{max} is the predefined maximum number of generations. Then, with the same setup as in random mutation, the allele x_i is replaced by one of the two values

$$x_i' = x_i + \Lambda\,(t, b_i - x_i)$$
$$x_i'' = x_i - \Lambda\,(t, x_i - a_i)$$

The choice as to which of the two is taken is determined by a random experiment with two outcomes that have equal probabilities 1/2 and 1/2. The random variable $\Lambda\,(t, x)$ determines a mutation step from the range $|0, x|$ in the following way:

$$\Delta\,(t, x) = x(1 - \lambda^{[1-(t/t_{max})]^r})$$

In this formula, λ is a uniformly distributed random value from the unit interval. The parameter r determines the influence of the generation index t on the distribution of mutation step sizes over the interval $|0, x|$.

15.15 Holland Classifier Systems

A Holland classifier system is a classifier system of the Michigan type which processes binary messages of a fixed length through a rule base whose rules are adapted according to response of the environment.

15.15.1 The Production System

First of all, the communication of the production system with the environment is done via an arbitrarily long list of messages. The detectors translate responses from the environment into binary messages and place them on the message list which is then scanned and changed by the rule base. Finally, the effectors translate output messages into actions on the environment, such as forces or movements.

Messages are binary strings of the same length k. More formally, a message belongs to $\{0, 1\}^k$. The rule base consists of a fixed number (m) of rules (classifiers) which consist of a fixed number (r) of conditions and an action, where both conditions and actions are strings of length k over the alphabet {0, 1, *}. The asterisk plays the role of a wildcard, a 'don't care' symbol.

A condition is matched if and only if there is a message in the list which matches the condition in all nonwildcard positions. Moreover, conditions, except the first one, may be negated by adding a '–' prefix. Such a prefixed condition is satisfied if and only if there is no message in the list which matches the string associated with the condition. Finally, a rule fires if and only if all the conditions are satisfied, i.e., the conditions are connected with AND. Such 'firing' rules compete to put their action messages on the message list.

In the action parts, the wildcard symbols have a different meaning. They take the role of 'pass through' element. The output message of a firing rule, whose action part contains a wildcard, is composed from the

nonwildcard positions of the action and the message which satisfies the first condition of the classifier. This is actually the reason why negations of the first conditions are not allowed. More formally, the outgoing message $\underline{m}$ is defined as

$$\underline{m} = \begin{cases} a[i] & \text{if } a[i] \neq * \\ m[i] & \text{if } a[i] = * \end{cases} \quad i = 1, \ldots, k$$

where a is the action part of the classifier and m is the message which matches the first condition. Formally, a classifier is a string of the form

$$\text{Cond}_1, |\text{'}-\text{'}||\text{Cond}_2, \ldots, |\text{'}-\text{'}\,\text{Cond}_r/\text{Action}$$

where the brackets should express the optionality of the "–" prefixes. Depending on the concrete needs of the task to be solved, it may be desirable to allow messages to be preserved for the next step. More specifically, if a message is not interpreted and removed by the effectors interface, it can make another classifier fire in the next step. In practical applications, this is usually accomplished by reserving a few bits of the messages for identifying the origin of the messages (a kind of variable index called tag).

Tagging offers new opportunities to transfer information about the current step into the next step simply by placing tagged messages on the list, which are not interpreted, by the output interface. These messages, which obviously contain information about the previous step, can support the decisions in the next step. Hence, appropriate use of tags permits rules to be coupled to act sequentially. In some sense, such messages are the memory of the system.

A single execution cycle of the production system consists of the following steps:

1. Messages from the environment are appended to the message list.
2. All the conditions of all classifiers are checked against the message list to obtain the set of firing rules.
3. The message list is erased.
4. The firing classifiers participate in a competition to place their messages on the list.
5. The winning classifiers place their actions on the list.
6. The messages directed to the effectors are executed.

This procedure is repeated iteratively. How step 6 is done, if these messages are deleted or not, and so on, depends on the concrete implementation. It is, on the one hand, possible to choose a representation such that the effectors can interpret each output message. On the other hand, it is possible to direct messages explicitly to the effectors with a special tag. If no messages are directed to the effectors, the system is in a thinking phase.

A classifier R1 is called consumer of a classifier R2 if and only if there is a message m0 which fulfills at least one of R1's conditions and has been placed on the list by R2. Conversely, R2 is called a supplier of R1.

15.15.2 The Bucket Brigade Algorithm

As already mentioned, in each time step t, we assign a strength value $u_{i,t}$ to each classifier Ri. This strength value represents the correctness and importance of a classifier. On the one hand, the strength value influences the chance of a classifier to place its action on the output list. On the other hand, the strength values are used by the rule discovery system, which we will soon discuss.

In Holland classifier systems, the adaptation of the strength values depending on the feedback (payoff) from the environment is done by the so-called bucket brigade algorithm. It can be regarded as a simulated economic system in which various agents, here the classifiers, participate in an auction, where the chance to buy the right to post the action depends on the strength of the agents.

The bid of classifier Ri at time t is defined as

$$B_{i,t} = c_L u_{i,t} s_i$$

where $c_L \in [0, 1]$ is a learning parameter, similar to learning rates in artificial neural nets, and s_i is the specificity, the number of nonwildcard symbols in the condition part of the classifier. If c_L is chosen small, the system adapts slowly. If it is chosen too high, the strengths tend to oscillate chaotically. Then the rules have to compete for the right for placing their output messages on the list. In the simplest case, this can be done by a random experiment like the selection in a genetic algorithm. For each bidding classifier it is decided randomly if it wins or not, where the probability that it wins is proportional to its bid:

$$P[R_i \text{ wins}] = \frac{B_{i,t}}{\sum_{j \in \text{Sat}_t} B_{j,t}}$$

In this equation, Sat_t is the set of indices of all classifiers which are satisfied at time t. Classifiers which get the right to post their output messages are called winning classifiers.

Obviously, in this approach more than one winning classifier is allowed. Of course, other selection schemes are reasonable, for instance, the highest bidding agent wins alone. This is necessary to avoid the conflict between two winning classifiers. Now let us discuss how payoff from the environment is distributed and how the strengths are adapted. For this purpose, let us denote the set of classifiers, which have supplied a winning agent R_i in step t with $S_{i,t}$. Then the new strength of a winning agent is reduced by its bid and increased by its portion of the payoff P_t received from the environment:

$$u_{i,t+1} = u_{i,t} + \frac{P_t}{w_t} - B_{i,t}$$

where w_t is the number of winning agents in the actual time step. A winning agent pays its bid to its suppliers which share the bid among each other equally in the simplest case:

$$u_{i,t+1} = u_{i,t} + \frac{B_{i,t}}{|S_{i,t}|} \quad \text{for all } R_i \in S_{i,t}$$

If a winning agent has also been active in the previous step and supplies another winning agent, the value above is additionally increased by one portion of the bid the consumer offers. In the case that two winning agents have supplied each other mutually, the portions of the bids are exchanged in the above manner. The strengths of all other classifiers R_n, which are neither winning agents nor suppliers of winning agents, are reduced by a certain factor (they pay a tax):

$$u_{n,t+1} = u_{n,t}(1 - T)$$

T is a small value lying in the interval [0, 1]. The intention of taxation is to punish classifiers which never contribute anything to the output of the system. With this concept, redundant classifiers, which never become active, can be filtered out.

The idea behind credit assignment in general and bucket brigade in particular is to increase the strengths of rules, which have set the stage for later successful actions. The problem of determining such classifiers, which were responsible for conditions under which it was later on possible to receive a high payoff, can be

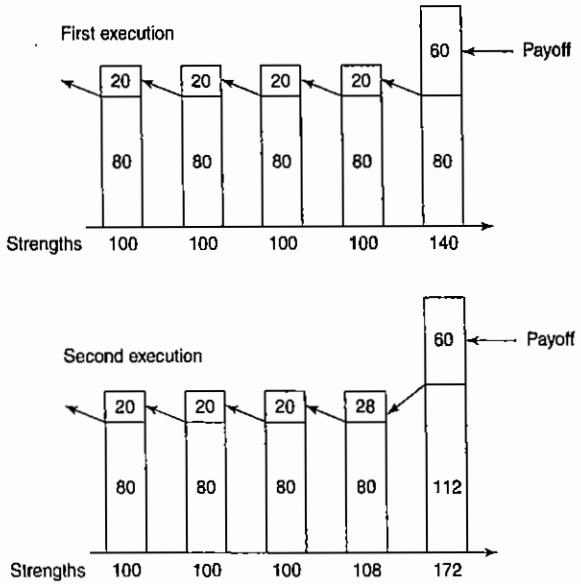


Figure 15-46 The bucket brigade principle.

very difficult. Consider, for instance, the game of chess again, in which very early moves can be significant for a late success or failure. In fact, the bucket brigade algorithm can solve this problem, although strength is only transferred to the suppliers, which were active in the previous step. Each time the same sequence is activated, however, a little bit of the payoff is transferred one step back in the sequence. It is easy to see that repeated successful execution of a sequence increases the strengths of all involved classifiers.

Figure 15-46 shows a simple example of how the bucket brigade algorithm works. For simplicity, we consider a sequence of five classifiers which always bid 20% of their strength. Only after the fifth step, after the activation of the fifth classifier, a payoff of 60 is received. The further development of the strengths in this example is shown in the Table 15-7. It is easy to see from this example that the reinforcement of the strengths is slow at the beginning, but it accelerates later. Exactly this property contributes much to the robustness of classifier systems – they tend to be cautious at the beginning, trying not to rush conclusions, but, after a certain number of similar situations, the system adopts the rules more and more.

It might be clear that a Holland classifier system only works if successful sequences of classifier activations are observed sufficiently often. Otherwise the bucket brigade algorithm does not have a chance to reinforce the strengths of the successful sequence properly.

15.15.3 Rule Generation

The purpose of the rule discovery system is to eliminate low-fitted rules and to replace them by hopefully better ones. The fitness of a rule is simply its strength. Since the classifiers of a Holland classifier system themselves are strings, the application of a GA to the problem of rule induction is straightforward, though many variants are reasonable. Almost all variants have one thing in common: the GA is not invoked in each time step, but only every nth step, where n has to be set such that enough information about the performance

Table 15-7 An example for repeated propagation of payoffs

Strength after the					
3rd	100.00	100.00	101.60	120.80	172.00
4th	100.00	100.32	103.44	136.16	197.60
5th	100.06	101.34	111.58	152.54	234.46
6th	100.32	103.39	119.78	168.93	247.57
⋮					
10th	106.56	124.17	164.44	224.84	278.52
⋮					
25th	215.86	253.20	280.36	294.52	299.24
⋮					
execution of the sequence					

of new classifiers can be obtained in the meantime. A. Geyer-Schulz, for instance, suggests the following procedure, where the strength of new classifiers is initialized with the average strength of the current rule base:

1. Select a subpopulation of a certain size at random.
2. Compute a new set of rules by applying the genetic operations – selection, crossingover and mutation – to this subpopulation.
3. Merge the new subpopulation with the rule base omitting duplicates and replace the worst classifiers.

This process of acquiring new rules has an interesting sideffect. It is more than just the exchange of parts of conditions and actions. Since we have not stated restrictions for manipulating tags, the GA can recombine parts of already existing tags to invent new tags. In the following, tags spawn related tags establishing new couplings. These new tags survive if they contribute to useful interactions. In this sense, the GA additionally creates experience-based internal structures autonomously.

15.16 Genetic Programming

Genetic programming (GP) is also part of the growing set of evolutionary algorithms that apply the search principles of natural evolution in a variety of different problem domains, notably parameter optimization. Evolutionary algorithms, and GP in particular, follow Darwin's principle of differential natural selection. This principle states that the following preconditions must be fulfilled for evolution to occur via (natural) selection:

1. There are entities called individuals which form a population. These entities can reproduce or can be reproduced.
2. There is heredity in reproduction, that is to say that individuals produce similar offspring.
3. In the course of reproduction, there is variety which affects the likelihood of survival and therefore of reproducibility of individuals.
4. There are finite resources which cause the individuals to compete. Owing to over reproduction of individuals not all can survive the struggle for existence. Differential natural selections will exert a continuous pressure towards improved individuals.

In the long run, GP and other evolutionary computing technologies will revolutionize program development. Present methods are not mature enough for deployment as automatic programming systems. Nevertheless, GP has already made inroads into automatic programming and will continue to do so in the foreseeable future. Likewise, the application of evolution in machine-learning problems is one of the potentials we will exploit over the coming decade.

GP is part of a more general field known as evolutionary computation. Evolutionary computation is based on the idea that basic concepts of biological reproduction and evolution can serve as a metaphor on which computer-based, goal-directed problem solving can be based. The general idea is that a computer program can maintain a population of artifacts represented using some suitable computer-based data structures. Elements of that population can then mate, mutate, or otherwise reproduce and evolve, directed by a fitness measure that assesses the quality of the population with respect to the goal of the task at hand.

GP is an automated method for creating a working computer program from a high-level problem statement of a problem. GP starts from a high-level statement of 'what needs to be done' and automatically creates a computer program to solve the problem.

One of the central challenges of computer science is to get a computer to do what needs to be done, without telling it how to do it. GP addresses this challenge by providing a method for automatically creating a working computer program from a high-level problem statement of the problem. GP achieves this goal of *automatic programming* (also sometimes called *program synthesis* or *program induction*) by genetically breeding a population of computer programs using the principles of Darwinian natural selection and biologically inspired operations. The operations include reproduction, crossover, mutation and architecture-altering operations patterned after gene duplication and gene deletion in nature.

GP is a domain-independent method that genetically breeds a population of computer programs to solve a problem. Specifically, GP iteratively transforms a population of computer programs into a new generation of programs by applying analogs of naturally occurring genetic operations. The genetic operations include crossover, mutation, reproduction, gene duplication and gene deletion. GP is an excellent problem solver, a superb function approximator and an effective tool for writing functions to solve specific tasks. However, despite all these areas in which it excels, it still does not replace programmers; rather, it helps them. A human still must specify the fitness function and identify the problem to which GP should be applied.

15.16.1 Working of Genetic Programming

GP typically starts with a population of randomly generated computer programs composed of the available programmatic ingredients. GP iteratively transforms a population of computer programs into a new generation of the population by applying analogs of naturally occurring genetic operations. These operations are applied to individual(s) selected from the population. The individuals are probabilistically selected to participate in the genetic operations based on their fitness (as measured by the fitness measure provided by the human user in the third preparatory step). The iterative transformation of the population is executed inside the main generational loop of the run of GP.

The executional steps of GP (i.e., the flowchart of GP) are as follows:

1. Randomly create an initial population (generation 0) of individual computer programs composed of the available functions and terminals.
2. Iteratively perform the following substeps (called a *generation*) on the population until the termination criterion is satisfied:
   - Execute each program in the population and ascertain its fitness (explicitly or implicitly) using the problem's fitness measure.
   - Select one or two individual program(s) from the population with a probability based on fitness (with reselection allowed) to participate in the genetic operations in the next substep.
   - Create new individual program(s) for the population by applying the following genetic operations with specified probabilities:
     (a) *Reproduction*: Copy the selected individual program to the new population.
     (b) *Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts from two selected programs.
     (c) *Mutation*: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one selected program.
     (d) *Architecture-altering operations*: Choose an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the chosen architecture-altering operation to one selected program.
3. After the termination criterion is satisfied, the single best program in the population produced during the run (the best-so-far individual) is harvested and designated as the result of the run. If the run is successful, the result may be a solution (or approximate solution) to the problem.

GP is problem-independent in the sense that the flowchart specifying the basic sequence of executional steps is not modified for each new run or each new problem. There is usually no discretionary human intervention or interaction during a run of genetic programming (although a human user may exercise judgment as to whether to terminate a run).

Figure 15-47 below is a flowchart showing the executional steps of a run of GP. The flowchart shows the genetic operations of crossover, reproduction and mutation as well as the architecture-altering operations. This flowchart shows a two-offspring version of the crossover operation.

The flowchart of GP is explained as follows: GP starts with an initial population of computer programs composed of functions and terminals appropriate to the problem. The individual programs in the initial population are typically generated by recursively generating a rooted point-labeled program tree composed of random choices of the primitive functions and terminals (provided by the human user as part of the first and second preparatory steps, a run of GP). The initial individuals are usually generated subject to a pre-established maximum size (specified by the user as a minor parameter as part of the fourth preparatory step). In general, the programs in the population are of different sizes (number of functions and terminals) and of different shapes (the particular graphical arrangement of functions and terminals in the program tree).

Each individual program in the population is executed. Then, each individual program in the population is either measured or compared in terms of how well it performs the task at hand (using the fitness measure provided in the third preparatory step). For many problems, this measurement yields a single explicit numerical value called *fitness*. The fitness of a program may be measured in many different ways, including, for example, in terms of the amount of error between its output and the desired output, the amount of time (fuel, money, etc.) required to bring a system to a desired target state, the accuracy of the program in recognizing patterns or classifying objects into classes, the payoff that a game-playing program produces, or the compliance of a complex structure (such as an antenna, circuit, or controller) with user-specified design criteria. The execution of the program sometimes returns one or more explicit values. Alternatively, the execution of a program may consist only of side effects on the state of a world (e.g., a robot's actions). Alternatively, the execution of a program may produce both return values and side effects.

The fitness measure is, for many practical problems, multiobjective in the sense that it combines two or more different elements. The different elements of the fitness measure are often in competition with one another to some degree.

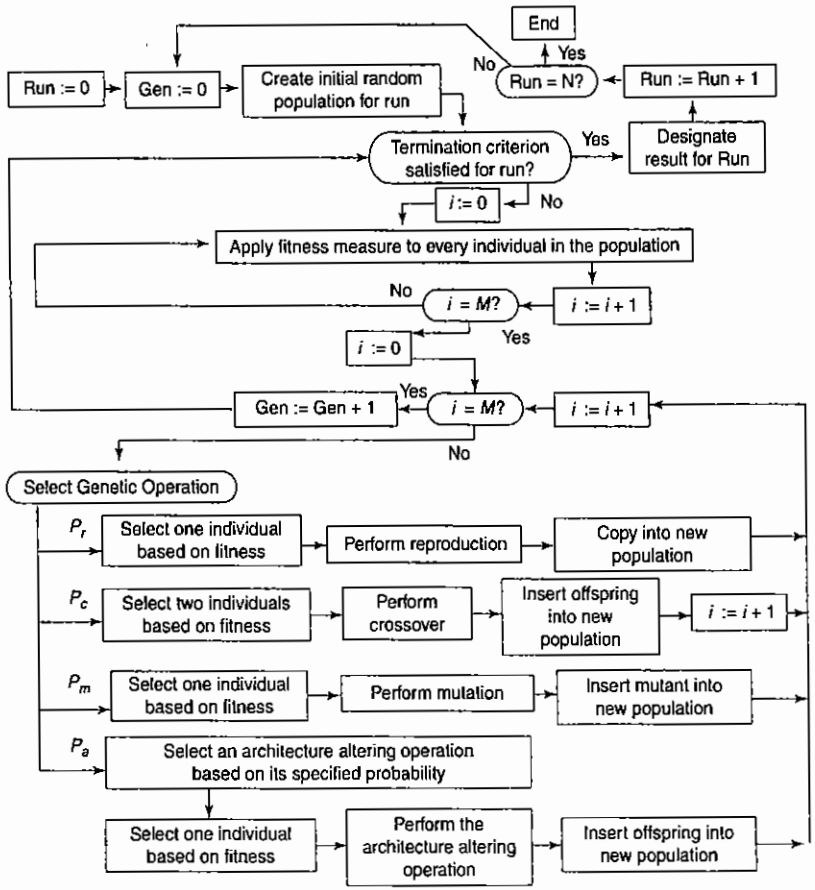


Figure 15-47 Flowchart of genetic programming.

For many problems, each program in the population is executed over a representative sample of different *fitness cases*. These fitness cases may represent different values of the program's input(s), different initial conditions of a system, or different environments. Sometimes the fitness cases are constructed probabilistically.

The creation of the initial random population is, in effect, a blind random search of the search space of the problem. It provides a baseline for judging future search efforts. Typically, the individual programs in generation 0 all have exceedingly poor fitness. Nevertheless, some individuals in the population are (usually) more fit than others. The differences in fitness are then exploited by GP. GP applies Darwinian selection and the genetic operations to create a new population of offspring programs from the current population.

The genetic operations include crossover, mutation, reproduction and the architecture-altering operations. These genetic operations are applied to individual(s) that are probabilistically selected from the population based on fitness. In this probabilistic selection process, better individuals are favored over inferior individuals.

However, the best individual in the population is not necessarily selected and the worst individual in the population is not necessarily passed over.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the current population (i.e., the now-old generation). This iterative process of measuring fitness and performing the genetic operations is repeated over many generations.

The run of GP terminates when the termination criterion (as provided by the fifth preparatory step) is satisfied. The outcome of the run is specified by the method of result designation. The best individual ever encountered during the run (i.e., the best-so-far individual) is typically designated as the result of the run.

All programs in the initial random population (generation 0) of a run of GP are syntactically valid, executable programs. The genetic operations that are performed during the run (i.e., crossover, mutation, reproduction and the architecture-altering operations) are designed to produce offspring that are syntactically valid, executable programs. Thus, every individual created during a run of genetic programming (including, in particular, the best-of-run individual) is a syntactically valid, executable program.

15.16.2 Characteristics of Genetic Programming

GP now routinely delivers high-return human-competitive machine intelligence, the next four subsections explain what we mean by the terms human-competitive, high-return, routine and machine intelligence.

15.16.2.1 Human-Competitive

In attempting to evaluate an automated problem-solving method, the question arises as to whether there is any real substance to the demonstrative problems that are published in connection with the method. Demonstrative problems in the fields of artificial intelligence and machine learning are often contrived to problems that circulate exclusively inside academic groups that study a particular methodology. These problems typically have little relevance to any issues pursued by any scientist or engineer outside the fields of artificial intelligence and machine learning.

In his 1983 talk entitled "*AI: Where It Has Been and Where It Is Going*," machine learning pioneer Arthur Samuel said:

> *The aim is . . . to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence.*

Samuel's statement reflects the common goal articulated by the pioneers of the 1950s in the fields of artificial intelligence and machine learning. Indeed, getting machines to produce human-like results is the reason for the existence of the fields of artificial intelligence and machine learning. To make this goal more concrete, we say that a result is "human-competitive" if it satisfies one or more of the eight criteria in Table 15-8. These eight criteria have the desirable attribute of being at arms-length from the fields of artificial intelligence, machine learning and GP. That is, a result cannot acquire the rating of 'human-competitive' merely because it is endorsed by researchers *inside* the specialized fields that are attempting to create machine intelligence. Instead, a result produced by an automated method must earn the rating of human-competitive *independent* of the fact that it was generated by an automated method.

15.16.2.2 High-Return

What is delivered by the actual automated operation of an artificial method in comparison to the amount of knowledge, information, analysis and intelligence that is pre-supplied by the human employing the method?

We define the *AI ratio* (the 'artificial-to-intelligence' ratio) of a problem-solving method as the ratio of that which is delivered by the automated operation of the *artificial* method to the amount of *intelligence* that is supplied by the human applying the method to a particular problem.

Table 15-8 Eight criteria for saying that an automatically created result is human-competitive

	Criterion
A	The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
B	The result is equal to or better than a result that was accepted as a new scientific result at the time when it was published in a peer-reviewed scientific journal.
C	The result is equal to or better than a result that was placed into a database or archive of results maintained by an internationally recognized panel of scientific experts.
D	The result is publishable in its own right as a new scientific result–independent of the fact that the result was mechanically created.
E	The result is equal to or better than the most recent human-created solution to a long-standing problem for which there has been a succession of increasingly better human-created solutions.
F	The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.
G	The result solves a problem of indisputable difficulty in its field.
H	The result holds its own or wins a regulated competition involving human contestants (in the form of either live human players or human-written computer programs).

The AI ratio is especially pertinent to methods for getting computers to automatically solve problems because it measures the value added by the artificial problem-solving method. Manifestly, the aim of the fields of artificial intelligence and machine learning is to generate human-competitive results with a high AI ratio.

Deep Blue: An Artificial Intelligence Milestone (Newborn, 2002) describes the 1997 defeat of the human world chess champion Garry Kasparov by the Deep Blue computer system. This outstanding example of machine intelligence is clearly a human-competitive result (by virtue of satisfying criterion H of Table 15-8). Feng-Hsiung Hsu (the system architect and chip designer for the Deep Blue project) recounts the intensive work on the Deep Blue project at IBM's T. J. Watson Research Center between 1989 and 1997 (Hsu, 2002). The team of scientists and engineers spent years developing the software and the specialized computer chips to efficiently evaluate large numbers of alternative moves as part of a massive parallel state-space search. In short, the human developers invested an enormous amount of "I" in the project. In spite of the fact that Deep Blue delivered a high (human-competitive) amount of "A," the project has a low return when measured in terms of the A-to-I ratio.

The aim of the fields of artificial intelligence and machine learning is to get computers to automatically generate human-competitive results with a high AI ratio – not to have humans generate human-competitive results themselves.

15.16.2.3 Routine

Generality is a precondition to what we mean when we say that an automated problem-solving method is "routine." Once the generality of a method is established, "routineness" means that relatively little human effort is required to get the method to successfully handle new problems within a particular domain and to successfully handle new problems from a different domain. The ease of making the transition to new problems lies at the heart of what we mean by routine. A problem-solving method cannot be considered routine if its executional steps must be substantially augmented, deleted, rearranged, reworked or customized by the human user for each new problem.

15.16.2.4 Machine Intelligence

We use the term machine intelligence to refer to the broad vision articulated in Alan Turing's 1948 paper entitled "*Intelligent Machinery*" and his 1950 paper entitled "*Computing Machinery and Intelligence.*"

In the 1950s, the terms machine intelligence, artificial intelligence and machine learning all referred to the *goal* of getting "machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence" (to again quote Arthur Samuel).

However, in the intervening five decades, the terms "artificial intelligence" and "machine learning" progressively diverged from their original goal-oriented meaning. These terms are now primarily associated with particular *methodologies* for attempting to achieve the goal of getting computers to automatically solve problems. Thus, the term "artificial intelligence" is today primarily associated with attempts to get computers to solve problems using methods that rely on knowledge, logic, and various analytical and mathematical methods. The term "machine learning" is today primarily associated with attempts to get computers to solve problems that use a particular small and somewhat arbitrarily chosen set of methodologies (many of which are statistical in nature). The narrowing of these terms is in marked contrast to the broad field envisioned by Samuel at the time when he coined the term "machine learning" in the 1950s, the charter of the original founders of the field of artificial intelligence, and the broad vision encompassed by Turing's term "machine intelligence." Of course, the shift in focus from broad goals to narrow methodologies is an all too common sociological phenomenon in academic research.

Turing's term "machine intelligence" did not undergo this arteriosclerosis because, by accident of history, it was never appropriated or monopolized by any group of academic researchers whose primary dedication is to a particular methodological approach. Thus, Turing's term remains catholic today. We prefer to use Turing's term because it still communicates the broad *goal* of getting computers to automatically solve problems in a human-like way.

In his 1948 paper, Turing identified three broad approaches by which human competitive machine intelligence might be achieved: The first approach was a logic-driven search. Turing's interest in this approach is not surprising in light of Turing's own pioneering work in the 1930s on the logical foundations of computing. The second approach for achieving machine intelligence was what he called a "cultural search" in which previously acquired knowledge is accumulated, stored in libraries and brought to bear in solving a problem – the approach taken by modern knowledge-based expert systems. Turing's first two approaches have been pursued over the past 50 years by the vast majority of researchers using the methodologies that are today primarily associated with the term "artificial intelligence."

15.16.3 Data Representation

Without any doubt, programs can be considered as strings. There are, however, two important limitations which make it impossible to use the representations and operations from our simple GA:

1. It is mostly inappropriate to assume a fixed length of programs.
2. The probability to obtain syntactically correct programs when applying our simple initialization, crossover and mutation procedures is hopelessly low.

It is, therefore, indispensable to modify the data representation and the operations such that syntactical correctness is easier to guarantee. The common approach to represent programs in GP is to consider programs as trees. By doing so, initialization can be done recursively, crossover can be done by exchanging subtrees and random replacement of subtrees can serve as mutation operation.

Since their only construct are nested lists, programs in LISP-like languages already have a kind of tree-like structure. Figure 15-48 shows an example how the function $3x + \sin(x + 1)$ can be implemented in a LISP-like language and how such an LISP-like function can be split up into a tree. It can be noted that the tree representation corresponds to the nested lists. The program consists of tomic expressions, like variables and constants, which act as leave nodes while functions act as nonleave nodes.

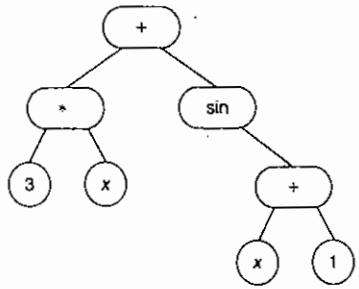


Figure 15-48 The tree representation of $3x + \sin(x + 1)$.

There is one important disadvantage of the LISP approach – it is difficult to introduce type checking. In case of a purely numeric function like in the above example, there is no problem at all. However, it can be desirable to process numeric data, strings and logical expressions simultaneously. This is difficult to handle if we use a tree representation like that in Figure 15-48.

A. Geyer-Schulz has proposed a very general approach, which overcomes this problem allowing maximum flexibility. He suggested representing programs by their syntactical derivation trees with respect to a recursive definition of underlying language in Backus-Naur form (BNF). This works for any context-free language. It is far beyond the scope of this lecture to go into much detail about formal languages. We will explain the basics with the help of a simple example. Consider the following language which is suitable for implementing binary logical expressions:

```
S     := <exp>
<exp> := (var) | "("<neg> <exp>")" | "("<exp> <bin> <exp>")";
<var> := "x" | "y";
<neg> := "NOT"
<bin> := "AND" | "OR";
```

The BNF description consists of so-called syntactical rules. Symbols in angular brackets < > are called nonterminal symbols, i.e., symbols which have to be expanded. Symbols between quotation marks are called terminal symbols, i.e., they cannot be expanded any further. The first rule S:=<exp> defines the starting symbol. A BNF rule of the general shape,

$$\text{<nonterminal>} := \text{<deriv}_1\text{>} \mid \text{<deriv}_2\text{>} \mid \ldots \mid \text{<deriv}_n\text{>};$$

defines how a non-terminal symbol may be expanded, where the different variants are separated by vertical bars.

In order to get a feeling of how to work with the BNF grammar description, we will now show step-by-step how the expression (NOT (*x* OR *y*)) can be derivated from the above language. For simplicity, we omit quotation marks for the terminal symbols:

1. We have to begin with the start symbol: <exp>
2. We replace hexpi with the second possible derivation:

$$\text{<exp>} \rightarrow (\text{<neg> <exp>})$$

3. The symbol <neg> may only he expanded with the terminal symbol NOT:

$$(\text{<neg> <exp>}) \rightarrow (\text{NOT <exp>})$$

4. Next, we replace <exp> with the third possible derivation:

$$(\text{NOT <exp>}) \rightarrow (\text{NOT (<exp> <bin> <exp>)})$$

5. We expand the second possible derivation for <bin>:

$$(\text{NOT (<exp> <bin> <exp>)}) \rightarrow (\text{NOT (<exp> OR <exp>)})$$

6. The first occurrence of <exp> is expanded with the first derivation:

$$(\text{NOT(<exp> OR <exp>)}) \rightarrow (\text{NOT (<var> OR <exp>)})$$

7. The second occurrence of <exp> is expanded with the first derivation, too:

$$(\text{NOT (<var> OR <exp>)}) \rightarrow (\text{NOT (<var> OR <var>)})$$

8. Now we replace the first <var> with the corresponding first alternative:

$$(\text{NOT (<var> OR <var>)}) \rightarrow (\text{NOT (}x\text{ OR <var>)})$$

9. Finally, the last non-terminal symbol is expanded with the second alternative:

$$(\text{NOT (}x\text{ OR <var>)}) \rightarrow (\text{NOT (}x\text{ OR }y\text{)})$$

Such a recursive derivation has an inherent tree structure. For the above example, this derivation tree has been visualized in Figure 15-49. The syntax of modern programming languages can be specified in BNF. Hence, our data model would be applicable to all of them. The question is whether this is useful. Koza's hypothesis includes that the programming language has to be chosen such that the given problem is solvable. This does not necessarily imply that we have to choose the language such that virtually any solvable problem can be solved. It is obvious that the size of the search space grows with the complexity of the language. We know that the size of the search space influences the performance of a GA – the larger the slower.

It is, therefore, recommendable to restrict the language to necessary constructs and to avoid superfluous constructs. Assume, for example, that we want to do symbolic regression, but we are only interested in polynomials with integer coefficients. For such an application, it would be an overkill to introduce rational constants or to include exponential functions in the language. A good choice could be the following:

```
S       := <func>;
<func>  := <var> | "("<const> | "("<func> <bin> <func>")";
<var>   := "x";
<const> := <int> | <const> <int>;
<int>   := "0" | ... | "9";
<bin>   := "+" | "-" | "*";
```

For representing rational functions with integer coefficients, it is sufficient to add the division symbol "/" to the possible derivations of the binary operator <bin>.

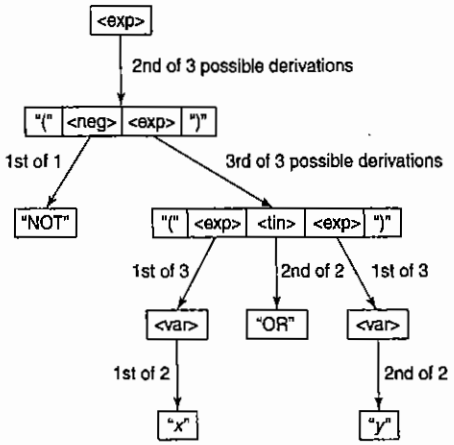


Figure 15-49 The derivation tree of (NOT (x OR y)).

Another example: The following language could be appropriate for discovering trigonometric identities:

```
S     := <func>;
<func> := <var> | <const> | <trig> "("<func>")" | "("<func> <bin> <func>")"
<var>  := "x";
<const> := "0" | "1" | "π";
<trig>  := "sin" | "cos";
<bin>   := "+" | "−" | "+";
```

There are basically two different variants of how to generate random programs with respect to a given BNF grammar:

1. Beginning from the starting symbol, it is possible to expand nonterminal symbols recursively, where we have to choose randomly if we have more than one alternative derivation. This approach is simple and fast, but has some disadvantages: First, it is almost impossible to realize a uniform distribution. Second, one has to implement some constraints with respect to the depth of the derivation trees in order to avoid excessive growth of the programs. Depending on the complexity of the underlying grammar, this can be a tedious task.
2. Geyer-Schulz has suggested to prepare a list of all possible derivation trees up to a certain depth and to select from this list randomly applying a uniform distribution. Obviously, in this approach, the problems in terms of depth and the resulting probability distribution are elegantly solved, but these advantages go along with considerably long computation times.

15.16.3.1 Crossing Programs

It is trivial to see that primitive string-based crossover of programs almost never yields syntactically correct programs. Instead, we should use the perfect syntax information a derivation tree provides. Already in the LISP times of Gp, sometime before the BNF-based representation was known, crossover was usually implemented

as the exchange of randomly selected subtrees. In case that the subtrees (subexpressions) may have different types of return values (e.g., logical and numerical), it is not guaranteed that crossover preserves syntactical correctness.

The derivation tree-based representation overcomes this problem in a very elegant way. If we only exchange subtrees which start from the same nonterminal symbol, crossover can never violate syntactical correctness. In this sense, the derivation tree model provides implicit type checking. In order to demonstrate in more detail how this crossover operation works, let us reconsider the example of binary logical expressions. As parents, we take the following expressions:

(NOT (x OR y))

((NOT x) OR (x AND y))

Figure 15-50 shows graphically how the two children (NOT (x OR (x AND y))) ((NOT x) OR y) are obtained.

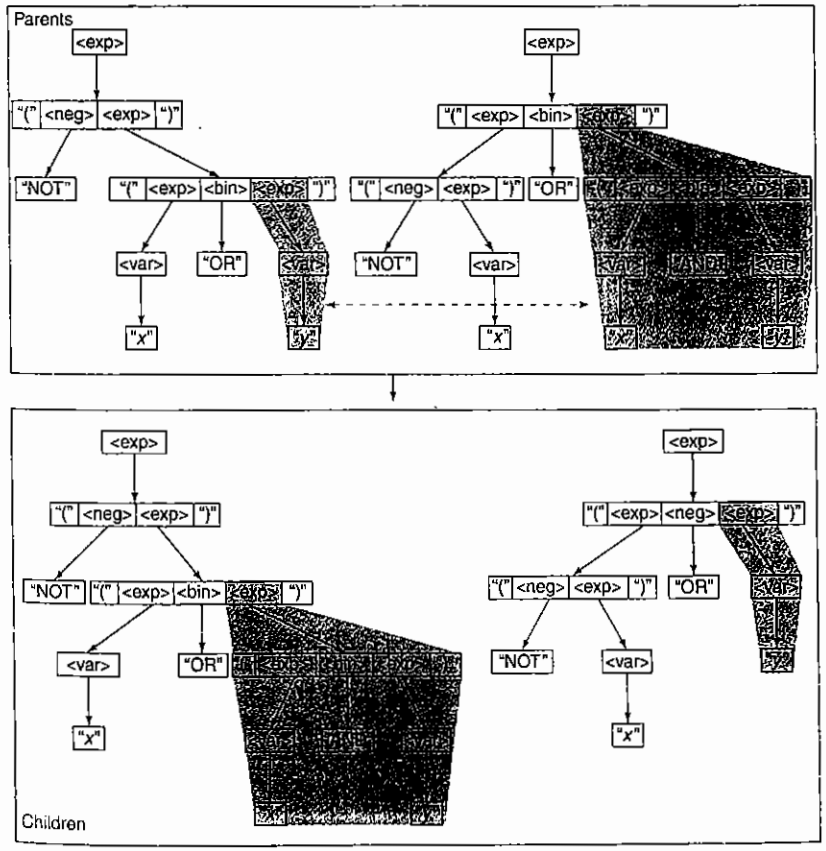


Figure 15-50 An example for crossing two binary logical expressions.

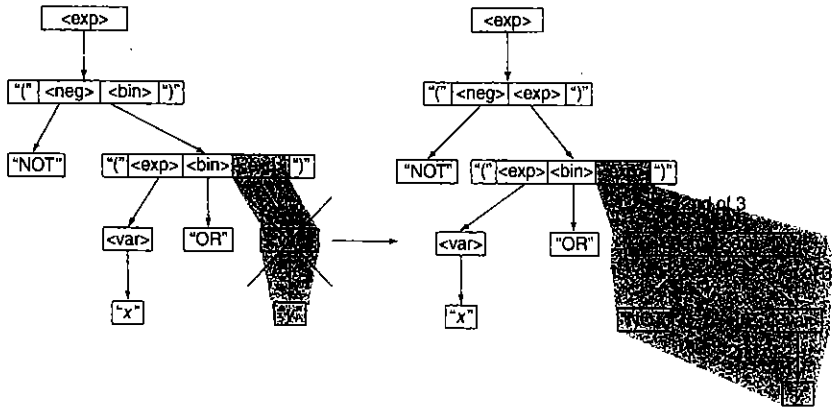


Figure 15-51 An example for mutating a derivation tree.

15.16.3.2 Mutating Programs

We have always considered mutation as the random deformation of a small part of a chromosome. It is, therefore, not surprising that the most common mutation in genetic programming is the random replacement of a randomly selected subtree. The only modification is that we do not necessarily start from the start symbol, but from the nonterminal symbol at the root of the subtree we consider. Figure 15-51 shows an example where in the logical expression (NOT (x OR y)), the variable y is replaced by (NOT y).

15.16.3.3 The Fitness Function

There is no common recipe for specifying an appropriate fitness function which strongly depends on the given problem. It is, however, worth emphasizing that it is necessary to provide enough information to guide the GA to the solution. More specifically, it is not sufficient to define a fitness function which assigns 0 to a program which does not solve the problem and 1 to a program which solves the problem – such a fitness function would correspond to a needle-in-haystack problem. In this sense, a proper fitness measure should be a gradual concept for judging the correctness of programs.

In many applications, the fitness function is based on a comparison of desired and actually obtained output. Koza, for instance, uses the simple sum of quadratic errors for symbolic regression and the discovery of trigonometric identities:

$$f(F) = \sum_{i=1}^{N} |y_i - F(x_i)|^2$$

In this definition, F is the mathematical function which corresponds to the program under evaluation. The list (x_i, y_i), $1 \le i \le N$ consists of reference pairs – a desired output y_i is assigned to each input x_i. Clearly, the samples have to be chosen such that the considered input space is covered sufficiently well.

Numeric error-based fitness functions usually imply minimization problems. Some other applications may imply maximization tasks. There are basically two well-known transformations which allow to standardize fitness functions such that always minimization or maximization tasks are obtained.

Consider an arbitrary "raw" fitness function f. Assuming that the number of individuals in the population is not fixed (m_t at time t), the *standardized fitness* is computed as

$$f_S(b_{i,t}) = f(b_{i,t}) - \max_{j=1}^{m_t} f(b_{j,t})$$

if f has to be maximized and as

$$f_S(b_{i,t}) = f(b_{i,t}) - \min_{j=1}^{m_t} f(b_{j,t})$$

if f has to be minimized. One possible variant is to consider the best individual of the last k generations instead of only considering the actual generation.

Obviously, standardized fitness transforms any optimization problem into a minimization task. Roulette wheel selection relies on the fact that the objective is maximization of the fitness function. Koza has suggested a simple transformation such that, in any case, a maximization problem is obtained.

With the assumptions of previous definition, the adjusted fitness is computed as

$$f_A(b_{i,t}) = \max_{j=1}^{m_t} f_S(b_{j,t}) - f_S(b_{j,t})$$

Another variant of adjusted fitness is defined as

$$f'_A(b_{i,t}) = \frac{1}{1 + f_S(b_{j,t})}$$

For applying GP to a given problem, the following points have to be satisfied.

1. An appropriate fitness function, which provides enough information to guide the GA to the solution (mostly based on examples).
2. A syntactical description of a programming language, which contains as much elements as necessary for solving the problem.
3. An interpreter for the programming language.

The main application areas of GP include: Computer Science, Science, Engineering, Art and Entertainment.

15.17 Advantages and Limitations of Genetic Algorithm

The advantages of GA are as follows:

1. Parallelism.
2. Liability.
3. Solution space is wider.
4. The fitness landscape is complex.
5. Easy to discover global optimum.
6. The problem has multiobjective function.

The limitations of GA are as follows:

1. The problem of identifying fitness function.
2. Definition of representation for the problem.
3. Premature convergence occurs.
4. The problem of choosing various parameters such as the size of the population, mutation rate, crossover rate, the selection method and its strength.

15.18 Applications of Genetic Algorithm

An effective GA representation and meaningful fitness evaluation are the keys of the success in GA applications. The appeal of GAs comes from their simplicity and elegance as robust search algorithms as well as from their power to discover good solutions rapidly for difficult high-dimensional problems. GAs are useful and efficient when

1. the search space is large, complex or poorly understood;
2. domain knowledge is scarce or expert knowledge is difficult to encode to narrow the search space;
3. no mathematical analysis is available;
4. traditional search methods fail.

The advantage of the GA approach is the ease with which it can handle arbitrary kinds of constraints and objectives; all such things can be handled as weighted components of the fitness function, making it easy to adapt the GA scheduler to the particular requirements of a very wide range of possible overall objectives.

GAs have been used for problem-solving and for modeling. GAs are applied to many scientific, engineering problems, in business and entertainment, including:

1. *Optimization*: GAs have been used in a wide variety of optimization tasks, including numerical optimization and combinatorial optimization problems such as traveling salesman problem (TSP), circuit design (Louis, 1993), job shop scheduling (Goldstein, 1991) and video & sound quality optimization.
2. *Automatic programming*: GAs have been used to evolve computer programs for specific tasks and to design other computational structures, for example, cellular automata and sorting networks.
3. *Machine and robot learning*: GAs have been used for many machine-learning applications, including classifications and prediction, and protein structure prediction. GAs have also been used to design neural networks, to evolve rules for learning classifier systems or symbolic production systems, and to design and control robots.
4. *Economic models*: GAs have been used to model processes of innovation, the development of bidding strategies and the emergence of economic markets.
5. *Immune system models*: GAs have been used to model various aspects of the natural immune system, including somatic mutation during an individual's lifetime and the discovery of multi-gene families during evolutionary time.
6. *Ecological models*: GAs have been used to model ecological phenomena such as biological arms races, host–parasite co-evolutions, symbiosis and resource flow in ecologies.
7. *Population genetics models*: GAs have been used to study questions in population genetics, such as 'under what conditions will a gene for recombination be evolutionarily viable?'
8. *Interactions between evolution and learning*: GAs have been used to study how individual learning and species evolution affect one another.

9. *Models of social systems*: GAs have been used to study evolutionary aspects of social systems, such as the evolution of cooperation (Chughtai, 1995), the evolution of communication and trail-following behavior in ants.

15.19 Summary

Genetic algorithms are original systems based on the supposed functioning of the living. The method is very different from classical optimization algorithms as it:

1. Uses the encoding of the parameters, not the parameters themselves.
2. Works on a population of points, not a unique one.
3. Uses the only values of the function to optimize, not their derived function or other auxiliary knowledge.
4. Uses probabilistic transition function and not determinist ones.

It is important to understand that the functioning of such an algorithm does not guarantee success. The problem is in a stochastic system and a genetic pool may be too far from the solution, or for example, a too fast convergence may halt the process of evolution. These algorithms are, nevertheless, extremely efficient, and are used in fields as diverse as stock exchange, production scheduling or programming of assembly robots in the automotive industry.

GAs can even be faster in finding global maxima than conventional methods, in particular when derivatives provide misleading information. It should be noted that in most cases where conventional methods can be applied, GAs are much slower because they do not take auxiliary information such as derivatives into account. In these optimization problems, there is no need to apply a GA, which gives less accurate solutions after much longer computation time. The enormous potential of GAs lies elsewhere – in optimization of non-differentiable or even discontinuous functions, discrete optimization, and program induction.

It has been claimed that via the operations of selection, crossover and mutation, the GA will converge over successive generations towards the global (or near global) optimum. This simple operation should produce a fast, useful and robust technique largely because of the fact that GAs combine direction and chance in the search in an effective and efficient manner. Since population implicitly contain much more information than simply the individual fitness scores, GAs combine the good information hidden in a solution with good information from another solution to produce new solutions with good information inherited from both parents, inevitably (hopefully) leading towards optimality.

In this chapter we have also discussed the various classifications of GAs. The class of parallel GAs is very complex, and its behavior is affected by many parameters. It seems that the only way to achieve a greater understanding of parallel GAs is to study individual facets independently, and we have seen that some of the most influential publications in parallel GAs concentrate on only one aspect (migration rates, communication topology or deme size) either ignoring or making simplifying assumptions on the others. Also the hybrid GA, adaptive GA, independent sampling GA and messy GA has been included with the necessary information.

Genetic programming has been used to model and control a multitude of processes and to govern their behavior according to fitness-based automatically generated algorithms. Implementation of genetic programming will benefit in the coming years from new approaches which include research from developmental biology. Also, it will be necessary to learn to handle the redundancy forming pressures in the evolution of code. Application of genetic programming will continue to broaden. Many applications focus on controlling behavior of real or virtual agents. In this role, genetic programming may contribute considerably to the growing field of social and behavioral simulations. A brief discussion on Holland classifier system is also included in this chapter.

15.20 Review Questions

1. State Charles Darwin's theory of evolution.
2. What is meant by genetic algorithm?
3. Compare and contrast traditional algorithm and genetic algorithm.
4. State the importance of genetic algorithm.
5. Explain in detail about the various operators involved in genetic algorithm.
6. What the various types of crossover and mutation techniques?
7. With a neat flowchart, explain the operation of a simple genetic algorithm.
8. State the general genetic algorithm.
9. Discuss in detail about the various types of genetic algorithm in detail.
10. State schema theorem.
11. Write short note on Holland classifier systems.
12. Differentiate between messy GA and parallel GA.
13. What is the importance of hybrid GAs?
14. Describe the concepts involved in real-coded genetic algorithm.
15. What is genetic programming?
16. Compare genetic algorithm and genetic programming.
17. List the characteristics of genetic programming.
18. With a neat flowchart, explain the operation of genetic programming.
19. How are data represented in genetic programming?
20. Mention the applications of genetic algorithm.

15.21 Exercise Problems

1. Determine the maximum of function $x \times x^5 \times (0.007x + 2)$ using genetic algorithm by w iting a program.
2. Determine the maximum of function $\exp(-3x) + \sin(6\pi x)$ using genetic algorithm. Given range = [0.004 0.7]; bits = 6; population = 12; generations = 36; mutation = 0.005; matenum = 0.3.
3. Optimize the logarithmic function using a genetic algorithm by writing a program.
4. Solve the logical AND function using genetic algorithm by writing a program.
5. Solve the XNOR problem using genetic algorithm by writing a program.
6. Determine the maximum of function $\exp(5x) + \sin(7\pi x)$ using genetic algorithm. Given range = [0.002 0.6]; bits = 3; popu ation = 14; generations = 36; mutation = 0.006; matenum = 0.3.

Hybrid Soft Computing Techniques

16

Learning Objectives

- Neuro-fuzzy hybrid systems.
- Comparison of fuzzy systems with neural networks.
- Properties of neuro-fuzzy hybrid systems.
- Characteristics of neuro-fuzzy hybrids.
- Cooperative neural fuzzy systems.
- General neuro-fuzzy hybrid systems.
- Adaptive Neuro-fuzzy Inference System (ANFIS) in MATLAB.
- Genetic neuro hybrid systems.
- Properties of genetic neuro hybrid systems.
- Genetic algorithm based back-propagation network (BPN).
- Advantages of neuro-genetic hybrids.
- Genetic fuzzy hybrid and fuzzy genetic hybrid systems.
- Genetic fuzzy rule based systems (GFRBSs).
- Advantages of genetic fuzzy hybrids.
- Simplified fuzzy ARTMAP.
- Supervised ARTMAP system.

16.1 Introduction

In general, neural networks, fuzzy systems and genetic algorithms are distinct soft computing techniques evolved from the biological computational strategies and nature's way to solve problems.

Neural networks are the simplified models of the human nervous systems mimicking our ability to adapt to certain situations and to learn from the past experiences. Chapters 2–6 of the book discuss the basics of artificial neural networks, supervised and unsupervised learning neural networks, associative memory networks, and few other special networks. Fuzzy logic or fuzzy systems deal with uncertainty or vagueness existing in a system and formulating fuzzy rules to find a solution to problems. Fuzzy logic does not operate on accurate boundaries and it provides a transition between membership and non-membership of the variables for a particular problem. Chapters 7–14 of the book discuss the basic concepts of fuzzy sets, fuzzy relations, and methods for formulation of membership functions and for converting fuzzy entities to crisp entities, fuzzy arithmetic, fuzzy rule base, and fuzzy control system with its applications. Genetic algorithms inspired by the natural evolution process are adaptive search and optimization algorithms. Chapter 15 of the book discusses on the fundamental genetic operators and general genetic algorithm used for finding an optimal solution.

All the above three techniques individually have provided efficient solutions to a wide range of simple and complex problems pertaining to different domains. As discussed in Section 1.5 of Chapter 1, these three techniques can be combined together in whole or in part, and may be applied to find solution to the problems, where the techniques do not work individually. The main aim of the concept of hybridization is to overcome the weakness in one technique while applying it and bringing out the strength of the other

technique to find solution by combining them. Every soft computing technique has particular computational parameters (e.g., ability to learn, decision making) which make them suited for a particular problem and not for others. It has to be noted that neural networks are good at recognizing patterns but they are not good at explaining how they reach their decisions. On the contrary, fuzzy logic is good at explaining the decisions but cannot automatically acquire the rules used for making the decisions. Also, the tuning of membership functions becomes an important issue in fuzzy modeling. Since this tuning can be viewed as an optimization problem, either neural network (Hopfield neural network gives solution to optimization problem) or genetic algorithms offer a possibility to solve this problem. These limitations act as a central driving force for the creation of hybrid soft computing systems where two or more techniques are combined in a suitable manner that overcomes the limitations of individual techniques.

The importance of hybrid system is based on the varied nature of the application domains. Many complex domains have several different component problems each of which may require different types of processing. When there is a complex application which has two distinct sub-problems, say for example, a signal processing and serial shift reasoning, then a neural network and fuzzy logic can be used for solving these individual tasks, respectively. The use of hybrid systems is growing rapidly with successful applications in areas such as engineering design, stock market analysis and prediction, medical diagnosis, process control, credit card analysis, and few other cognitive simulations.

Thus, even though the hybrid soft computing systems have a great potential to solve problems, if not applied appropriately they may result in adverse solutions. It is not necessary that when individual techniques give good solution, hybrid systems would give an even better solution. The key driving force is to build highly automated, intelligent machines for the future generations using all these techniques.

16.2 Neuro-Fuzzy Hybrid Systems

A *neuro-fuzzy hybrid system* (also called fuzzy neural hybrid), proposed by J. S. R. Jang, is a learning mechanism that utilizes the training and learning algorithms from neural networks to find parameters of a fuzzy system (i.e., fuzzy sets, fuzzy rules, fuzzy numbers, and so on). It can also be defined as a fuzzy system that determines its parameters by processing data samples by using a learning algorithm derived from or inspired by neural network theory. Alternately, it is a hybrid intelligent system that fuses artificial neural networks and fuzzy logic by combining the learning and connectionist structure of neural networks with human-like reasoning style of fuzzy systems.

Neuro-fuzzy hybridization is widely termed as Fuzzy Neural Network (FNN) or Neuro-Fuzzy System (NFS). The human-like reasoning style of fuzzy systems is incorporated by NFS (the more popular term is used henceforth) through the use of fuzzy sets and a linguistic model consisting of a set of IF–THEN fuzzy rules. NFSs are universal approximators with the ability to solicit interpretable IF–THEN rules; this is their main strength. However, the strength of NFSs involves interpretability versus accuracy, requirements that are contradictory in fuzzy modeling.

In the field of fuzzy modeling research, the neuro-fuzzy is divided into two areas:

1. Linguistic fuzzy modeling focused on interpretability (mainly the Mamdani model).
2. Precise fuzzy modeling focused on accuracy [mainly the Takagi–Sugeno–Kang (TSK) model].

16.2.1 Comparison of Fuzzy Systems with Neural Networks

From the existing literature, it can be noted that neural networks and fuzzy systems have some things in common. If there does not exist any mathematical model of a given problem, then neural networks and fuzzy

Table 16-1 Comparison of neural and fuzzy processing

Neural processing	Fuzzy processing
Mathematical model not necessary	Mathematical model not necessary
Learning can be done from scratch	*A priori* knowledge is needed
There are several learning algorithms	Learning is not possible
Black-box behavior	Simple interpretation and implementation

systems can be used for solving that problem (e.g., pattern recognition, regression, or density estimation). This is the main reason for the growth of these intelligent computing techniques. Besides having individual advantages, they do have certain disadvantages that are overcome by combining both concepts.

When neural networks are concerned, if one problem is expressed by sufficient number of observed examples then only it can be used. These observations are used to train the *black box*. Though no prior knowledge about the problem is needed extracting comprehensible rules from a neural network's structure is very difficult.

A fuzzy system, on the other hand, does not need learning examples as prior knowledge; rather linguistic rules are required. Moreover, linguistic description of the input and output variables should be given. If the knowledge is incomplete, wrong or contradictory, then the fuzzy system must be tuned. This is a time-consuming process. Table 16.1 shows how combining both approaches brings out the advantages, leaving out the disadvantages.

16.2.2 Characteristics of Neuro-Fuzzy Hybrids

The general architecture of neuro-fuzzy hybrid system is as shown in Figure 16-1. A fuzzy system-based NFS is trained by means of a data-driven learning method derived from neural network theory. This heuristic causes local changes in the fundamental fuzzy system. At any stage of the learning process – before, during, or after – it can be represented as a set of fuzzy rules. For ensuring the semantic properties of the underlying fuzzy system, the learning procedure is constrained.

An NFS approximates an n-dimensional unknown function, partly represented by training examples. Thus fuzzy rules can be interpreted as vague prototypes of the training data. As shown in Figure 16-1, an NFS is

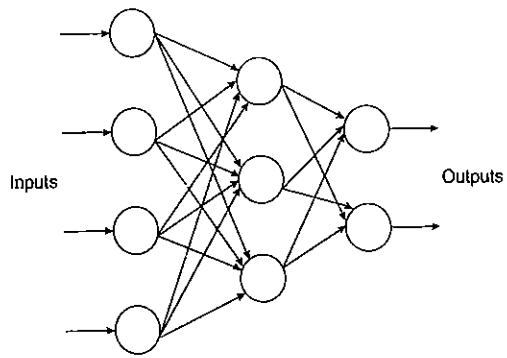


Figure 16-1 Architecture of neuro-fuzzy hybrid system.

given by a three-layer feedforward neural network model. It can also be observed that the first layer corresponds to the input variables, and the second and third layers correspond to the fuzzy rules and output variables, respectively. The fuzzy sets are converted to (fuzzy) connection weights.

NFS can also be considered as a system of fuzzy rules wherein the system can be initialized in the form of fuzzy rules based on the prior knowledge available. Some researchers use five layers – the fuzzy sets being encoded in the units of the second and the fourth layer, respectively. It is, however, also possible for these models to be transformed into three-layer architecture.

16.2.3 Classifications of Neuro-Fuzzy Hybrid Systems

NFSs can be classified into the following two systems:

1. Cooperative NFSs.
2. General neuro-fuzzy hybrid systems.

16.2.3.1 Cooperative Neural Fuzzy Systems

In this type of system, both artificial neural network (ANN) and fuzzy system work independently from each other. The ANN attempts to learn the parameters from the fuzzy system. Four different kinds of cooperative fuzzy neural networks are shown in Figure 16-2.

The FNN in Figure 16-2(A) learns fuzzy set from the given training data. This is done, usually, by fitting membership functions with a neural network; the fuzzy sets then being determined offline. This is followed by their utilization to form the fuzzy system by fuzzy rules that are given, and not learned. The NFS in Figure 16-2(B) determines, by a neural network, the fuzzy rules from the training data. Here again, the neural networks learn offline before the fuzzy system is initialized. The rule learning happens usually by clustering on self-organizing feature maps. There is also the possibility of applying fuzzy clustering methods to obtain rules.

For the neuro-fuzzy model shown in Figure 16-2(C), the parameters of membership function are learnt online, while the fuzzy system is applied. This means that, initially, fuzzy rules and membership functions must be defined beforehand. Also, in order to improve and guide the learning step, the error has to be measured. The model shown in Figure 16-2(D) determines the rule weights for all fuzzy rules by a neural network. A rule is determined by its rule weight–interpreted as the influence of a rule. They are then multiplied with the rule output.

16.2.3.2 General Neuro-Fuzzy Hybrid Systems (General NFHS)

General neuro-fuzzy hybrid systems (NFHS) resemble neural networks where a fuzzy system is interpreted as a neural network of special kind. The architecture of general NFHS gives it an advantage because there is no communication between fuzzy system and neural network. Figure 16-3 illustrates an NFHS. In this figure the rule base of a fuzzy system is assumed to be a neural network; the fuzzy sets are regarded as weights and the rules and the input and output variables as neurons. The choice to include or discard neurons can be made in the learning step. Also, the fuzzy knowledge base is represented by the neurons of the neural network; this overcomes the major drawbacks of both underlying systems.

Membership functions expressing the linguistic terms of the inference rules should be formulated for building a fuzzy controller. However, in fuzzy systems, no formal approach exists to define these functions. Any shape, such as Gaussian or triangular or bell shaped or trapezoidal, can be considered as a membership function with an arbitrary set of parameters. Thus for fuzzy systems, the optimization of these functions in terms of generalizing the data is very important; this problem can be solved by using neural networks.

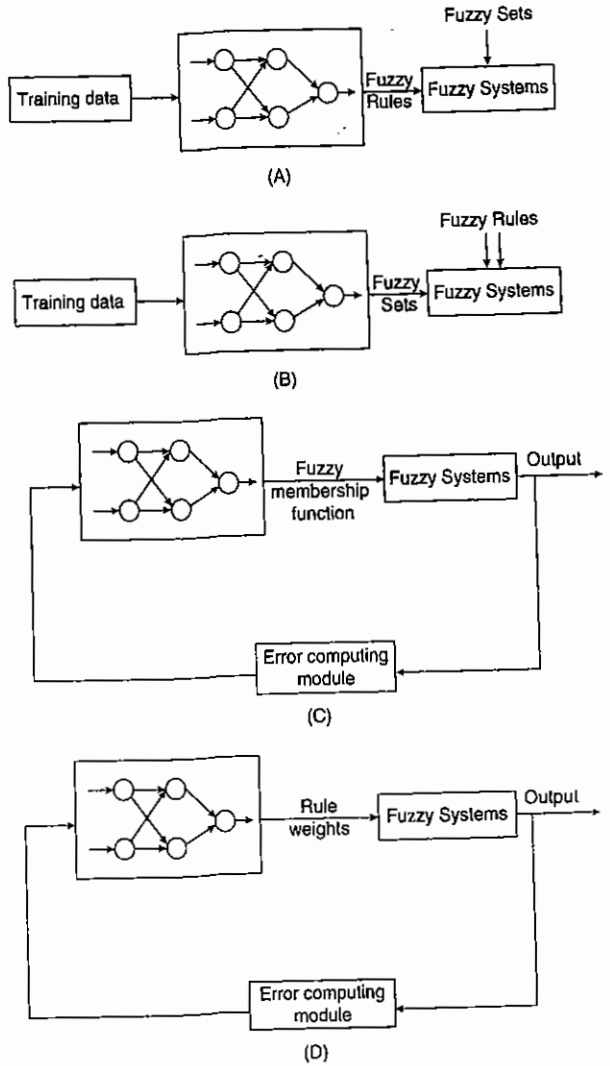


Figure 16-2 Cooperative neural fuzzy systems.

Using learning rules, the neural network must optimize the parameters by fixing a distinct shape of the membership functions; for example, triangular. But regardless of the shape of the membership functions, training data should also be available.

The neuro fuzzy hybrid systems can also be modeled in an another method. In this case, the training data is grouped into several clusters and each cluster is designed to represent a particular rule. These rules are defined by the crisp data points and are not defined linguistically. Hence a neural network, in this case, might

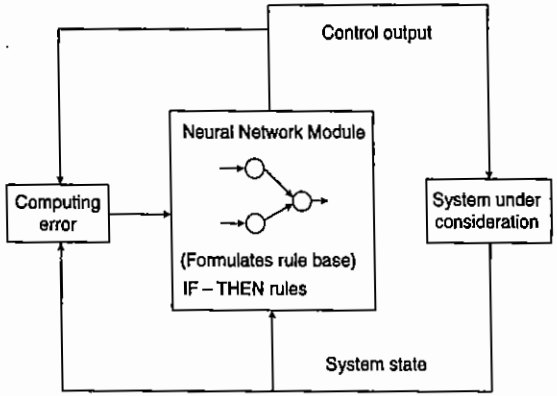


Figure 16-3 A general neuro-fuzzy hybrid system.

be applied to train the defined clusters. The testing can be carried out by presenting a random testing sample to the trained neural network. Each and every output unit will return a degree which extends to fit to the antecedent of rule.

16.2.4 Adaptive Neuro-Fuzzy Inference System (ANFIS) in MATLAB

The basic idea behind this neuro-adaptive learning technique is very simple. This technique provides a method for the fuzzy modeling procedure to learn information about a data set, in order to compute the membership function parameters that best allow the associated fuzzy inference system to track the given input/output data. This learning method works similarly to that of neural networks.

ANFIS Toolbox in MATLAB environment performs the membership function parameter adjustments. The function name used to activate this toolbox in `anfis`. ANFIS toolbox can be opened in MATLAB either at command line prompt or at Graphical User Interface. Based on the given input–output data set, ANFIS toolbox builds a Fuzzy Inference System whose membership functions are adjusted either using back propagation network training algorithm or Adaline network algorithm, which uses least mean square learning rule. This makes the fuzzy system to learn from the data they model.

The Fuzzy Logic Toolbox function that accomplishes this membership function parameter adjustment is called `anfis`. The acronym ANFIS derives its name from adaptive neuro-fuzzy inference system. The `anfis` function can be accessed either from the command line or through the ANFIS Editor GUI. Using a given input/output data set, the toolbox function `anfis` constructs a fuzzy inference system (FIS) whose membership function parameters are adjusted using either a back-propagation algorithm alone or in combination with a least squares type of method. This enables fuzzy systems to learn from the data they are modeling.

16.2.4.1 FIS Structure and Parameter Adjustment

A network-type structure similar to that of a neural network can be used to interpret the input/output. This structure maps inputs through input membership functions and associated parameters, and then through output membership functions and associated parameters to outputs. During the learning process, the parameters associated with the membership functions will change. A gradient vector facilitates the computation

(or adjustment) of these parameters, providing a measure of how well the fuzzy inference system models the input/output data for a given set of parameters. After obtaining the gradient vector, any of several optimization routines could be applied to adjust the parameters for reducing some error measure (defined usually by the sum of the squared difference between the actual and desired outputs). `anfis` makes use of either back-propagation or a combination of adaline and back-propagation, for membership function parameter estimation.

16.2.4.2 Constraints of ANFIS

When compared to the general fuzzy inference systems `anfis` is more complex. It is not available for all of the fuzzy inference system options and only supports Sugeno-type systems. Such systems have the following properties:

1. They should be the first- or zeroth-order Sugeno-type systems.
2. They should have a single output that is obtained using weighted average defuzzification. All output membership functions must be the same type and can be either linear or constant.
3. They do not share rules. The number of output membership functions must be equal to the number of rules.
4. They must have unity weight for each rule.

If FIS structure does not comply with these constraints then an error would occur. Also, all the customization options that basic fuzzy inference allows cannot be accepted by `anfis`. In simpler words, membership functions and defuzzification functions cannot be made according to one's choice, rather those provided should be used.

16.2.4.3 The ANFIS Editor GUI

To get started with the ANFIS Editor GUI, type `anfisedit` at the MATLAB command prompt. The GUI as in Figure 16-4 will appear on your screen.

From this GUI one can:

1. Load data (training, testing and checking) by selecting appropriate radio buttons in the Load Data portion of the GUI and then clicking Load Data. The loaded data is plotted on the plot region.
2. Generate an initial FIS model or load an initial FIS model using the options in the Generate FIS portion of the GUI.
3. View the FIS model structure once an initial FIS has been generated or loaded by clicking the Structure button.
4. Choose the FIS model parameter optimization method: back-propagation or a mixture of back-propagation and least squares (hybrid method).
5. Choose the number of training epochs and the training error tolerance.
6. Train the FIS model by clicking the Train Now button. This training adjusts the membership function parameters and plots the training (and/or checking data) error plot(s) in the plot region.
7. View the FIS model output versus the training, checking, or testing data output by clicking the Test Now button. This function plots the test data against the FIS output in the plot region.

One can also use the ANFIS Editor GUI menu bar to load an FIS training initialization, save your trained FIS, open a new Sugeno system, or open any of the other GUIs to interpret the trained FIS model.

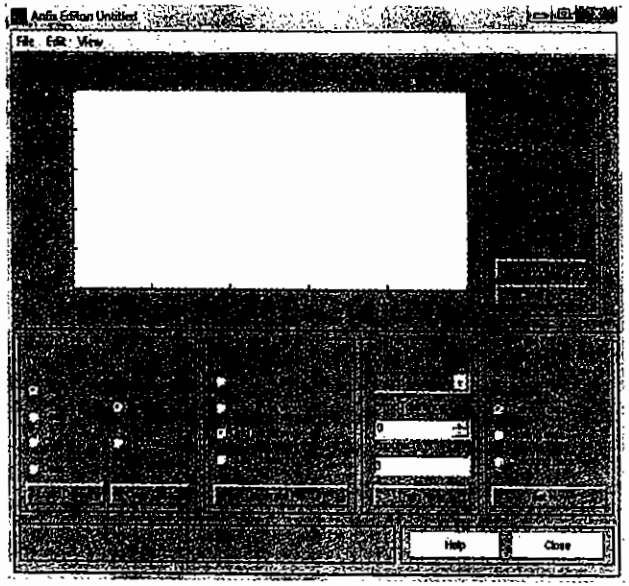


Figure 16-4 ANFIS Editor in MATLAB.

16.2.4.4 Data Formalities and the ANFIS Editor GUI

To start training an FIS using either `anfis` or the ANFIS Editor GUI, one needs to have a training data set that contains desired input/output data pairs of the target system to be modeled. In certain cases, optional testing data set may be available that can check the generalization capability of the resulting fuzzy inference system, and/or a checking data set that helps with model overfitting during the training. One can account for overfitting by testing the FIS trained on the training data against the checking data and choosing the membership function parameters to be those associated with the minimum checking error, if these errors indicate model overfitting. To determine this, their training error plots have to be examined fairly closely. Usually, these training and checking data sets are stored in separate files after being collected based on observations of the target system.

16.2.4.5 More on ANFIS Editor GUI

A minimum of two and maximum six arguments can be taken up by the command `anfis` whose general format is

```
[fismat1,trnError,ss,fismat2,chkError]=...
anfis(trnData,fismat,trnOpt,dispOpt,chkData,method);
```

Here `trnOpt` (training options), `dispOpt` (display options), `chkData` (checking data), and `method` (training method) are optional. All of the output arguments are also optional. In this section we will discuss the arguments and range components of the command line function `anfis` as well as the analogous functionality of the ANFIS Editor GUI. Only the training data set must exist before implementing `anfis` when the ANFIS Editor GUI is invoked using `anfisedit`. The step-size will be fixed when the adaptive NFS is trained using this GUI tool.

Training Data

Both `anfis` and ANFIS Editor GUI require the training data, `trnData`, as an argument. For the target system to be modeled each row of `trndata` is a desired input/output pair; a row starts with an input vector and is followed by an output value. So, the number of rows of `trndata` is equal to the number of training data pairs. Also, because there is only one output, the number of columns of `trndata` is one more than the number of inputs.

Input FIS Structure

The input FIS Structure, `fismat`, can be obtained from any of the following fuzzy editors:

1. The FIS Editor.
2. The Membership Function Editor.
3. The Rule Editor from the ANFIS Editor GUI (which allows a FIS structure to be loaded from a file or the MATLAB workspace).
4. The command line function, `genfis1` (for which one needs to give only numbers and types of membership functions).

The FIS structure contains both the model structure (specifying, e.g., number of rules in the FIS, the number of membership functions for each input, etc.) and the parameters (which specify the shapes of the membership functions).

For updating membership function parameters, `anfis` learning employs two methods:

1. Back-propagation for all parameters (a steepest descent method).
2. A hybrid method involving back-propagation for the parameters associated with the input membership functions and least-squares estimation for the parameters associated with the output membership functions.

This means that throughout the learning process, at least locally, the training error decreases. So, as the initial membership functions increasingly resemble the optimal ones, it becomes easier for the model parameter training to converge. In the setting up of these initial membership function parameters in the FIS structure, it may be helpful to have human expertise about the target system to be modeled.

Based on a fixed number of membership functions, the `genfis1` function produces a FIS structure. This structure invokes the so-called *curse of dimensionality* and causes excessive propagation of the number of rules when the number of inputs is moderately large (more than four or five). To enable some dimension reduction in the fuzzy inference system, the Fuzzy Logic Toolbox software provides a method – a FIS structure can be generated using the clustering algorithm discussed in Subtractive Clustering. To use this clustering algorithm, select the **Sub. Clustering** option in the **Generate FIS** portion of the ANFIS Editor GUI, before the FIS is generated. The data is partitioned by the subtractive clustering method into groups called clusters and generates a FIS with the minimum number of rules required to distinguish the fuzzy qualities associated with each of the clusters.

Training Options

One can choose a desired error tolerance and number of training epochs in the ANFIS Editor GUI tool. For the command line `anfis`, training option `trnOpt` is a vector specifying the stopping criteria and the step-size adaptation strategy:

1. `trnOpt(1)`: number of training epochs; default = 10

2. `trnOpt(2)`: error tolerance; default = 0
3. `trnOpt(3)`: initial step-size; default = 0.01
4. `trnOpt(4)`: step-size decrease rate; default = 0.9
5. `trnOpt(5)`: step-size increase rate; default = 1.1

The default value is taken if any element of `trnOpt` is missing or is an `NaN`. The training process stops if the designated epoch number is reached or the error goal is achieved, whichever comes first.

The step-size profile is usually a curve that increases initially, reaches a maximum, and then decreases for the remainder of the training. This ideal step-size profile can be achieved by adjusting the initial step-size and the increase and decrease rates (`trnOpt(3) - trnOpt(5)`). The default values are set up to cover a wide range of learning tasks. These step-size options may have to be modified, for any specific application, in order to optimize the training. There are, however, no user-specified step-size options for training the adaptive neuro-fuzzy inference system generated using the ANFIS Editor GUI.

Display Options

They apply only to the command line function `anfis`. The display options argument, `dispOpt`, is a vector of either 1s or 0s that specifies the information to be displayed (print in the MATLAB command window) before, during, and after the training process. To denote *print this option*, 1 is used and to denote *do not print this option*, 0 is used.

1. `dispOpt(1)`: display ANFIS information; default = 1
2. `dispOpt(2)`: display error (each epoch); default = 1
3. `dispOpt(3)`: display step-size (each epoch); default = 1
4. `dispOpt(4)`: display final results; default = 1

All available information is displayed in the default mode. If any element of `dispOpt` is missing or is `NaN`, the default value is used.

Method

To estimate membership function parameters, both the command line `anfis` and the ANFIS Editor GUI apply either a back-propagation form of the steepest descent method, or a combination of back-propagation and the least-squares method. The choices for this argument are `hybrid` or `backpropagation`. In the command line function, `anfis`, these method choices are designated by 1 and 0, respectively.

Output FIS Structure for Training Data

The output FIS structure corresponding to a minimal training error is `fismat1`. This is the FIS structure one uses to represent the fuzzy system when there is no checking data used for model cross-validation. Also, when the checking data option is not used, this data represents the FIS structure that is saved by the ANFIS Editor GUI. When one uses the checking data option, the output saved is that associated with the minimum checking error.

Training Error

This is the difference between the training data output value and the output of the fuzzy inference system corresponding to the same training data input value (the one associated with that training data output value.)

The root mean squared error (RMSE) of the training data set at each epoch is recorded by the training error `trnError`; and `fismat1` is the snapshot of the FIS structure when the training error measure is at its minimum. As the system is trained, the ANFIS Editor GUI plots the training error versus epochs curve.

Step-Size

With the ANFIS Editor GUI, one cannot control the step-size options. The step-size array `ss` records the step-size during the training, using the command line `anfis`. If one plots `ss`, one gets the step-size profile which serves as a reference for adjusting the initial step-size, and the corresponding decrease and increase rates. The guidelines followed for updating the step-size (`ss`) for the command line function `anfis` are:

1. If the error undergoes four consecutive reductions, increase the step-size by multiplying it by a constant (`ssinc`) greater than one.
2. If the error undergoes two consecutive combinations of one increase and one reduction, decrease the step-size by multiplying it by a constant (`ssdec`) less than one.

For the initial step-size, the default value is 0.01; for `ssinc` and `ssdec`, they are 1.1 and 0.9, respectively. All the default values can be changed via the training option for the command line `anfis`.

Checking Data

For testing the generalization capability of the fuzzy inference system at each epoch, the checking data, `chkData`, is used. The checking data and the training data have the same format and elements of the former are generally distinct from those of the latter.

For learning tasks for which the input number is large and/or the data itself is noisy, the checking data is important. A fuzzy inference system needs to track a given input/output data set well. The model structure used for `anfis` is fixed, which means that there is a tendency for the model to overfit the data on which it is trained, especially for a large number of training epochs. In case overfitting occurs, the fuzzy inference system may not respond well to other independent data sets, especially if they are corrupted by noise. In these situations, a validation or checking data set can be useful. To cross-validate the fuzzy inference model, this data set is used; cross-validation requires applying the checking data to the model and then seeing how well the model responds to this data.

The checking data is applied to the model at each training epoch, when the checking data option is used with `anfis` either via the command line or using the ANFIS Editor GUI. Once the command line `anfis` is invoked, the model parameters that correspond to the minimum checking error are returned via the output argument `fismat2`. The FIS membership function parameters computed using the ANFIS Editor GUI when both training and checking data are loaded, are associated with the training epoch that has a minimum checking error.

The assumptions made when using the minimum checking data error epoch to set the membership function parameters are:

1. The similarity between checking data and the training data means that the checking data error decreases as the training begins.
2. The checking data increases at some point in the training after the data overfitting occurs.

The resulting FIS may or may not be the one which is required to be used, depending on the behavior of the checking data error.

Output FIS Structure for Checking Data

The output FIS structure with the minimum checking error is the output of the command line `anfis, fismat2`. If checking data is used for cross-validation, this FIS structure is the one that should be used for further calculation.

Checking Error

This is the difference between the checking data output value and the output of the fuzzy inference system corresponding to the same checking data input value, which is the one associated with that checking data output value. The Root Mean Square Error (RMSE) is recorded for the checking data at each epoch, by the checking error `chkError`. The snapshot of the FIS structure when the checking error has its minimum value is `fismat2`. The checking error versus epochs curve is plotted by the ANFIS Editor GUI, as the system is trained.

16.3 Genetic Neuro-Hybrid Systems

A neuro-genetic hybrid or a genetic–neuro-hybrid system is one in which a neural network employs a genetic algorithm to optimize its structural parameters that define its architecture. In general, neural networks and genetic algorithm refers to two distinct methodologies. Neural networks learn and execute different tasks using several examples, classify phenomena, and model nonlinear relationships; that is neural networks solve problems by self-learning and self-organizing. On the other hand, genetic algorithms present themselves as a potential solution for the optimization of parameters of neural networks.

16.3.1 Properties of Genetic Neuro-Hybrid Systems

Certain properties of genetic neuro-hybrid systems are as follows:

1. The parameters of neural networks are encoded by genetic algorithms as a string of properties of the network, that is, chromosomes. A large population of chromosomes is generated, which represent the many possible parameter sets for the given neural network.
2. Genetic Algorithm – Neural Network, or GANN, has the ability to locate the neighborhood of the optimal solution quickly, compared to other conventional search strategies.

Figure 16-5 shows the block diagram for the genetic-neuro-hybrid systems. Their drawbacks are: the large amount of memory required for handling and manipulation of chromosomes for a given network; and also the question of scalability of this problem as the size of the networks become large.

16.3.2 Genetic Algorithm Based Back-Propagation Network (BPN)

BPN is a method of teaching multi-layer neural networks how to perform a given task. Here learning occurs during this training phase. The basic algorithm with architecture is discussed in Chapter 3 (Section 3.5) of this book in detail. The limitations of BPN are as follows:

1. BPN do not have the ability to recognize new patterns; they can recognize patterns similar to those they have learnt.
2. They must be sufficiently trained so that enough general features applicable to both seen and unseen instances can be extracted; there may be undesirable effects due to over training the network.

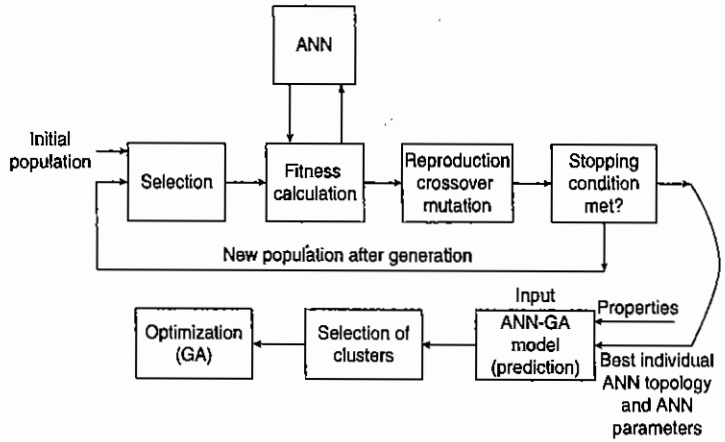


Figure 16-5 Block diagram of genetic–neuro hybrids.

Also, it may be noted that the BPN determines its weight based on gradient search technique and hence it may encounter a local minima problem. Though genetic algorithms do not guarantee to find global optimum solution, they are good in quickly finding good acceptable solutions. Thus, hybridization of BPN with genetic algorithm is expected to provide many advantages compared to what they alone can. The basic concepts and working of genetic algorithm are discussed in Chapter 15. However, before a genetic algorithm is executed,

1. A suitable coding for the problem has to be devised.
2. A fitness function has to be formulated.
3. Parents have to be selected for reproduction and then crossed over to generate offspring.

16.3.2.1 Coding

Assume a BPN configuration n-l-m where n is the number of neurons in the input layer, l is the number of neurons in the hidden layer and m is the number of output layer neurons. The number of weights to be determined is given by

$$(n + m)l$$

Each weight (which is a gene here) is a real number. Let d be the number of digits (gene length) in weight. Then a string S of decimal values having string length $(n + m)ld$ is randomly generated. It is a string that represents weight matrices of input-hidden and the hidden-output layers in a linear form arranged as row-major or column-major depending upon the style selected. Thereafter a population of p (which is the *population size*) chromosomes is randomly generated.

16.3.2.2 Weight Extraction

In order to determine the fitness values, weights are extracted from each chromosome. Let $a_1, a_2, \ldots, a_d, \ldots, a_L$ represent a chromosome and let $a_{pd+1}, a_{pd+2}, \ldots, a_{(p+1)d}$ represent pth gene $(p \geq 0)$ in the chromosomes.

The actual weight w_p is given by

$$w_p = \begin{cases} -\dfrac{a_{pd+2}10^{d-2} + a_{pd+3}10^{d-3} + \cdots + a_{(p+1)d}}{10^{d-2}} & \text{if } 0 \le a_{pd+1} < 5 \\ +\dfrac{a_{pd+2}10^{d-2} + a_{pd+3}10^{d-3} + \cdots + a_{(p+1)d}}{10^{d-2}} & \text{if } 5 \le a_{pd+1} \le 9 \end{cases}$$

16.3.2.3 Fitness Function

A fitness has to be formulated for each and every problem to be solved. Consider the matrix given by

$$\begin{Bmatrix} (x_{11}, x_{21}, x_{31}, \ldots, x_{n1}) & (y_{11}, y_{21}, y_{31}, \ldots, y_{n1}) \\ (x_{12}, x_{22}, x_{32}, \ldots, x_{n2}) & (y_{12}, y_{22}, y_{32}, \ldots, y_{n2}) \\ (x_{13}, x_{23}, x_{33}, \ldots, x_{n3}) & (y_{13}, y_{23}, y_{33}, \ldots, y_{n3}) \\ \vdots & \vdots \\ (x_{1m}, x_{2m}, x_{3m}, \ldots, x_{nm}) & (y_{1m}, y_{2m}, y_{3m}, \ldots, y_{nm}) \end{Bmatrix}$$

where X and Y are the inputs and targets, respectively. Compute initial population I_0 of size 'j'. Let $O_{10}, O_{20}, \ldots, O_{j0}$ represent 'j' chromosomes of the initial population I_0. Let the weights extracted for each of the chromosomes upto jth chromosome be $w_{10}, w_{20}, w_{30}, \ldots, w_{j0}$. For n number of inputs and m number of outputs, let the calculated output of the considered BPN be

$$\begin{Bmatrix} (c_{11}, c_{21}, c_{31}, \ldots, c_{n1}) \\ (c_{12}, c_{22}, c_{32}, \ldots, c_{n2}) \\ (c_{13}, c_{23}, c_{33}, \ldots, c_{n3}) \\ \vdots \\ (c_{1m}, c_{2m}, c_{3m}, \ldots, c_{nm}) \end{Bmatrix}$$

As a result, the error here is calculated by

$$ER_1 = (y_{11} - c_{11})^2 + (y_{21} - c_{21})^2 + (y_{31} - c_{31})^2 + \cdots + (y_{n1} - c_{n1})^2$$
$$ER_2 = (y_{12} - c_{12})^2 + (y_{22} - c_{22})^2 + (y_{32} - c_{32})^2 + \cdots + (y_{n2} - c_{n2})^2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$ER_m = (y_{1m} - c_{1m})^2 + (y_{2m} - c_{2m})^2 + (y_{3m} - c_{3m})^2 + \cdots + (y_{nm} - c_{nm})^2$$

The fitness function is further derived from this root mean square error given by

$$FF_n = \frac{1}{E_{rmse}}$$

The process has to be carried out for all the total number of chromosomes.

16.3.2.4 Reproduction of Offspring

In this process, before the parents produce the offspring with better fitness, the mating pool has to be formulated. This is accomplished by neglecting the chromosome with minimum fitness and replacing it with a chromosome having maximum fitness, In other words, the fittest individuals among the chromosomes will be given more chances to participate in the generations and the worst individuals will be eliminated. Once the mating pool is formulated, parent pairs are selected randomly and the chromosomes of respective pairs are combined using crossover technique to reproduce offspring. The selection operator is suitably used to select the best parent to participate in the reproduction process.

16.3.2.5 Convergence

The convergence for genetic algorithm is the number of generations with which the fitness value increases towards the global optimum. Convergence is the progression towards increasing uniformity. When about 95% of the individuals in the population share the same fitness value then we say that a population has converged.

16.3.3 Advantages of Neuro-Genetic Hybrids

The various advantages of neuro-genetic hybrid are as follows:

- GA performs optimization of neural network parameters with simplicity, ease of operation, minimal requirements and global perspective.
- GA helps to find out complex structure of ANN for given input and the output data set by using its learning rule as a fitness function.
- Hybrid approach ensembles a powerful model that could significantly improve the predictability of the system under construction.

The hybrid approach can be applied to several applications, which include: load forecasting, stock forecasting, cost optimization in textile industries, medical diagnosis, face recognition, multi-processor scheduling, job shop scheduling, and so on.

16.4 Genetic Fuzzy Hybrid and Fuzzy Genetic Hybrid Systems

Currently, several researches has been performed combining fuzzy logic and genetic algorithms (GAs), and there is an increasing interest in the integration of these two topics. The integration can be performed in the following two ways:

1. By the use of fuzzy logic based techniques for improving genetic algorithm behavior and modeling GA components. This is called *fuzzy genetic algorithms* (FGAs).
2. By the application of genetic algorithms in various optimization and search problems involving fuzzy systems.

An FGA is considered as a genetic algorithm that uses techniques or tools based on fuzzy logic to improve the GA behavior modeling. It may also be defined as an ordering sequence of instructions in which some of the instructions or algorithm components may be designed with tools based on fuzzy logic. For example, fuzzy operators and fuzzy connectives for designing genetic operators with different properties, fuzzy logic control systems for controlling the GA parameters according to some performance measures, stop criteria, representation tasks, etc.

GAs are utilized for solving different fuzzy optimization problems. For example, fuzzy flowshop scheduling problems, vehicle routing problems with fuzzy due-time, fuzzy optimal reliability design problems, fuzzy mixed integer programming applied to resource distribution, job-shop scheduling problem with fuzzy processing time, interactive fuzzy satisfying method for multi-objective 0-1, fuzzy optimization of distribution networks, etc.

16.4.1 Genetic Fuzzy Rule Based Systems (GFRBSs)

For modeling complex systems in which classical tools are unsuccessful, due to them being complex or imprecise, an important tool in the form of fuzzy rule based systems has been identified. In this regard, for mechanizing the definition of the knowledge base of a fuzzy controller GAs have proven to be a powerful tool, since adaptive control, learning, and self-organization may be considered in a lot of cases as optimization or search processes. Over the last few years their advantages have extended the use of GAs in the development of a wide range of approaches for designing fuzzy controllers. In particular, the application to the design, learning and tuning of knowledge bases has produced quite good results. In genetal these approaches can be termed as *Genetic Fuzzy Systems* (GFSs). Figure 16-6 shows a system where genetic design and fuzzy processing are the two fundamental constituents. Inside GFRBSs, it is possible to distinguish between either parameter optimization or rule generation processes, that is, adaptation and learning.

The main objectives of optimization in fuzzy rule based system are as follows:

1. The task of finding an appropriate knowledge base (KB) for a particular problem. This is equivalent to parameterizing the fuzzy KB (rules and membership functions).
2. To find those parameter values that are optimal with respect to the design criteria.

Considering a GFRBS, one has to decide which parts of the knowledge base (KB) are subject to optimization by the GA. The KB of a fuzzy system is the union of qualitatively different components and not a homogeneous

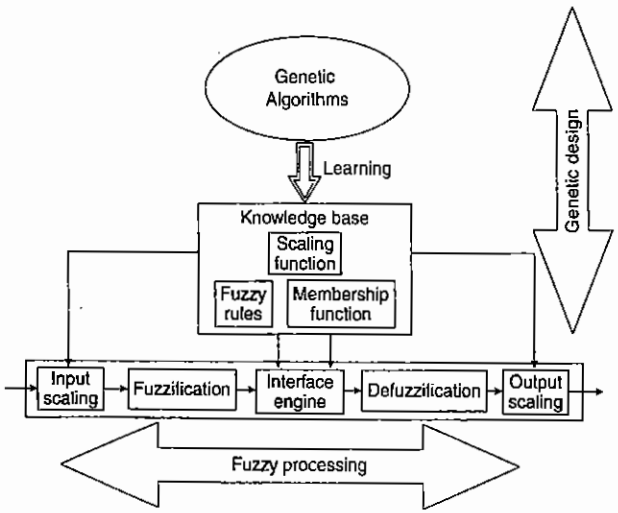


Figure 16-6 Block diagram of genetic fuzzy system.

Table 16-2 Tuning versus learning problems

Tuning	Learning Problems
It is concerned with optimization of an existing FRBS.	It constitutes an automated design method for fuzzy rule sets that start from scratch.
Tuning processes assume a predefined RB and have the objective to find a set of optimal parameters for the membership and/or the scaling functions, DB parameters.	Learning processes perform a more elaborated search in the space of possible RBs or whole KB and do not depend on a predefined set of rules.

structure. As an example, the KB of a descriptive Mamdani-type fuzzy system has two components: a rule base (RB) containing the collection of fuzzy rules and a data base (DB) containing the definitions of the scaling factors and the membership functions of the fuzzy sets associated with the linguistic labels.

In this phase, it is important to distinguish between tuning (alternatively, adaptation) and learning problems. See Table 16-2 for the differences.

16.4.1.1 Genetic Tuning Process

The task of tuning the scaling functions and fuzzy membership functions is important in FRBS design. The adoption of parameterized scaling functions and membership functions by the GA is based on the fitness function that specifies the design criteria quantitatively. The responsibility of finding a set of optimal parameters for the membership and/or the scaling functions rests with the tuning processes which assume a predefined rule base. The tuning process can be performed *a priori* also. This can be done if a subsequent process derives the RB once the DB has been obtained, that is, *a priori* genetic DB learning. Figure 16-7 illustrates the process of genetic tuning.

Tuning Scaling Functions

The universes of discourse where fuzzy membership functions are defined are normalized by scaling functions applied to the input and output variables of FRBSs. In case of linear scaling, the scaling functions are parameterized by a single scaling factor or either by specifying a lower and upper bound. On the other hand, in case of non-linear scaling, the scaling functions are parameterized by one or several contraction/dilation parameters. These parameters are adapted such that the scaled universe of discourse matches the underlying variable range.

Ideally, in these kinds of processes the approach is to adapt one to four parameters per variable: one when using a scaling factor, two for linear scaling, and three or four for non-linear scaling. This approach leads to a fixed length code as the number of variables is predefined as is the number of parameters required to code each scaling function.

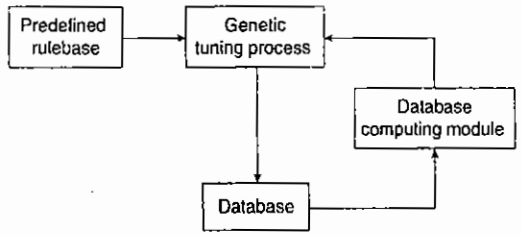


Figure 16-7 Process of tuning the DB.

Tuning Membership Functions

It can be noted that during the tuning of membership functions, an individual represents the entire DB. This is because its chromosome encodes the parameterized membership functions associated to the linguistic terms in every fuzzy partition considered by the fuzzy rule based system. Triangular (either isosceles or asymmetric), trapezoidal, or Gaussian functions are the most common shapes for the membership functions (in GFRBSs). The number of parameters per membership function can vary from one to four and each parameter can be either binary or real coded.

For FRBSs of the descriptive (using linguistic variables) or the approximate (using fuzzy variables) type, the structure of the chromosome is different. In the process of tuning the membership functions in a linguistic model, the entire fuzzy partitions are encoded into the chromosome and in order to maintain the global semantic in the RB, it is globally adapted. These approaches usually consider a predefined number of linguistic terms for each variable – with no requirement to be the same for each of them – which leads to a code of fixed length in what concerns membership functions. Despite this, it is possible to evolve the number of linguistic terms associated to a variable; simply define a maximum number (for the length of the code) and let some of the membership functions be located out of the range of the linguistic variable (which reduces the actual number of linguistic terms).

Descriptive fuzzy systems working with strong fuzzy partitions, is a particular case where the number of parameters to be coded is reduced. Here, the number of parameters to code is reduced to the ones defining the core regions of the fuzzy sets: the modal point for triangles and the extreme points of the core for trapezoidal shapes.

Tuning the membership functions of a model working with fuzzy variables (scatter partitions), on the other hand, is a particular instance of knowledge base learning. This is because, instead of referring to linguistic terms in the DB, the rules are defined completely by their own membership functions.

16.4.1.2 Genetic Learning of Rule Bases

As shown in Figure 16-8, genetic learning of rule bases assumes a predefined set of fuzzy membership functions in the DB to which the rules refer, by means of linguistic labels. As in the approximate approach adapting rules, it only applies to descriptive FRBSs, which is equivalent to modifying the membership functions. When considering a rule based system and focusing on learning rules, there are three main approaches that have been applied in the literature:

1. Pittsburgh approach.
2. Michigan approach.
3. Iterative rule learning approach.

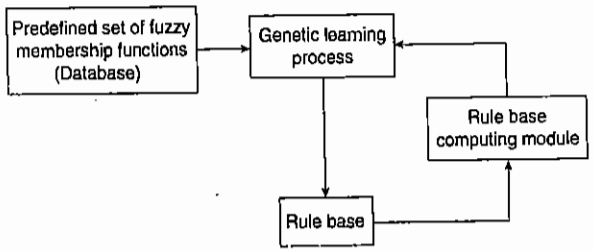


Figure 16-8 Genetic learning of rule base.

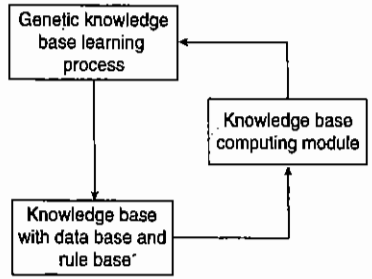


Figure 16-9 Genetic learning of the knowledge base.

The Pittsburgh approach is characterized by representing an entire rule set as a genetic code (chromosome), maintaining a population of candidate rule sets and using selection and genetic operators to produce new generations of rule sets. The Michigan approach considers a different model where the members of the population are individual rules and a rule set is represented by the entire population. In the third approach, the iterative one, chromosomes code individual rules, and a new rule is adapted and added to the rule set, in an iterative fashion, in every run of the genetic algorithm.

16.4.1.3 Genetic Learning of Knowledge Base

Genetic learning of a KB includes different genetic representations such as variable length chromosomes, multi-chromosome genomes and chromosomes encoding single rules instead of a whole KB as it deals with heterogeneous search spaces. As the complexity of the search space increases, the computational cost of the genetic search also grows. To combat this issue an option is to maintain a GFRBS that encodes individual rules rather than entire KB. In this manner one can maintain a flexible, complex rule space in which the search for a solution remains feasible and efficient. The three learning approaches as used in case of rule base can also be considered here: Michigan, Pittsburgh, and iterative rule learning approach. Figure 16-9 illustrates the genetic learning of KB.

16.4.2 Advantages of Genetic Fuzzy Hybrids

The hybridization between fuzzy systems and GAs in GFSs became an important research area during the last decade. GAs allow us to represent different kinds of structures, such as weights, features together with rule parameters, etc., allowing us to code multiple models of knowledge representation. This provides a wide variety of approaches where it is necessary to design specific genetic components for evolving a specific representation. Nowadays, it is a growing research area, where researchers need to reflect in order to advance towards strengths and distinctive features of the GFSs, providing useful advances in the fuzzy systems theory. Genetic algorithm efficiently optimizes the rules, membership functions, DB and KB of fuzzy systems. The methodology adopted is simple and the fittest individual is identified during the process.

16.5 Simplified Fuzzy ARTMAP

The basic concepts of Adaptive Resonance Theory Neural Networks are discussed in Chapter 5. Both the types of ART Networks, ART-1 and ART-2, are discussed in detail in Section 5.6.

Apart from these two ART networks, the other two maps are ARTMAP and Fuzzy ARTMAP. ARTMAP is also known as Predictive ART. It combines two slightly modified ART-1 or ART-2 units into a supervised learning structure. Here, the first unit takes the input data and the second unit takes the correct output data. Then minimum possible adjustment of the vigilance parameter in the first unit is made using the correct output data so that correct classification can be made.

The Fuzzy ARTMAP model has fuzzy-logic-based computations incorporated in the ARTMAP model. Fuzzy ARTMAP is neural network architecture for conducting supervised learning in a multidimensional setting. When Fuzzy ARTMAP is used on learning problem, it is trained till it correctly classifies all training data. This feature causes Fuzzy ARTMAP to "overfit" some data sets, especially those in which the underlying pattern has to overlap. To avoid the problem of "overfitting" one must allow for error in the training process.

16.5.1 Supervised ARTMAP System

Figure 16-10 shows the supervised ARTMAP system. Here, two ART modules are linked by an inter-ART module called the Map Field. The Map Field forms predictive associations between categories of the ART modules and realizes a match tracking rule. If ARTa and ARTb are disconnected, then each module would be of self-organize category, grouping their respective input sets. In supervised mode, the mappings are learned between input vectors *a* and *b*.

16.5.2 Comparison of ARTMAP with BPN

1. ARTMAP networks are self-stabilizing, while in BPNs the new information gradually washes away old information. A consequence of this is that a BPN has separate training and performance phases while ARTMAP systems perform and learn at the same time.

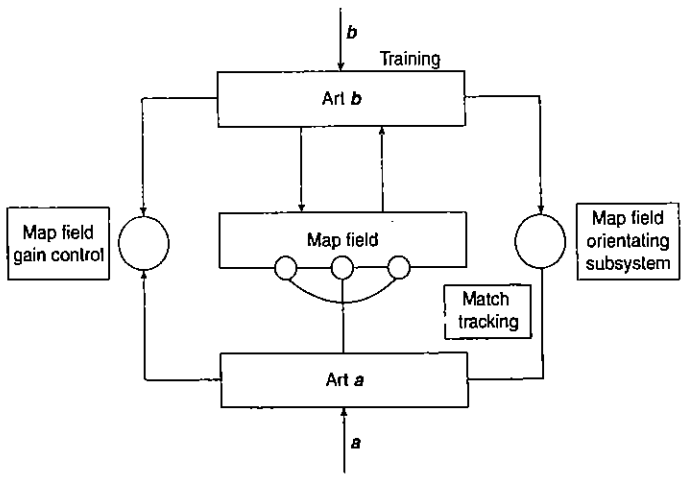


Figure 16-10 Supervised ARTMAP system.

2. ARTMAP networks are designed to work in real-time, while BPNs are typically designed to work off-line, at least during their training phase.
3. ARTMAP systems can learn both in a fast as well as inslow match configuration, while, the BPN can only learn in slow mismatch configuration. This means that an ARTMAP system learns, or adapts its weights, only when the input matches an established category, while BPNs learn when the input does not match an established category.
4. In BPNs there is always a chance of the system getting trapped in a local minimum while this is impossible for ART systems.

However, the systems based on ART modules learning may depend upon the ordering of the input patterns.

16.6 Summary

In this chapter, the various hybrids of individual neural networks, fuzzy logic and genetic algorithm have been discussed in detail. The advantages of each of these techniques are combined together to give a better solution to the problem under consideration. Each of these systems possesses certain limitations when they operate individually and these limitations are met by bringing out the advantages of combining these systems. The hybrid systems are found to provide better solution for complex problems and the advent of hybrid systems makes it applicable to be applied in various application domains.

16.7 Solved Problems using MATLAB

1. Write a MATLAB program to adapt the given input to sine wave form using adaptive neuro-fuzzy hybrid technique.

Source code

```
%program to adapt the given input to sine wave form using adaptive neuro
%fuzzy hybrid technique.

clc;
clear all;
close a...;

%input data
x=[0:0.1:10];
disp('The input data given x is : ');
disp(x);

%target data
t=sin(x);
disp('The target data given t is : ');
disp(t);

%training data
trndata=[x;t];
mfs=7;
```

```
epochs=570;

%creating fuzzy inference engine
fis=genfis1(trndata,mfs);
plotfis(fis);
figure
r=showrule(fis);

%creating adaptive neuro fuzzy inference engine
nfis=anfis(trndata,fis,epochs);
r1=showrule(nfis);

%evaluating anfis with given input
y=evalfis(x,nfis);
disp('The output data from anfis : ');
disp(y);

%calculating error rate
e=y-t;
plot(e);
title('Error rate')
figure

%ploting given training data and anfis output
plot(x,t,'o',x,y,'*');
title('Training data vs Output data');
legend('Training data','ANFIS Output');
```

Output

```
The input data given x is:
         0
    0.3000
    0.6000
    0.9000
    1.2000
    1.5000
    1.8000
    2.1000
    2.4000
    2.7000
    3.0000
    3.3000
    3.6000
    3.9000
    4.2000
    4.5000
    4.8000
    5.1000
    5.4000
    5.7000
```



```
    6.0000
    6.3000
    6.6000
    6.9000
    7.2000
    7.5000
    7.8000
    8.1000
    8.4000
    8.7000
    9.0000
    9.3000
    9.6000
    9.9000
   10.2000
   10.5000
   10.8000
   11.1000
   11.4000
   11.7000
   12.0000
   12.3000
   12.6000
   12.9000
   13.2000
   13.5000
   13.8000
   14.1000
   14.4000
   14.7000
   15.0000
   15.3000
   15.6000
   15.9000
   16.2000
   16.5000
   16.8000
   17.1000
   17.4000
   17.7000
   18.0000
   18.3000
   18.6000
   18.9000
   19.2000
```

```
   19.5000
   19.8000

The target data given ... :
         0
    0.2955
    0.5646
    0.7833
    0.9320
    0.9975
    0.9738
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
    0.0336
[illegible]
    0.5921
    0.8038
```



```
    0.9437
    0.9993
    0.9657
    0.8457
    0.6503
    0.3967
    0.1078
   -0.1909
   -0.4724
   -0.7118
   -0.8876
   -0.9841
   -0.9927
   -0.9126
   -0.7510
   -0.5223
   -0.2470
    0.0504
    0.3433
    0.6055
    0.8137

ANFIS info:
    Number of nodes: 32
    Number of linear parameters: 14
    Number of nonlinear parameters: 21
    Total number of parameters: 35
    Number of training data pairs: 67
    Number of checking data pairs: 0
    Number of fuzzy rules: 7

Start training ANFIS ...
   1    0.0517485
   2    0.0513228
   3    0.0508992
   4    0.0504776
   5    0.0500581

Step size increases to 0.011000 after epoch 5.
   6    0.0496406
   7    0.0491837
   8    0.0487291
   .
   .
   .
   .
   .
 568    0.00105594
```

```
Step size increases to 0.000450 after epoch 568.
 569   0.0010555
 570   0.00105516

Designated epoch number reached --> ANFIS training completed at epoch 570.

The output data from anfis:
   -0.0014
    0.2981
    0.5647
    0.7817
    0.9314
    0.9984
    0.9747
    0.8629
    0.6746
    0.4271
    0.1416
   -0.1571
   -0.4425
   -0.6884
   -0.8720
   -0.9772
   -0.9955
   -0.9260
   -0.7735
   -0.5509
   -0.2788
    0.0174
    0.3112
    0.5777
    0.7935
    0.9387
    0.9991
    0.9697
    0.8540
    0.6627
    0.4122
    0.1247
   -0.1741
   -0.4574
   -0.7000
   -0.8801
   -0.9812
   -0.9941
   -0.9189
   -0.7623
   -0.5371
   -0.2629
    0.0346
```



```
    0.3277
    0.5908
    0.8024
    0.9442
    1.0014
    0.9667
    0.8443
    0.6484
    0.3969
    0.1093
   -0.1900
   -0.4731
   -0.7130
   -0.8879
   -0.9833
   -0.9916
   -0.9125
   -0.7521
   -0.5232
   -0.2457
    0.0526
    0.3426
    0.6015
    0.8163

<end of program>
```

Figure 16-11 illustrates the ANFIS system module; figure 16-12 the error rate; and Figure 16-13 the performance of training data and output data. Thus it can be noted from Figure 16-13, that `anfis` has adapted the given input to sine wave form.

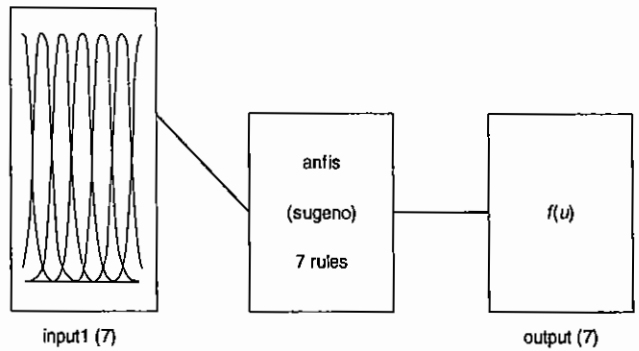


Figure 16-11 ANFIS system module.

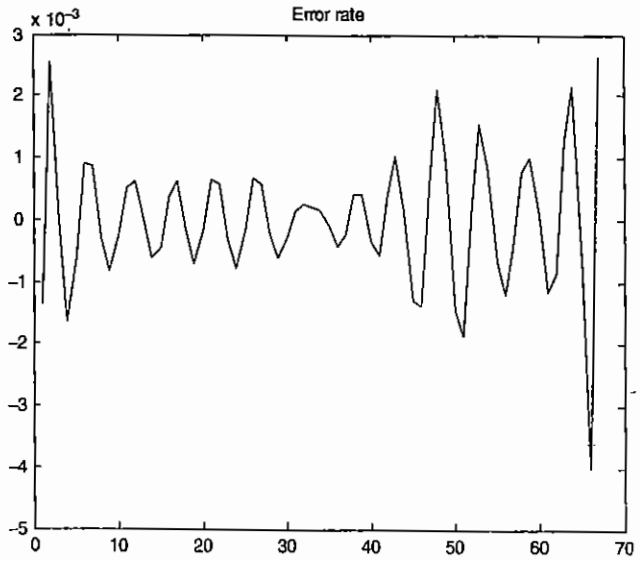


Figure 16-12 Error rate.

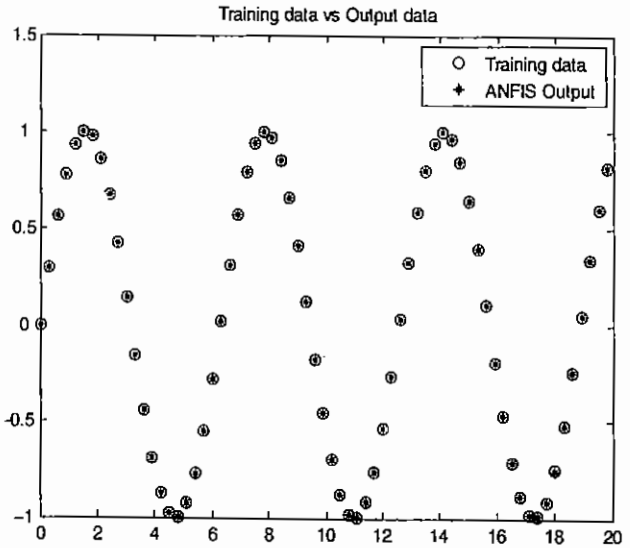


Figure 16-13 Performance of training data and output data.

2. Write a MATLAB program to recognize the given input of alphabets to its respective outputs using adaptive neuro-fuzzy hybrid technique.

Source code

```
%program to recognize the given input of alphabets to its respective
%outputs using adaptive neuro fuzzy hybrid technique.

clc;
clear all;
close all;

%input data
x=[0,1,0,0;1,0,1,1;1,1,1,2;1,0,1,3;1,0,1,4;
   1,1,0,5;1,0,1,6;1,1,0,7;1,0,1,8;1,1,0,9;
   0,1,1,10;1,0,0,11;1,0,0,12;1,0,0,13;0,1,1,14;
   1,1,0,15;1,0,1,16;1,0,1,17;1,0,1,18;1,1,0,19;
   1,1,1,20;1,0,0,21;1,1,0,22;1,0,0,23;1,1,1,24;]

%target data
t=[0;0;0;0;0;
   1;1;1;1;1;
   2;2;2;2;2;
   3;3;3;3;3;
   4;4;4;4;4;]

%training data
trndata=[x,t];
mfs=3;
epochs=400;

%creating fuzzy inference engine
fis=genfis1(trndata,mfs);
plotmf(fis,'input',1);
r=showrule(fis);

%creating adaptive neuro fuzzy inference engine
nfis = anfis(trndata,fis,epochs);
surfview(nfis);
figure
r1=showrule(nfis);

%evaluating anfis with given input
y=evalfis(x,nfis);
disp('The output data from anfis:');
disp(y);

%calculating error rate
e=y-t;
plot(e);
title('Error rate');
figure
```

```
%ploting given training data and anfis output
plot(x,t,'ro',x,y,'kx');
title('Training data vs Output data');
legend('Training data','ANFIS Output','location','North');
```

Output

```
x =
     0     1     0     0
     1     0     1     1
     1     1     1     2
     1     0     1     3
     1     0     1     4
     1     1     0     5
     1     0     1     6
     1     1     0     7
     1     0     1     8
     1     1     0     9
     0     1     1    10
     1     0     0    11
     1     0     0    12
     1     0     0    13
     0     1     1    14
     1     1     0    15
     1     0     1    16
     1     0     1    17
     1     0     1    18
     1     1     0    19
     1     1     1    20
     1     0     0    21
     1     1     0    22
     1     0     0    23
     1     1     1    24

t =
     0
     0
     0
     0
     0
     1
     1
     1
     1
     1
     2
     2
     2
     2
     2
     3
```



```
     3
     3
     3
     3
     4
     4
     4
     4
     4

ANFIS info:
    Number of nodes: 193
    Number of linear parameters: 405
    Number of nonlinear parameters: 36
    Total number of parameters: 441
    Number of training data pairs: 25
    Number of checking data pairs: 0
    Number of fuzzy rules: 81

Start training ANFIS ...
   1    0.08918
   2    0.0889038
   3    0.0886229
   4    0.0883371
   5    0.0880464

Step size increases to 0.011000 after epoch 5.
   6    0.0877506
   7    0.0874193
   .
   .
   .
   .
 398    0.00102161
 399    0.00102102
 400    0.0010191

Step size increases to 0.003347 after epoch 400.

Designated epoch number reached --> ANFIS training completed at epoch 400.

The output data from anfis:
   -0.0000
    0.0009
    0.0000
   -0.0031
    0.0024
    1.0000
    0.9997
```

```
    1.0000
    1.0002
    1.0001
    2.0000
    2.0001
    1.9998
    2.0001
    2.0000
    2.9999
    2.9982
    3.0022
    2.9994
    3.0001
    4.0000
    4.0000
    3.9999
    4.0000
    4.0000

<end of program>
```

Figure 16-14 shows the degree of membership. Figure 16-15 illustrates the surface view of the given system; Figure 16-16 the error rate; and Figure 16-17 the performance of training data with output data.

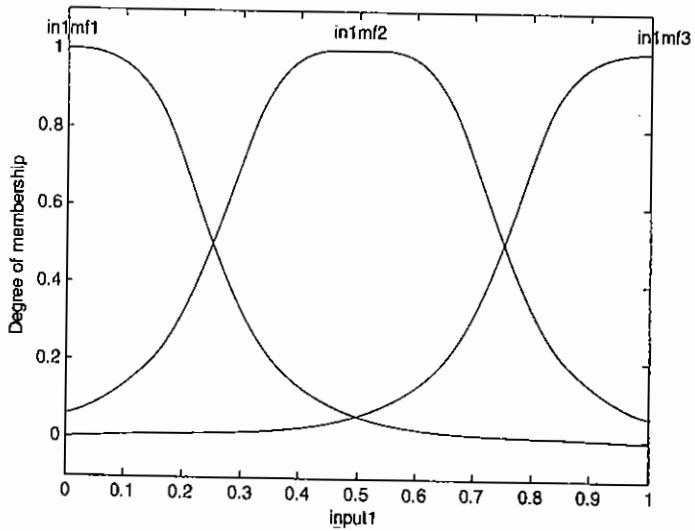


Figure 16-14 Degree of membership.

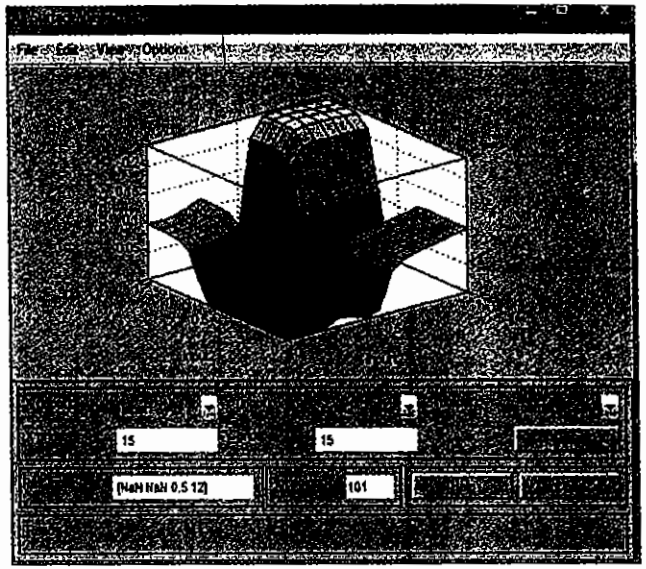


Figure 16-15 Surface view of the given system.

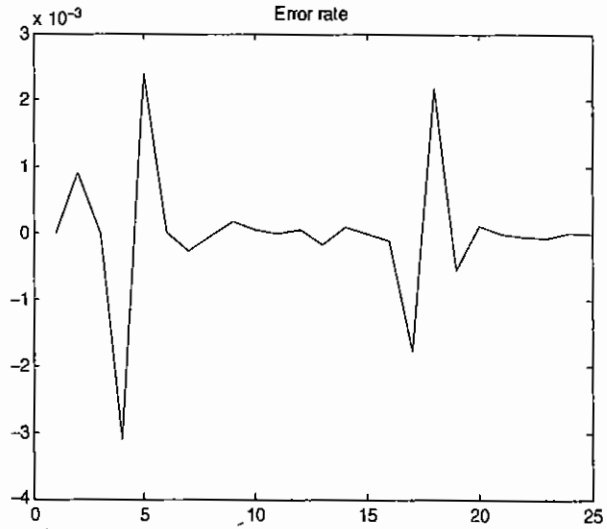


Figure 16-16 Error rate.

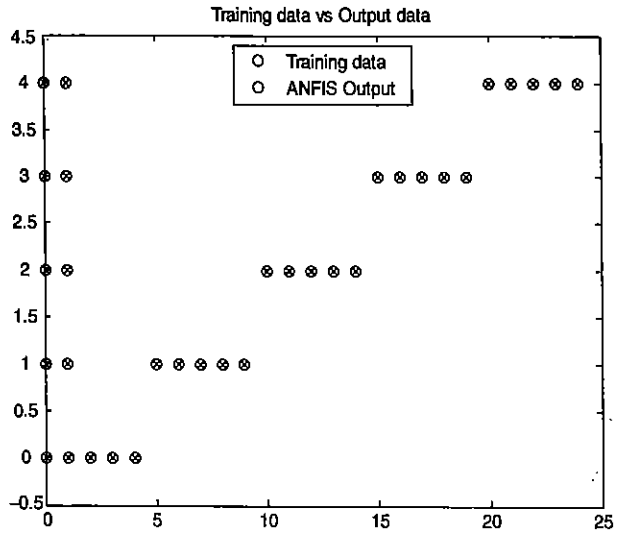


Figure 16-17 Performance of training data with output data.

3. Write a MATLAB program to train the given truth table using adaptive neuro-fuzzy hybrid technique.

Source code

```
%Program to train the given truth table using adaptive neuro fuzzy
%hybrid technique.

clc;
clear all;
close all;

%input data
x=[0,0,0;0,0,1;0,1,0;0,1,1;1,0,0;1,0,1;1,1,0;1,1,1;]

%target data
t=[0;0;0;1;0;1;1;1;]

%training data
trndata=[x,t];
mfs=3;
mfType = 'gbellmf';
epochs=49;

%creating fuzzy inference engine
fis=genfis1(trndata,mfs,mfType);
plotfis(fis);
title('The created fuzzy logic');
```



```
figure
plotmf(fis,'input',1);
title('The membership function of the fuzzy');
surfview(fis);
figure
ruleview(fis);
r=showrule(fis);

%creating adaptive neuro fuzzy inference engine
nfis = anfis(trndata,fis,epochs);
plotfis(nfis);
title('The created anfis');
figure
plotmf(nfis,'input',1);
title('The membership function of the anfis');
surfview(nfis);
figure
ruleview(nfis);
r1=showrule(nfis);

%evaluating anfis with given input
y=evalfis(x,nfis);
disp('The output data from anfis:');
disp(y);

%calculating error rate
e=y-t;
plot(e);
title('Error rate');
figure

%ploting given training data and anfis output
plot(x,t,'o',x,y,'*');
title('Training data vs Output data');
legend('Training data','ANFIS Output');
```

Output

```
x =
     0     0     0
     0     0     1
     0     1     0
     0     1     1
     1     0     0
     1     0     1
     1     1     0
     1     1     1

t =
     0
     0
```

```
      0
      1
      0
      1
      1
      1

ANFIS info:
    Number of nodes: 78
    Number of linear parameters: 108
    Number of nonlinear parameters: 27
    Total number of parameters: 135
    Number of training data pairs: 8
    Number of checking data pairs: 0
    Number of fuzzy rules: 27

Start training ANFIS ...
    1   3.13863e-007
    2   3.0492e-007
    3   2.97841e-007
    4   2.90245e-007
    5   2.84305e-007

Step size increases to 0.011000 after epoch 5
    6   2.78077e-007
    .
    .
    .
    .
   47   2.22756e-007
   48   2.22468e-007
   49   2.22431e-007

Step size increases to 0.015627 after epoch 49.

Designated epoch number reached --> ANFIS training completed at epoch 49.

The output data from anfis:
   -0.0000
    0.0000
    0.0000
    1.0000
    0.0000
    1.0000
    1.0000
    1.0000

<end of program>
```

Figure 16-18 shows the ANFIS module for the given system with specified inputs. Figure 16-19 illustrates the rule viewer for the ANFIS module. Figure 16-20 gives the error rate. Figure 16-21 shows the performance of Training data and output data.

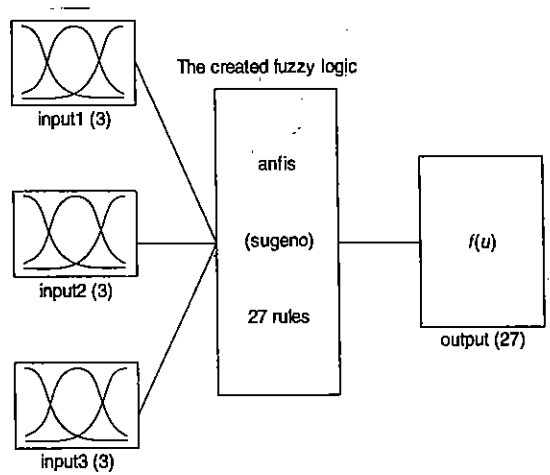


Figure 16-18 ANFIS module for the given system with specified inputs.

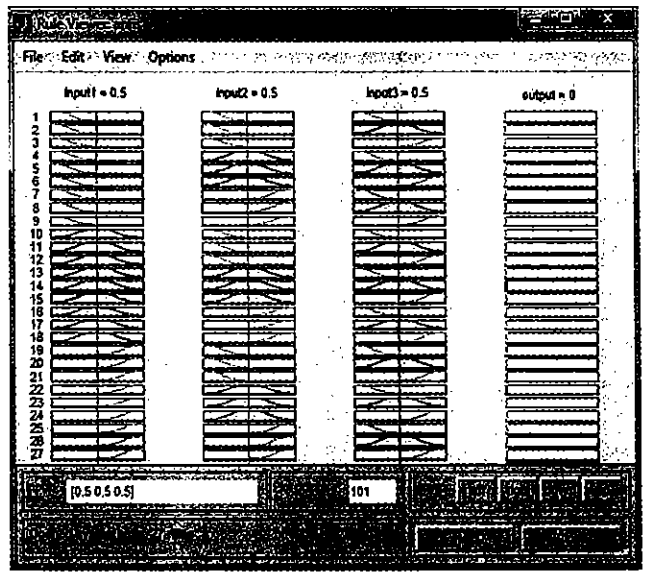


Figure 16-19 Rule viewer for the ANFIS module.

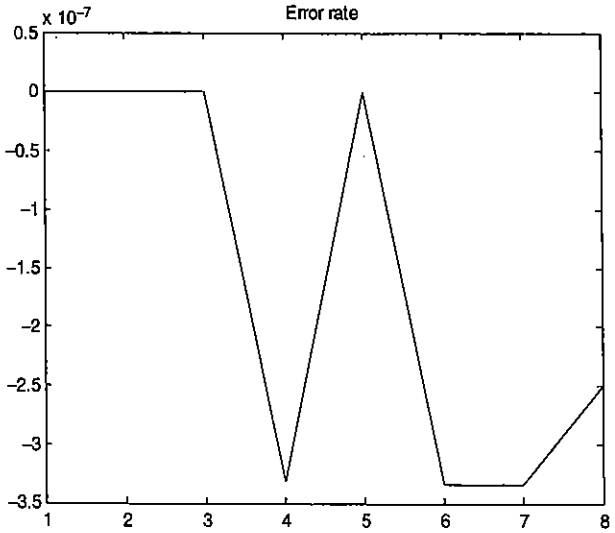


Figure 16-20 Error rate.

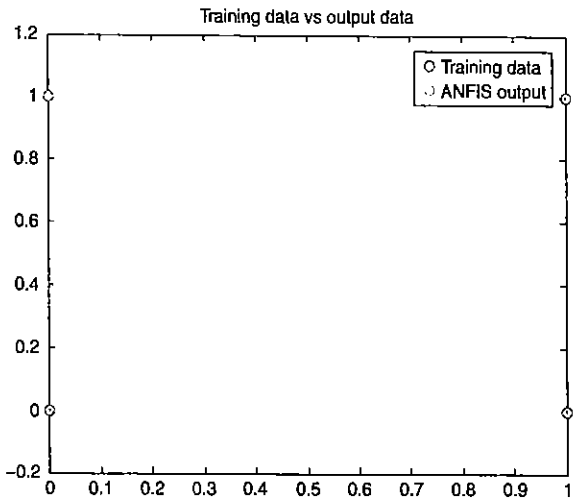


Figure 16-21 Performance of training data and output data.

4. Write a MATLAB program to optimize the neural network parameters for the given truth table using genetic algorithm.

Source code

```
%Program to optimize the neural network parameters from given truth table
%using genetic algorithm

clc;
clear all;
close all;

%input data
p = [0 0 1 1;0 1 0 1];

%target data
t = [-1 1 -1 1];

%creating a feedforeward neural network
net=newff(minmax(p),[2,1]);

%creating two layer net with two neurons in hidden(1) layer
net.inputs{1}.size = 2;
net.numLayers = 2;

%initializing network
net = init(net);
net.initFcn = 'initlay';

%initializing weights and bias
net.layers{1}.initFcn = 'initwb';
net.layers{2}.initFcn = 'initwb';

%Assigning weights and bias from function 'gawbinit'
net.inputWeights{1,1}.initFcn = 'gawbinit';
net.layerWeights{2,1}.initFcn = 'gawbinit';
net.biases{1}.initFcn='gawbinit';
net.biases{2}.initFcn='gawbinit';

%configuring training parameters
net.trainParam.lr = 0.05;        %learning rate
net.trainParam.min_grad=0e-10;   %min. gradient
net.trainParam.epochs = 60;      %No. of iterations
%Training neural net
net=train(net,p,t);

%simulating the net with given input
y = sim(net,p);
disp('The output of the net is :');
disp(y);
%ploting given training data and anfis output
plot(p,t,'o',p,y,'*');
title('Training data vs Output data');
```

```
%calculating error rate
e= gsubtract(t,y);  % e=t-y
disp('The error(t-y) of the net is :');
disp(e);

%program to calculate weights and bias of the net

function out1 = gawbinit(in1,in2,in3,in4,in5,~)

%%=============================================================================

%Implementing genetic algorithm

%configuring ga arguments
A = []; b = [];             %linear constraints
Aeq = []; beq = [];         %linear inequalities
lb = [-2 -2 -2 -2 -2 -2];   %lower bound
ub = [2 2 2 2 2 2];         %upper bound

%ploting ga parameters
options = gaoptimset('PlotFcns',{@gaplotscorediversity,@gaplotbestf});

%creating a multi objective genetic algorithm

%number of variables , for 2 layer 1 output 5 neuron net there are
%6 weights and 3 biases(6+3=9)
nvars=9;
[X,fval,exitFlag,Output]=gamultiobj(@fitnesfun,nvars,A,b,Aeq,beq,lb,
                                    ub,options);
figure

%displaying the ga output parameters
disp(X);
fprintf('The number of generations was : %d\n', Output.generations);
fprintf('The number of function evaluations was : %d\n', Output.funccount);
fprintf('The best function value found was : %g\n', fval);

%%=============================================================================

%Assigning the values of weights and bias respectively

%getting information of the net
persistent INFO;
if isempty(INFO), INFO = nnfcnWeightInit(mfilename,'Random Symmetric',
7.0,... true,true,true, true,true,true,true, true);  end
if ischar(in1)
switch lower(in1)
   case 'info', out1 = INFO;

   %configuring function
   case 'configure'
         out1 = struct;
```



```
   case 'initialize'

   %selecting input weights , layer weights and bias separately
   switch(upper(in3))
     case {'IW'}    %for input weights
       if INFO.initInputWeight
       if in2.inputConnect(in4,in5)
       x=X;  %Assigning ga output 'X' to input weights

%Taking first 4 ga outputs to create input weight matrix 'wi'
        wi(1,1)=x(1,1);
        wi(1,2)=x(1,2);
        wi(2,1)=x(1,3);
        wi(2,2)=x(1,4);
        disp(wi);
        out1 = wi;%Returning input layer matrix
     else
        out1 = [];
     end
   else
     nerr.throw([upper(mfilename) ' does not initialize input weights.']);
end
       case {'LW'}    %for layer weights
         if INFO.initLayerWeight
         if in2.layerConnect(in4,in5)
         x=X; %Assigning ga output 'X' to layer weights

%Taking 7th and 8th ga outputs to create layer weight matrix 'wl'
         wl(1,1)=x(1,7);
         wl(1,2)=x(1,8);
         disp(wl);
         out1 = wl;%Returning layer weight matrix
     else
         out1 = [];
    end
  else
    nnerr.throw([upper(mfilename) ' does not initialize input weights.']);
end
      case {'B'}     %for bias
         if INFO.initBias
         if in2.biasConnect(in4)
         x=X;  %Assigning ga output 'X' to bias

%Taking 5th,6th and 9th ga outputs to create bias matrix 'bl'
         bl(1)=x(1,5);
         bl(2)=x(1,6);
         bl(3)=x(1,9);
         disp(bl);
         out1 = bl;
      else
```

```
        out1 = [];%Returning bias matrix
      end
    else
      nnerr.throw([upper(mfilename) ' does not initialize biases.']);
  end
  otherwise,
      nnerr.throw('Unrecognized value type.');
  end
  end
  end
  end

  %Creating fitness function for genetic algorithm
  function z = fitnesfun(e)
  %The error(t-y) for all 4 i/o pairs are summed to get overall error

  %For 4 input target pairs the overall error is divided by 4 to get average

  %error value (1/4=0.25)
  z=0.25*sum(abs(e));
  end
```

Output

```
Optimization terminated: average change in the spread of Pareto solutions
less than options.TolFun.
  Columns 1 through 7
    0.0280    0.0041    0.0112    0.0069    0.0050    0.0062    0.0075
  Columns 8 through 9
    0.0018    0.0003
The number of generations was : 102
The number of function evaluations was : 13906
The best function value found was : 0.0177734
Optimization terminated: average change in the spread of Pareto solutions
less than options.TolFun.
  Columns 1 through 7
    0.0012    0.0020    0.0096    0.0014    0.0018    0.0044    0.0084
  Columns 8 through 9
    0.0084    0.0025
The number of generations was : 102
The number of function evaluations was : 13906
The best function value found was : 0.00988699
The output of the net is :
   -1.0000    1.0000   -1.0000    1.0000
The error(t-y) of the net is :
   1.0e-011 *
    -0.3097    0.2645   -0.2735    0.3006
```

Figure 16-22 shows the plot of the generations versus fitness value and histogram. Figure 16-23 illustrates the Neural Network Training Tool for the given input and output pairs. Figure 16-24 shows the neural network training performance. Neural network training state is shown in Figure 16-25. Figure 16-26 displays the performance of training data versus output data.

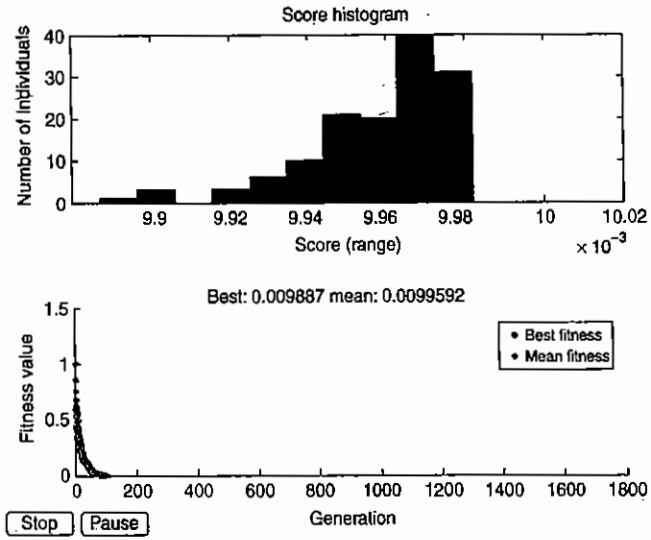


Figure 16-22 Plot of the generations versus fitness value and histogram.

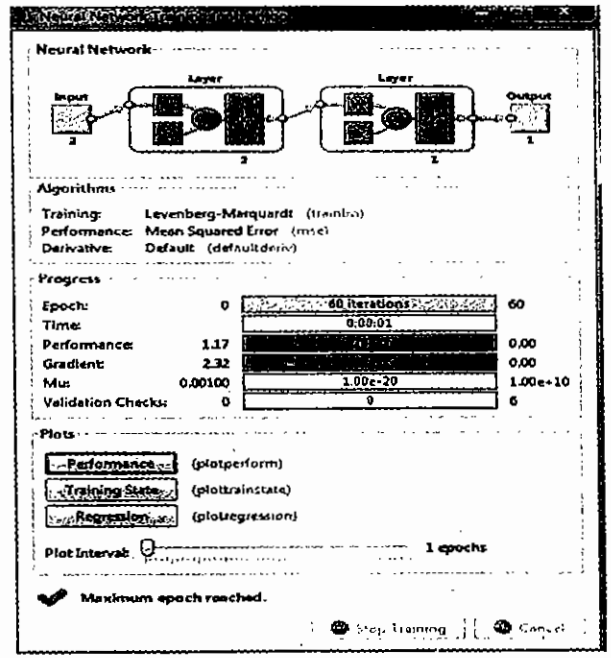


Figure 16-23 Neural Network Training Tool for the given input and output pairs.

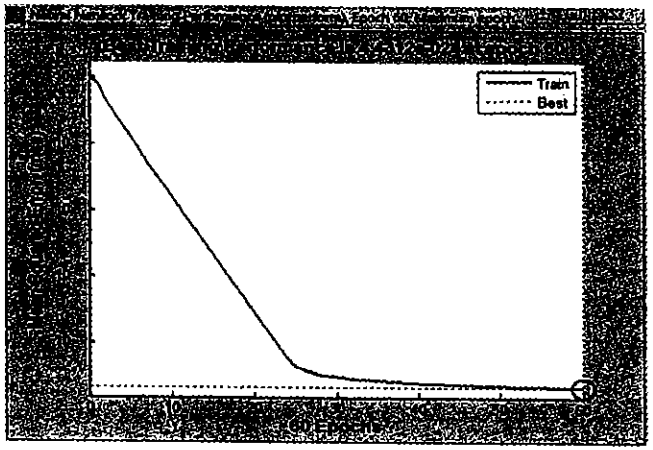


Figure 16-24 Neural network training performance.

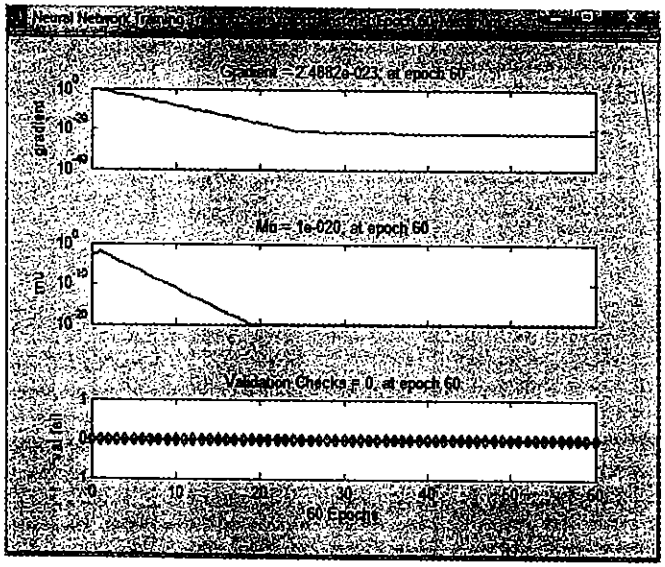


Figure 16-25 Neural network training state.

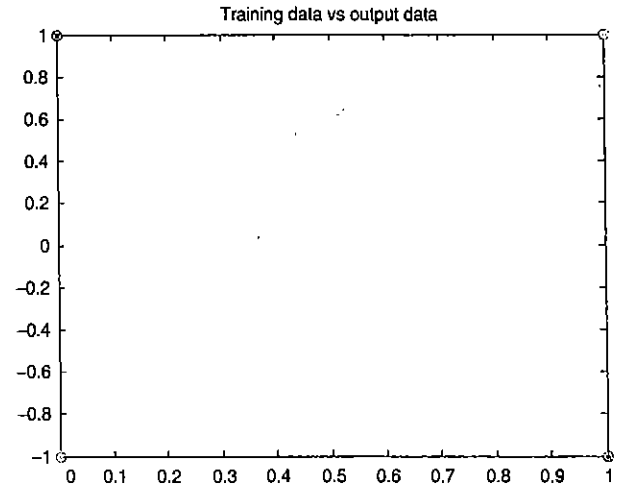


Figure 16-26 Performance of training data versus output data.

16.8 Review Questions

1. State the limitations of neural networks and fuzzy systems when operated individually.
2. List the various types of hybrid systems.
3. Mention the characteristics and properties of neuro-fuzzy hybrid systems.
4. What are the classifications of neuro-fuzzy hybrid systems? Explain in detail any one of the neuro-fuzzy hybrid systems.
5. Give details on the various applications of neuro-fuzzy hybrid systems.
6. How are genetic algorithms utilized for optimizing the weights in neural network architecture?
7. Explain in detail the concepts of fuzzy genetic hybrid systems.
8. Differentiate: ARTMAP and Fuzzy ARTMAP, Fuzzy ARTMAP and back-propagation neural networks.
9. Write notes on the supervised fuzzy ARTMAPs.
10. Give description on the operation of ANFIS Editor in MATLAB.

16.9 Exercise Problems

1. Write a MATLAB program to train NAND gate with binary inputs and targets (two input–one output) using adaptive neuro-fuzzy hybrid technique.
2. Consider some alphabets of your own and recognize the assumed characters using ANFIS Editor module in MATLAB.

3. Perform Problem 2 for any assumed numeral characters.
4. Design a genetic algorithm to optimize the weights of a neural network model while training an OR gate with 2 bipolar inputs and 1 bipolar targets.
5. Write a MATLAB M-file program for the working of washing machine using fuzzy genetic hybrids.

17 Applications of Soft Computing

Learning Objectives

- Deals with the various applications of soft computing in detail.
- Discuss how SAR image are solved using neural network approach.
- Gives a methology for solving traveling salesman problem and Internet-based search using genetic algorithms.
- Provides knowledge to develop hybrid fuzzy controllers using soft computing techniques
- Details how pocket engine control is acheived using various soft computing methods.

17.1 Introduction

In this chapter we are going to discuss few applications of neural networks, fuzzy logic, genetic algorithm and hybrid systems. As we already know soft computing has a wide range of applications. Here a few topics of its applications are being covered. We believe that the chapter would give the reader a brief idea of how the soft computing can be applied to any practical problem.

17.2 A Fusion Approach of Multispectral Images with SAR (Synthetic Aperture Radar) Image for Flood Area Analysis

There have been several efforts to monitor and assess the area destroyed by floods, especially, the monsoon regions that were suddenly inundated by slash flood caused by the storm and other natural hazards, such as El Nino, LA Nina I, etc. Floods cause much damage to the environment, people's live and properties. Several techniques have been applied to estimate the flood area, important ones being the NDVI (normalized difference vegetation index) derived from multispectral data and 3-second grid DEMs (digital elevation models) to investigate and identify the damaged area depending on elevation intervals. However, the SAR images have been known to efficiently detect floods, because of the object absorption property depending on the moisture of the backscattering wave in radar image. As the multispectral images provide necessary information for land cover interpretation, the fusion of multisensor images achieves the complementary nature of these different data types. Therefore, the fusion techniques have been adopted to perform the flood area classification. To assess the flood areas, JERS-1 SAR data acquired on June 3, 1997 and August 30, 1997 (Figures 17-1 and 17-2) were taken before and during the flood hazard from the tropical storm Zita in Surat Thani province. The cloud penetration capability is shown in Figures 17-1 and 17-2, respectively. Fusion of these images with JERS-1 OPS data acquired on March 14, 1997 (Figure 17-3) was performed to distinguish flood area.

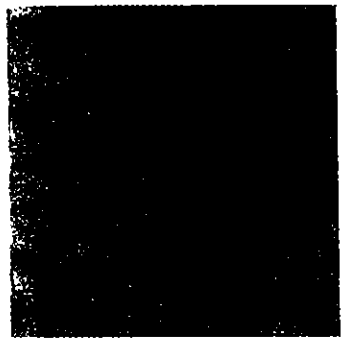


Figure 17-1 SAR image acquired on June 3, 1997, Surat Thani Province.

Figure 17-2 JERS-1 SAR image acquired on August 30, 1997.

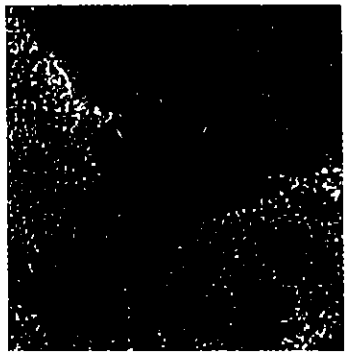


Figure 17-3 JERS-1 OPS data acquired on March 14, 1997.

To study the flood assessment, the image classification is performed using a revolutionary computing methodology known as the artificial neural network (ANN) computing. This method presents how the neuron in the human brain processes the data to identify the complex and noisy patterns of information. An error back-propagation ANN structure is used in this section.

17.2.1 Image Fusion

Image fusion integrates both spatial and spectral data to hold the superior characteristics of multisensor images and improve the knowledge of scene. Therefore, the fused images could improve the accuracy image classification and help the feature extraction and recognition. The image fusion can be divided into two classes: spatial domain method and spectral domain method. The second method is used in most applications including color space transformation. In this section, the Intensity-Hue-Saturation (IHS) model will be used as a color space and the image fusion is done as follows.

1. The RGB color space of OPS images is transformed to the IHS model:

$$I = \frac{R+G+B}{3} \tag{17.1}$$

$$S = 1 - \frac{3}{R+G+B}[\min(R,G,B)] \tag{17.2}$$

$$H = \cos^{-1}\left\{\frac{\frac{1}{2}[(R-G)+(R-B)]}{(R-G)^2+(R-B)(G-B)^2}\right\} \tag{17.3}$$

2. The different gray value of pixel in the black-white of two SAR images (g1 and g2) is added into OPS images intensity:

$$I' = I + (g1 - g2) \tag{17.4}$$

The last term of the above equation is the difference of the images before and during flood. The flood area will be emphasized and nonflood area will be depressed. Adding this term to intensity component in IHS mode means transferring of flood area data to OPS image.

3. The IHS model is inversely transformed to the RGB space and is ready for further classification using neural networks.

17.2.2 Neural Network Classification

In this section, the multilayer perceptron (MLP) neural network based on back propagation (BP) algorithm is used as a classifier, which consists of set of nodes arranged in multiple layers with connection only between nodes in adjacent layer by weights. The input information is presented at input layer as the input vector, and the output vector is the processed information that was retrieved at the output layer. A schematic of a three-layer MLP model is shown in Figure 17-4.

The input and output of the node I in hidden layer of MLP neural network, according to BP algorithm, are:

$$\text{Input: } X_i = \sum_j W_x O_j + b_j \tag{17.5}$$

$$\text{Output: } O_i = f(X_i)$$

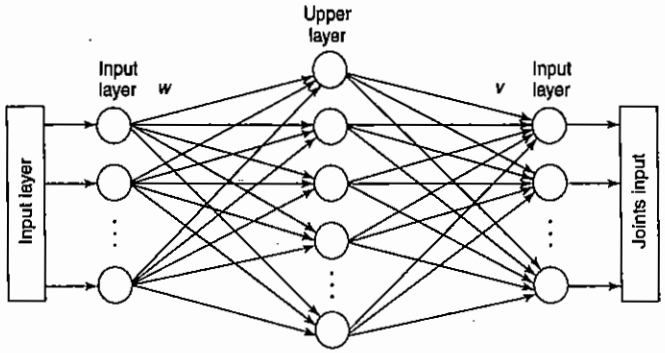


Figure 17-4 The three-layer MLP model of neural network.

where W_{ij} is the weight of connection from node i to node j; B_i the numerical value called bias; f the activation function. In this work, the nonlinear function – sigmoid function given in Eq. (17.6) – is used to determine the output state:

$$f(x_i) = \frac{1}{1 + e^{-x_i}} \tag{17.6}$$

The BP learning algorithm is designed to reduce an error between actual output and desired output in a gradient descent manner. The summed squared error (SSE) is defined as:

$$\text{SSE} = \frac{1}{2} \sum_p \sum_i (O_{pi} - t_{pi})^2 \tag{17.7}$$

where O_{pi} and t_{pi} are the actual and desired outputs of node i when applying the input vector p into the network.

17.2.3 Methodology and Results

17.2.3.1 Method

1. The SAR data obtain 16 bits and then are reduced to 8 bits by using linear scaling in order to obtain 256 values of intensity. From wavelet decomposition, the low wavelet coefficient of SAR images will be used for two reasons – to remove the speckle noise and to continue the proper data for applying to neural network training algorithm.
2. The 12.5 m × 12.5 m resolution of SAR data was reduced to 25 m × 25 m in the same order of OPS resolution. All images should be registered and geometrically corrected.
3. Data fusion technique as mentioned in Section 17.2.1, is used and is shown Figure 17-5.
4. After preprocessing, satellite image is prepared and then applied to neural network classification. Moreover, all data will be classified without fusing and the results will be compared against the fusion data.

17.2.3.2 Results

The results of flood assessment by neural network classification with data fusion and without data fusion are given in Tables 17-1 and 17-2, respectively.

Figure 17-5 Fused image for flood area assessment.

Table 17-1 The result of flood assessment by neural network classification with data fusion

			Urban		Vegetation		Bare soil	
Classification	Water	Cloud	Flood	Nonflood	Flood	Nonflood	Flood	Nonflood
Result/testing (pixel)	447/500	47/50	41/50	42/50	44/50	90/100	41/50	87/100
Correction (%)	89.4	94.0	82.0	84.0	88.00	90.0	82.0	87.0

Table 17-2 The result of flood assessment by neural network classification without data fusion

			Urban		Vegetation		Bare soil	
Classification	Water	Cloud	Flood	Nonflood	Flood	Nonflood	Flood	Nonflood
Result/testing (pixel)	457/500	45/50	38/50	40/50	41/50	92/100	37/50	89/100
Correction (%)	91.4	90.0	76.0	80.4	82.0	92.0	78.0	89.0

The study results show that multitemporal SAR data are very useful for flood assessment and monitoring. On the other hand, the OPS data provide the necessary information for land cover interpretation. The fusion of these data is very helpful for flood assessment classification, because it enhances the flood area and gives a highly reliable result.

17.3 Optimization of Traveling Salesman Problem using Genetic Algorithm Approach

The traveling salesman problem (TSP) is conceptually simple. The problem is to design a tour of a set of n cities in which the traveler visits each city exactly once and then returns to the starting point. One has to minimize the distance traveled. Solving this combinatorial problem is NP (nondeterministic polynomial time) difficult as the search space is n factorial. Examining all possible Hamiltonian circuits of n vertices by calculating edge weights is certain to reveal the optimum solution, but cannot guarantee to do so in a tractable time for all n.

Perhaps first considered by Euler as the knights' tour problem in 1759, and popularized by the RAND Corporation in the 1950s, TSP has many applications that involve large numbers of vertices. These include VLSI – for which size can exceed one million – circuit board drilling, X-ray crystallography and many

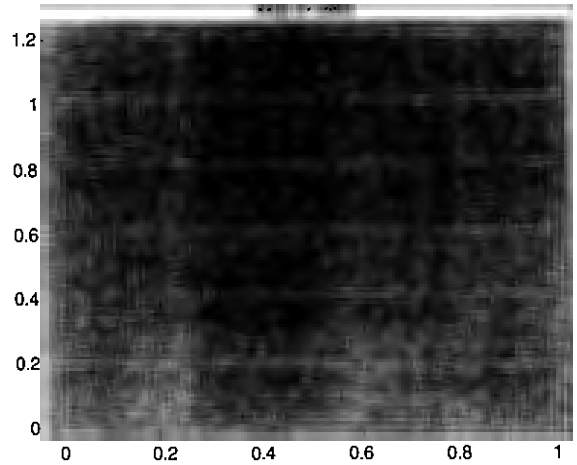


Figure 17-6 Points placed in a unit square representing a landscape of 14 cities.

scheduling problems. Therefore it is important to find algorithms that lower the time costs by providing reasonably good solutions.

This section explores application of GAs to TSP by examining combinations of different algorithms for the binary and unary operators used to generate better solutions and minimize the search space. Three binary and two unary operations were tested. These were compared to a base-line developed from a brute force algorithm for a tractable 14-city problem and to random tour generation, which provides mean and standard deviation statistics very close to brute force methods as the distribution of solutions is normal. For a 14-city problem, Figure 17-6 shows the points placed in a unit square representing a landscape of 14 cities.

The binary methods examined include uniform order-based crossover (OCX), heuristic order-based crossover (HCX) and edge recombination (ER). Unary operators were reciprocal exchange and inversion.

17.3.1 Genetic Algorithms

Genetic algorithms are modeled on biological processes in which parents pass character traits to their offspring. The next generation contains data inherited from its predecessors and in each generation the fittest members have the greatest potential to survive and send genetic material to the progeny of their population. As children are developed from the best parents, they are likely to introduce an improvement in fitness of the group.

Genetic algorithms mimic this survival of the fittest by randomly generating a population of solutions and then selecting members, with greater possibility of selection given to the fittest, from which to build the next generation. This section used populations of 100 subjects and evolved each trial for 1000 generations. The objective function used in this section was the length of the paths. Shorter circuits were given best fitness consideration by inverting their tour lengths through subtraction from the ceiling of the population's longest tour. The roulette wheel method was employed to choose parent solutions.

Successors were developed by binary operations called crossovers that create child solutions using information inherent in the two chosen parents. The type of information passed is problem dependent and affects the fitness of the resultant population. These experiments tested three crossover operators and in all cases allowed cloning of chosen solutions to occur at a rate of 0.30.

Unary operators mutate individual solutions and are applied at a low rate. In this case two mutation methods were employed at a rate of 0.01. Their purpose is to allow solutions to exceed local maximums but they are usually destructive to the mutated offspring.

17.3.2 Schemata

The solutions created by genetic algorithms are instances of schema. They belong to a set of other solutions that share common traits. A solution [0 1 0 0] is an instance of the schema [0 * * 0], where the asterisks may represent either bit value. The solution [0 1 1 0] would also be an instance of this schema of order two, which could be thought of as a regular expression representing all strings of length four over the alphabet {0, 1} beginning and ending with zeros. The fitness of a schema is the average of the fitness's of its instances.

The probability that a schema S found in one generation will occur in the next is given by,

$$P_S^{CX} = 1 - P_{CX}[L_d/(L-1)]$$

where $1 - P_{CX}$ is the rate at which cloning occurs and L_d is the defining length of the schema, or the number of bits between the outermost set bits, and L is the total length of the solution. For our purposes $P_S^{CX} = 1 - 0.70(L_d/13)$.

The probability of remaining in the population after the mutation cycle is given by

$$P_S^M = (1 - P_M)^n$$

where $1 - P_M$ is the probability that a bit will not undergo mutation and n is the order of the schema, or the number of set bits. In these experiments $P_S^M = 0.99^n$.

The success of genetic algorithms lies in the propagation of the fittest schemata.

17.3.3 Problem Representation

All reproduction and mutation operators tested employed a path representation of the problem. This implies that a list of cities (1 2 3 4) would represent the circuit 1-2-3-4-1. This representation was chosen over others such as adjacency, ordinal or matrix, because it is a very natural representation and accommodates many crossover functions that result in valid offspring. Hence it eliminates the need for repair algorithms or penalty functions.

Two of the binary operators tested augment this representation with additional information. The heuristic order-based crossover includes a list of the distances between each leg of the tour. The edge recombination crossover method utilizes edge lists for each city that are complied from the parent information.

17.3.4 Reproductive Algorithms

The first binary operator tested was the uniform-order-based crossover, which is useful when order is significant to a problem, and preserves the legality of solutions. This method first creates a random binary string which is of the same length as the parent tours. The child receives information from parent one in all positions corresponding to 1 in the binary string. On average 50% of the data then comes from parent one and reflects the positions of cities in this path. The rest of the tour is filled in from parent two using those cities not already in the child and having the order found in parent two. For example, if parent one is (1 2 3 4), parent two is (4 3 2 1) and the binary string is (1 0 1 0), then the child becomes (1 4 3 2).

The next crossover method employed was modeled on heuristic crossovers developed for adjacency representations, which favor edges with more desirable fitness values. This operator employed a list of distances

between edges of the tours such that a path (1 2 3) with a distance map of (0.2, 0.8, 0.3) would correspond to edge (1 2) having distance 0.2, edge (2 3) having distance 0.8 and edge (3 1) having distance 0.3. The offspring created receive edge and not position information for an average of 50% of their data. A roulette wheel function gives greater probability for selection to more fit edges. The rest of the information is filled by using the cities not already passed to the child in the order found in the remaining parent. Here shorter legs of a tour are treated like dominant genetic traits. They are features that are more likely to be passed to the child.

The final binary operator tested focused on passing as much edge information as possible to the child. Edge recombination (ER) uses no heuristic rules or fitness information, but insures that each offspring receives 95% of their edge information from the parents. ER uses edge maps such that if parent one is (1 2 3 4 5) and parent two is (5 2 3 4 1), then mapping is performed as given below:

1 : (2 5 4)
2 : (1 3 5)
3 : (2 4)
4 : (3 5 1)
5 : (4 1 2)

An initial city is chosen randomly between the first cities of the parents and placed in the offspring as the current city. The next city chosen is taken from the edge list of the current city giving priority to cities with shorter edge lists, the largest list possible being of length four. Ties are broken randomly and used cities are removed from choice available. In the event of edge failure, defined as a current city of edge list length zero, the next point is chosen randomly from the cities not yet visited.

17.3.5 Mutation Methods

The first unary operator tested was reciprocal exchange, which simply swaps two randomly chosen elements in the solution. Inversion was the other method applied. Here all values between two elements in a list are reversed. A solution (1 2 | 3 4 5 6 | 7), where the bars represent random break points with a window size of four, would produce the mutant (1 2 6 5 4 3 7).

17.3.6 Results

Examining measures of center for the data distributions, OCX was the overall best binary operation performer with inversion mutation method. HCX also performed well with the inversion operator. All genetic method distributions were skewed toward the shorter tour lengths with OCX showing the lowest modes.

The ER crossover posted the lowest standard deviations of the GAs coming very close to baseline.

All methods produced excellent shortest path means with many trials finding the optimum solution. OCX and ER, both using reciprocal exchange, found the optimum tour length with greatest frequency.

All genetic operators produced good shortest path means in less than 37 s. The generational time to best solution was 186 generations using the ER binary operator with reciprocal exchange mutation. Since all methods find good solutions quickly, the smallest number of generations necessary to produce near optimum results is an important criterion. ER was consistently the best number of generations to solution performer.

While ER descended to best tour times the fastest, OCX performed better finding the optimum solution and produced better results for all generations. HCX, which favored shorter edge lengths in child construction, did not perform as well as expected. Table 17-3 shows the scores obtained for various algorithms on an average of 20 trials.

Table 17-3 Averages of 20 trials where *Z* scores represent the number of standard deviations from the exhaustive search baseline

Algorithms	Mean	Mode	Standard deviation	Shortest path	Longest path	Z score	Time (ms)
HCX/Inversion	4.94	4.38	1.07	3.78	10.29	−4.09	8389
HCX/Reciprocal exchange	5.05	4.50	1.08	3.83	10.58	−3.95	7954
OCX/Inversion	4.78	3.88	1.04	3.87	10.36	−4.29	6796
OCX/Reciprocal exchange	5.02	3.88	1.07	3.77	10.61	−3.99	6821
ER/Inversion	5.00	4.62	0.81	3.81	10.21	−4.01	36280
ER/Reciprocal exchange	5.11	4.12	0.87	3.77	10.26	−3.88	36310
Random tours	8.21	8.38	0.81	4.23	10.99	0	6440
Exhaustive search*	8.21	8.38	0.80	3.76	11.08	0	36*E*6

*Only one trial.

An examination of the schemata of some final populations reveals that HCX produced the most homogenous solutions. Total population schemata of order seven, 50% of the total tour length of 14, were not uncommon. HCX schemata had much in common with greedy algorithm solutions. The effect of this operator was to favor a more narrow set of fit schemata rather than to produce shorter solutions more quickly. ER produced the most diverse final populations, with most final population schemata at order zero. While edge information is significant in TSP solutions, the relative positions of points within the solution also play a strong role in finding optimum values. The success of the OCX crossover, which passes relative position information to the offspring, is an evidence of this.

The inversion mutation operator performed well in center for data distribution, but the reciprocal exchange method performed well in optimum solutions. This operator actually introduces more variety to a population than inversion in TSP because the direction of a path does not change its length. Only the values at the two edges disrupted are changed when inversion is applied. Reciprocal exchange changes at minimum two and maximum four edge values.

17.4 Genetic Algorithm-Based Internet Search Technique

Among the huge number of documents and servers on Internet, it is hard to quickly locate documents that contain potentially useful information. Therefore, the key factor in software development nowadays should be the design of applications that efficiently locate and retrieve Internet documents that best meet user's requests. The accent is on intelligent content examination and selection of documents that are most similar to those submitted by the user as input.

One approach to this problem is indexing all accessible web pages and storing this information into the database. When the application is started, it extracts keywords from the user-supplied documents and consults the database to find documents in which given keywords appear with the greatest frequency. This approach, besides the need to maintain a huge database, suffers from the poor performance – it gives numerous documents totally unconnected to the user's topic of interest.

The second approach is to follow links from a number of documents submitted by the user and to find the most similar ones, performing a genetic search on the Internet. Namely, application starts from a set of input documents, and by following their links, it finds documents that are most similar to them. This search and evaluation is performed using GAs as a heuristic search method. If only links from input documents are followed, it is the Best First Search or genetic search without mutation. If, besides the links of the input documents, some other links are also examined, it is genetic search with mutation.

The second approach was realized and tested at the University of Hong Kong, and the results were presented at the 30th Annual Hawaii International Conference of System Sciences. The Best First Search was compared to genetic search where mutation was performed by picking a URL from a subset of URLs covering the selected topic. That subset is obtained from a compile-time generated darabase. It was shown that search using GAs gives better results than the Best First Search for a small set of input documents, because it is able to step outside the local domain and examine a larger search space.

There is an ongoing research at the University of Belgrade concerning mutation exploiting spatial and temporal localities. The idea of spatial locality exploitation is to examine documents in the neighborhood of the best-ranked documents so far, i.e., the same server or local network. Temporal locality concerns maintaining information about previous search results and performing mutation by picking URLs from that set.

If either of above described methods is performed, a lot of time is spent for transferring documents from the Internet onto the local disk, because content examination and evaluation must be performed off-line. Thus, a huge amount of data is transferred through the network in vain, because only a small percent of transferred documents will turn out to be useful. The logical improvement is construction of mobile agents that would browse through the network and perform the search locally, on the remote servers, transferring only the needed documents and data.

The first step of a genetic algorithm is to define a search space and describe a complete solution of a problem in the form of a data structure that can be processed by a computer. Strings and trees are generally used, but any other representation could be equally eligible, provided that the following steps can be accomplished, too. This solution is referred to as *genome* or *individual.*

The second step is to define a convenient evaluation function (*fitness function*) whose task is to determine what solutions are better than others. One approach, when meeting diverse requests, is to add a certain value for every request met and substract another value for every rule violated. For instance, in a class-scheduling problem, genetic algorithm can add 5 points for every solution that has Mr. Jones lecturing only in the afternoon and substract 10 for any one that has two lecturers teaching in the same classroom at the same time. Of course, many problems require a specific definition of the fitness function which works best in that case.

The third step in the creation of a GA is to define reproduction, crossover and mutation operators that should transform the current generation into the next one. Reproduction can be generalized, namely, for every problem one can pick out individuals for mating randomly or according to their fitness function (only few of the best are allowed to mate). The harder part is to define crossover and mutation operators.

These operators depend strongly on the problem representation and require thorough investigation, plus a lot of experimenting to become truly efficient. Crossover generates a new offspring by combining genetic material from two parents. It incarnates the assumption that the solution which has a high fitness value owes it to a combination of its *genes.* Combining good genetic material from two individuals, better solutions can be obtained. Mutation introduces some randomness into population. Using only a crossover operator is a highly unwise approach, because it might lead to a situation when one individual (in most cases only slightly better than others) dominates the population and the algorithm "gets stuck" with an averagely good solution, and no way to improve it by examining other alternatives. Mutation randomly changes some genes in an individual, introducing diversity into population and exploring a larger search space. Neverthless, a high rate of mutation can bring oscillations in the search, causing the algorithm to drift away from good solutions and to examine worse ones, thus converging more slowly and unpredictably.

The fourth step is to define stopping criteria. Algorithm can either stop after it has produced a definite number of generations or when the improvement in average fitness over two generations is below a threshold. The second approach is better, yet the goal might be hard to reach, so the first one is more resonable. Having done all this, one can write a code for a program performing the search (which is fairly simple at this point).

The GA strength comes from the implicitly parallel search of the solution space that it performs via a population of candidate solutions and this population is manipulated in the simulation.

A fitness function is used to evaluate individuals, and reproductive success varies with fitness. An effective GA representation (i.e., converting a problem domain into genes) and meaningful fitness evaluation are the keys of the success in GA applications.

Although this mechanism seems "too good to be true" it gives excellent results, when compared to other approaches, regarding the time spent in search and quality of the solutions found.

17.4.1 Genetic Algorithms and Internet

Basic idea in customizing Internet search is construction of an intelligent agent – a program that accepts a number of user-supplied documents and finds documents most similar to them on Internet. GA imposes itself as "a right tool for the job" since it can process many documents in parallel, evaluate them according to their similarity to the supplied ones, and generate a result in the form of a group of documents found.

Intelligent agent for the Internet search performs the following steps:

1. Processes a set of URLs given to it by a user and extracts keywords, if necessary, for evaluation.
2. Selects all links from the input set and fetches the corresponding www presentations; the resulting set represents the first generation.
3. Evaluates the fitness functions for all elements of the set.
4. Repeatedly performs reproduction, crossover and mutation, and thus transforms the current generation into the next one.

There are several issues of importance that have to be considered when designing a genetic algorithm for intelligent Internet search. These are:

1. representation of genomes;
2. definition of the crossover operator;
3. selection of the degree of crossover;
4. definition of the mutation operator;
5. definition of the fitness function;
6. generation of the output set.

Each of the issues given above is described next, in the form of a classification of possible ways to implement the issue. For each of the issues, two pictures are presented: a classification of possible approaches regarding that issue and the most frequently used implementation.

17.4.2 First Issue: Representation of Genomes

First issue to be discussed is how one can encode possible solutions. In this case, one solution is URL, the address of an Internet document. The aim is to create a result in the form of a list or an array of those documents, and that average fitness of this set be the highest possible. Figure 17-7 gives the possible representation approaches.

17.4.2.1 String Representation

String representation seems to be a natural choice since URL is already string-encoded. However, in this case, there is only one gene in a genome, and classical crossover and mutation cannot be performed. Therefore, a new

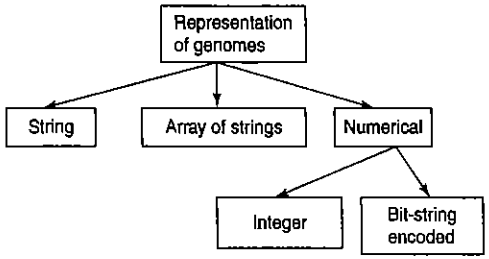


Figure 17-7 A taxonomy of representation of genomes.

definition of the crossover and mutation operators is needed that would be applicable in this environment. Nevertheless, redefinition of genetic operators seems to be the only reasonable thing to do, since classical crossover and mutation can transform GA's search into a "random walk" through the search space.

17.4.2.2 *Array of String Representation*

Each URL contains several fields that have different meanings, so it is convenient to represent it as an array or list of these fields that are string-encoded. Figure 17-8 shows the string representation of a URL terminated by a End-of-String character.

First, there is a name of the Internet protocol either *http* or *ftp*. Only the URLs starting with "http" are of interest to us. Therefore, the agent should take into consideration only the documents that these URLs point to.

Second, there is a server address consisting also of several fields: net name (*www, usenet,* etc.), server name, and additional information concerning the type of organization that the server belongs to (*com* for commercial organizations, *org* for noncommercial ones, *edu* for universities and schools, and so on). In some cases, this field can contain specification of the city and country the server is located in.

The third is the string that gives the path from the server root to a particular document.

All these fields must be variable in length so that the solution can be represented in the form of an array of variable-length strings. Mutation and crossover operators can be implemented more easily in this case than it was possible with the string representation, because there are several genes in a genome that can be crossed or mutated. Either classical or user-defined crossover and mutation can be performed.

17.4.2.3 *Numerical Representation*

Since every Internet address up to the document path is number-encoded, genetic algorithm can use this representation in order to perform classical crossover and mutation. However, this is not a promising approach since documents similar to one another seldom reside on adresses that have much in common. Therefore genetic algorithm would actually perform a random search. Moreover, many of the addresses generated may not exist at all, which places more overhead than can be tolerated. Numerical representation can be either left *integer-encoded* or transformed into a *bit-string representation.*

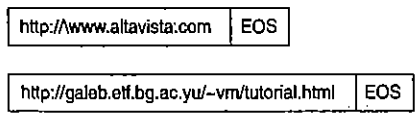


Figure 17-8 The most frequent approach regarding the representation of genomes: String representation. URL is represented as a string, terminated by an End-Of-String character.

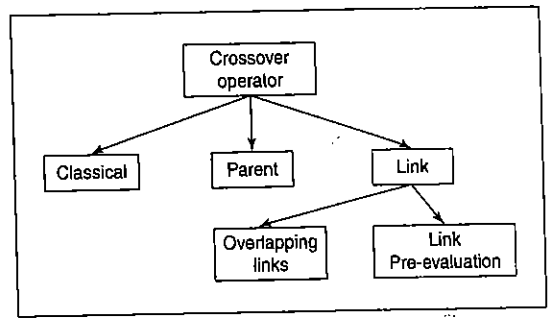


Figure 17-9 A taxonomy of the crossover operator.

17.4.3 Second Issue: Definition of the Crossover Operator

Crossover operator is used to produce a new offspring by combining genetic material from two parents, each one characterized with a high fitness function. The idea is to force out the domination of good genes in the future populations. Figure 17-9 gives the possible definitions of the crossover operator.

17.4.3.1 *Classical Crossover*

Classical crossover can be performed only if URLs are encoded as array of strings or numerically, since it requires that individual contains more than one gene.

It is performed by combining different fields of an address. This approach would, in most cases, produce URLs that do not exist on Internet. For example, combining *www.msu.edu* with *www.novagenetica.com* could result in *www.msu.com* or in *www.novagenetica.edu*. Neither of these is a valid Internet address. Though this technique is slightly better than a random search, but is not a wise choice.

17.4.3.2 *Parent Crossover*

Parent crossover is performed by picking out parents from the mating pool and choosing a constant number of their links as offspring, without any evaluation of those links. This approach is easy to conduct; however, it could result in many nonrelevant documents being picked out for the next generation, because one document can contain links to many sites that are not related to the user's subject of interest.

17.4.3.3 *Link Crossover*

Link crossover can, if carefully performed, produce more meaningful results than classical crossover. The idea is to examine links from the documents in the mating pool and pick those that are the best for the next generation. This evaluation of the links could be done in two ways: *overlapping links* and *link pre-evaluation.*

17.4.3.4 *Overlapping Links*

The links of the parent documents are being compared to the links of the input documents, and only those that they have in common are selected as offspring. This technique might not be the best one (it is always possible that the fittest document contains links that have nothing to do with the links of input documents), but can be easily conducted. Moreover, it is a common practice on Internet that documents contain links to related sites, so this approach could score high in most cases.

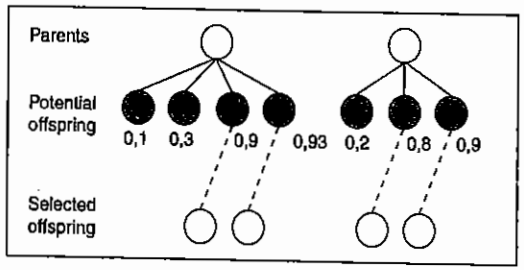


Figure 17-10 The most frequent approach regarding the crossover operator: Link pre-evaluation. Numbers next to the nodes represent normalized values of their fitness functions. Offspring with the greatest values are selected for the next generation.

17.4.3.5 Link Pre-Evaluation

Computation of the fitness function can be performed on the documents that parent links refer to, and the best ones can be picked out for the next generation. Since this computation must be done sooner or later, it places a small overhead on the program (because of the evaluation of the documents that will not be picked out for the next generation), but it gives good results. However, this approach could be time-consuming in the case when the documents in the mating pool contain many links, since genetic algorithm must wait for all documents that those links refer to, to be fetched and evaluated in order to proceed with its work. Figure 17-10 shows the approach of link pre-evaluation operator.

17.4.4 Third Issue: Selection of the Degree of Crossover

There are two different approaches regarding crossover and insertion of offspring into the next generation. Figure 17-11 gives possible approaches concerning the degree of crossover.

17.4.4.1 Limited Crossover

Only a fixed number of offsprings can be produced from each couple. This could result in rejection of documents that have higher fitness values than offsprings of other nodes but are ranked less than second among offspring documents of their parent node.

17.4.4.2 Unlimited Crossover

Genetic algorithm can rank the documents from the mating pool and all documents that parent links refer together according to the values of their fitness function. Then, it can pick from this set those individuals that can be forwarded to the next generation. Overall fitness would be better and there is no risk of losing some good solutions.

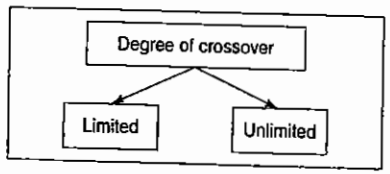


Figure 17-11 A taxonomy of the degree of crossover.

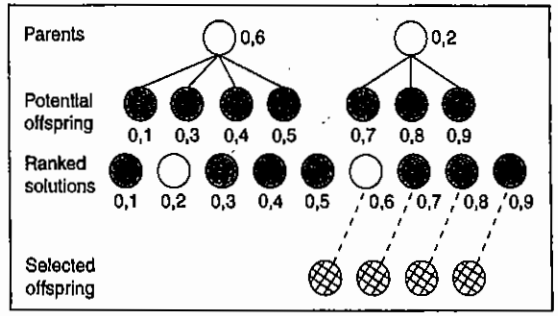


Figure 17-12 Unlimited crossover.

The most frequent approach regarding the selection of the degree of crossover is the Unlimited crossover. In this case parents and all offspring are ranked according to their fitness function values and the best among them are selected. Thus, instead of picking out nodes with the fitness function values of 0,4, 0,5, 0,8, 0,9, as would be done using limited crossover, only the best solutions are chosen for the next generation. Figure 17-12 shows the operation of unlimited crossover.

17.4.5 Fourth Issue: Definition of the Mutation Operator

Mutation is used to introduce some randomness in the population and thus slow down the convergence and cover more of the search space. Figure 17-13 gives the possible definitions of the mutation operator.

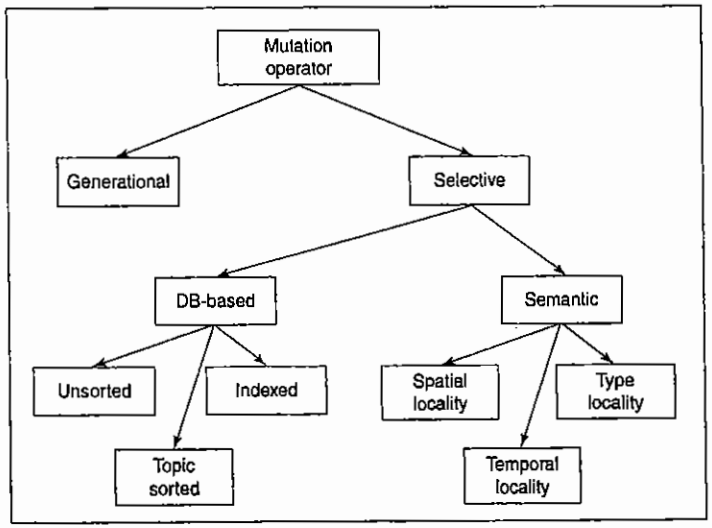


Figure 17-13 A taxonomy of the mutation operator.

17.4.5.1 Generational Mutation

Generational mutation is performed by generating a URL randomly. It is easily conducted, but has no significance since high percent of URLs generated in this way would not exist at all. So, conclusion is that URLs for mutation must be picked out from some set of existant addresses, such as database.

17.4.5.2 Selective Mutation

Selective mutation is performed by selecting URLs from a set of existing URLs. It can be either DB-based or semantic.

DB-based: DB-based mutation is based on the existence of a database that contains URLs that are somehow sorted. Few of them are picked out and inserted into the population. The URLs can be the following.

1. *Unsorted* – genetic algorithm picks any one of them. This approach usually does not promise good performance.
2. *Topic sorted* – there is a field that says to which topic URL belongs, i.e., entertainment, politics, business. GA chooses only from the set of URLs that belong to the same topic as the input URLs. This approach is a bit limited since one document can cover several topics but should produce reasonably good scores. Figure 17-14 depicts topic-sorted DB-based mutation.
3. *Indexed* – there is a database that contains all words that appear in documents with a certain frequency and also links to documents in which they appear. GA writes a query to a database with keywords from input documents and picks out URLs for mutation from the resulting set. This requires some effort for implementation and updation of the database but promises good scoring. All one has to worry about is finding a compromise between database size and quality of search results.

Semantic: Semantic techniques use some logical reasoning in order to produce URLs for mutation.

1. *Spatial locality mutation*: If GA finds a document of a high fitness value on a particular site, there is a strong possibility that it can find similar documents somewhere on the same server or on the same local network. This is because many people that have accounts on the same server or network usually have similar interests (which is most likely for academic networks). This approach is a bit hard to conduct since GA has to either examine all sites on a server/net (which is time-consuming) or randomly pick a subset of them.
2. *Temporal locality mutation*: A database is maintained of a huge number of documents that were in the result set, for every search made. GA keeps scoring them on how frequently they appear in that set. Those with high frequency promise to give good performance in the future too, so GA inserts them in the

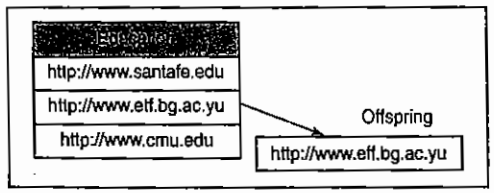


Figure 17-14 The most frequent approach regarding the definition of the mutation operator: Topic-sorted DB-based mutation. Offspring are selected from the set of the documents in a database that are related to a certain topic.

population, thus performing the mutation. This will yield good results for usual queries (from the field that many users are interested in) but will do poor for less popular ones.

3. *Type locality mutation*: This mutation is based on a type of the site the input documents are located on. If it is, say, an *edu* site then there is a strong probability that some other sites with same sufix have similar documents. A database is maintained containing types of sites and a set of URLs referencing those sites and GA chooses the candidates for mutation from this set.

Although last two types of mutations deal with databases, they involve logical reasoning and semantics consideration in picking out URLs for mutation, and therefore are not classified as DB-based.

17.4.6 Fifth Issue: Definition of the Fitness Function

To evaluate fitness of a document, GA must go through it and examine its contents. Figure 17-15 gives several possible definitions of the fitness function.

17.4.6.1 Simple Keyword Evaluation

Once the occurences of keywords (selected at the beginning from input files) in the document are counted, the GA can simply add those numbers and add a certain value when two or more keywords appear in the document (so it ranks it higher than those documents that contain only one keyword). It can also add a value for occurences of keywords in a title or hyperlinks. This method, though rough, can produce fairly good results with minimum time spent for evaluation.

17.4.6.2 Jaccard's Score

Jaccard's score is computed based on links and indexing of pages as follows:

Jaccard's score from links: Given two homepages x and y and their links, $X = x_1, x_2, \ldots, x_m$ and $Y = y_1, y_2, \ldots, y_n$, the Jaccard's score between x and y based on links is computed as follows:

$$JS_{links} = \frac{\#(X \cap Y)}{\#(X \cup Y)}$$

Jaccard's score from indexing: Given a set of homepages, the terms in these homepages are identified (keywords). The term frequency and the homepages' frequencies are then computed. Term frequency, tf_{xj}, represents the number of occurences of term j in homepage x. Homepage frequency, df_j, represents the number

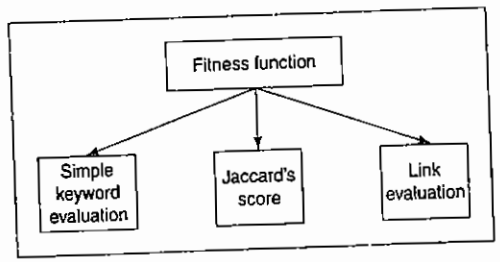


Figure 17-15 A taxonomy of the fitness function.

of homepages in a collection of N homepages in which term j occurs. The combined weight of term j in homepage x, dx_j, is computed as follows:

$$d_{xj} = tf_{xj} * \log\left(\frac{N}{df_j} * w_j\right)$$

where w_j represents the number of words in term j and N represents the total number of homepages. The Jaccard's score between homepages x and y based on indexing is then computed as follows:

$$JS_{indexing}(x, y) = \frac{\sum_{j=1}^{L} d_{xj} d_{yj}}{\sum_{j=1}^{L} d_{xj}^2 + \sum_{j=1}^{L} d_{yj}^2 + \sum_{j=1}^{L} d_{xj} d_{yj}}$$

where L is the total number of terms. Fitness function for homepage h_i is then computed as follows:

$$JS_{links}(h_i) = \frac{1}{N} \sum_{j=1}^{N} JS_{links}(input_j, h_i)$$

$$JS_{indexing}(h_i) = \frac{1}{N} \sum_{j=1}^{N} JS_{indexing}(input_j, h_i)$$

Fitness function is defined as

$$JS(h_i) = \frac{1}{2}[JS_{links}(h_i) + JS_{indexing}(h_i)]$$

Although computation of this fitness function can be time-consuming for a big population, it gives excellent results concerning quality of homepages retrieved. Figure 17-16 shows the use of Jaccard's score as an evlauation function.

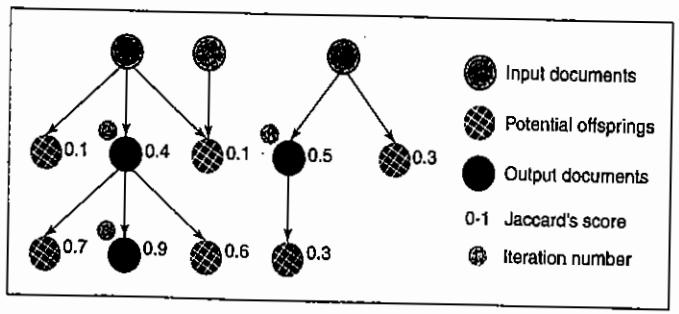


Figure 17-16 The most frequent approach regarding the fitness function: Jaccard's score. The figure illustrates Best First Search performed using Jaccard's score as the evaluation function.

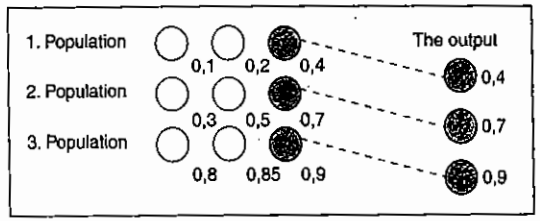


Figure 17-17 The most frequent approach regarding the generation of the output set: Interactive generation. Best individuals from each generation are selected for the output set.

17.4.6.3 *Link Evaluation*

Documents can be evaluated according to the number of links that belong to the set of links of input documents. This narrows the search space but can produce good results in most cases and is easily implemented.

17.4.7 Sixth Issue: Generation of the Output Set

Resulting output set can be either *interactively generated* or *postgenerated.*

17.4.7.1 *Interactive Generation*

From each generation of individuals one or few of them that have highest fitness values are picked out and inserted into the result set. Thus, solutions from earlier generations that are inserted in the output set disqualify later ones that are just below the line for insertion. An advantage is that the user is not required to wait for the end of the search, but can view documents found so far, while the search is performed. Sometimes it is possible even to modify some parameters during the search, i.e., add new input documents or new keywords. Figure 17-17 shows the process of an interactive generation.

17.4.7.2 *Post-Generation*

The final population is the one that represents the last generation and is declared to be the result set. The quality of the documents found is definitely better, and overall fitness is higher than for interactive generation, but user cannot view documents and make modifications until the end of the search.

Because of the fast growth of the quantity and variety of Internet sites, finding the information needed as quickly and thoroughly as possible becomes an important issue for research. There are two approaches to Internet search: indexed search and design of intelligent agents. GA is a search method that can be used in the design of intelligent agents. However, incorporating the knowledge about spatial and temporal localities, and making these agents mobile, can improve the performance of the application and reduce the network traffic.

17.5 Soft Computing Based Hybrid Fuzzy Controllers

Traditional methods, which address robotics control issues, rely upon strong mathematical modeling and analysis. The various approaches proposed till date are suitable for control of industrial robots and automatic guided vehicles, which operate in various environments and perform simple repetitive tasks that require end effectors positioning or motion along fixed paths. However, operations in unstructured environments require robots to perform more complex tasks for which analytical models for control is very difficult to determine.

In cases where models are available, it is questionable whether or not uncertainty and imprecision are sufficiently accounted for. Under such conditions fuzzy logic control is an attractive alternative that can be

successfully implemented on real-time complex systems. Fuzzy controllers and their hybridization with other paradigms are robust in the presence of perturbations, easy to design and implement, and efficient for systems that deal with continuous variables. The control schemes described in this section are examples of approaches that augment fuzzy logic with other soft computing techniques to achieve the level of intelligence required of complex robotic systems.

Three soft computing hybrid fuzzy paradigms for automated learning in robotic systems are briefly described. The first scheme concentrates on a methodology that uses neural networks (NNs) to adapt a fuzzy logic controller (FLC) in manipulator control tasks. The second paradigm develops a two-level hierarchical fuzzy control structure for flexible manipulators. It incorporates GAS in a learning scheme to adapt to various environmental conditions. The third paradigm employs GP to evolve rules for fuzzy behaviors to be used in mobile robot control.

17.5.1 Neuro-Fuzzy System

Neural networks exhibit the ability to learn patterns of static or dynamical systems. In the following neuro-fuzzy approach, the learning and pattern recognition of NN are exploited in two stages: first, to learn static response curves of a given system and second, to learn the real-time dynamical changes in a system to serve as a reference model. The neuro-fuzzy control architecture uses two neural networks to modify the parameters of an adaptive FLC. The adaptive capability of the fuzzy controller is manifested in a rule generation mechanism and automatic adjustment of scaling factors or shapes of membership functions. The NN functions as a classifier of the system's temporal responses.

A multilayer perceptron NN is used to classify the temporal response of the system into different patterns. Depending on the type of pattern such as "response with overshoot," "damped response," "oscillating response," etc. the scaling factor of the input and output membership functions is adjusted to make the system respond in a desired manner. The rule generation mechanism also utilizes the temporal response of the system to evaluate new fuzzy rules. The nonredundant rules are appended to the existing rule base during the tuning cycles. This controller architecture is used in real-time to control a direct drive motor.

17.5.2 Real-Time Adaptive Control of a Direct Drive Motor

In order to perform real-time control, it is necessary for the controller to stand alone with the sole task of calculating the output needed to control the object system. This means the task of communicating data for storing as well as acquiring controller parameters (if the controller is adaptive) should be performed by external processors. In this way a real-time control can be achieved with required sampling rate for high bandwidth operation.

The FLC algorithm requires processing of several functionalities such as fuzzification, inferencing and defuzzification.

This means the computation time taken by the FLC itself does not leave any room for an adaptive algorithm such as rule generation, calculating the scale factor of the membership function, or NN algorithms. In order to implement all these functionalities, a multiprocessing architecture is needed. This can be achieved by combining a sufficiently fast processor specifically designed for real-time processing, such as a TMS320C30 digital signal processor (DSP) combined with a PC Intel processor (Pentium or 486).

17.5.3 GA-Fuzzy Systems for Control of Flexible Robots

In this section, GAs are applied to fuzzy control of a single link flexible arm. GAs are guided probabilistic search routines modeled after the mechanics of Darwinian theory of natural evolution. GAs have demonstrated the

coding ability to represent parameters of fuzzy knowledge domains such as fuzzy rule sets and membership functions in a genetic structure, and hence are applicable to optimization of fuzzy rule sets.

Several issues should be addressed when designing a GA for optimizing fuzzy controllers: the design of a transformation (interpretation) function, the method of incorporating initial expert knowledge and the choice of an appropriate fitness function. Each of the above issues significantly influences the success of GA in finding improved solutions. These issues are briefly discussed below as they apply to design of a GA-fuzzy controller for a flexible link.

17.5.3.1 *Application to Flexible Robot Control*

The application of GA-fuzzy systems applied to flexible robot is discussed here. The GA-learning hierarchical fuzzy control architecture is shown in Figure 17-18. Within the hierarchical control architecture, the higher-level module serves as a fuzzy classifier by determining spatial features of the arm such as *straight, oscillatory, curved.* This information is supplied to the lower level of hierarchy where it is processed among other sensory information such as errors in position and velocity for the purpose of determining a desirable control input (torque). In this, control system is simulated using only a priori expert knowledge. In the given structure, a GA fine-tunes parameters of membership functions.

The following fitness function was used to evaluate individuals within a population of potential solutions:

$$\text{Fitness} = \int_{t_i}^{t_f} \frac{1}{e^2+\gamma^2+1}\,dt$$

where e represents the error in angular position and γ represents overshoot. Consequently, a fitter individual is an individual with a lower overshoot and a lower overall error (shorter rise time) in its time response. Here, results from previous simulations of the architecture are applied experimentally. The method of *grid-parenting* was used to create the initial population.

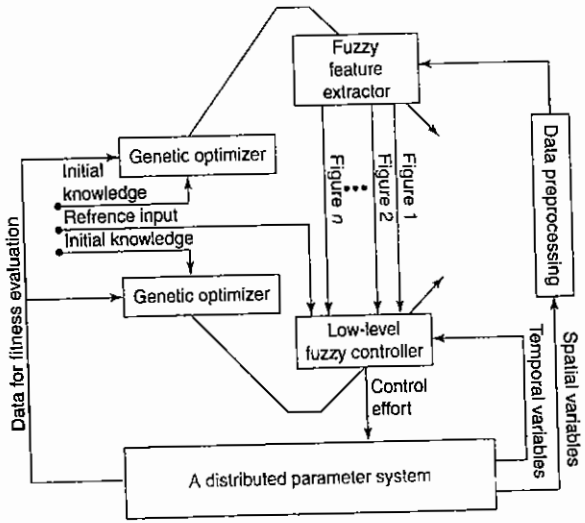


Figure 17-18 GA-based learning hierarchical control architecture.

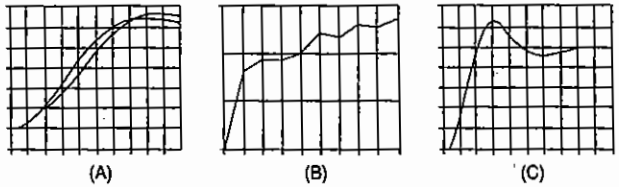


Figure 17-19 GA simulation: (A) Comparison of simulation responses; (B) plot of average fitness; (C) initial experimental results.

Members of the initial population are made up of mutation of the knowledgeable grandparent (sb). As a result, a higher fit initial population results in a faster rate of convergence as is exhibited in Figure 17-19(A). Figure 17-19(A) shows the time response of the GA-optimized controller when compared to previously obtained results through the non-GA fuzzy controller.

17.5.4 GP-Fuzzy Hierarchical Behavior Control

The robot control benefits to be gained from soft computing-based hybrid FLCs is not limited to rigid and flexible manipulators. Similar benefits can be gained in applications to control of mobile robot behavior. Autonomous navigation behavior in mobile robots can be decomposed into a finite number of special-purpose task-achieving behaviors. An effective arrangement of behaviors as a hierarchical network of distributed fuzzy rule bases was recently proposed for autonomous navigation in unstructured environments. The proposed approach represents a hybrid control scheme incorporating fuzzy logic theory into the framework of behavior-based control.

A behavior hierarchy that encompasses some necessary capabilities for autonomous navigation in indoor environments is shown in Figure 17-20. It implies that goal-directed navigation can be decomposed as a behavioral function of goal-seeking and route-following. These behaviors can be further decomposed into the lower-level behaviors shown, with dependencies indicated by the adjoining lines. Each block in Figure 17-20 is a set of fuzzy logic rules.

The circles in the figure represent dynamically adjustable weights in the unit interval, which specify the degree to which low-level behaviors can influence control of the robot's actuators. Higher-level behaviors

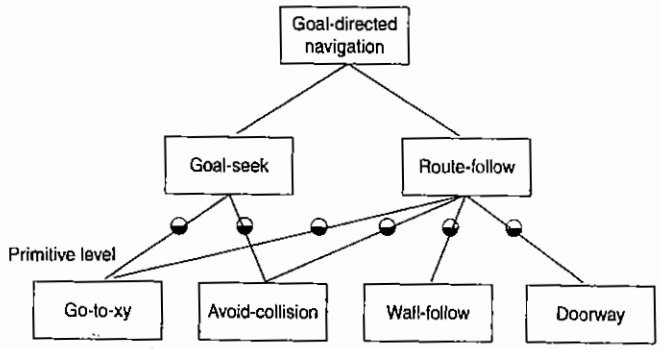


Figure 17-20 Hierarchical decomposition of mobile robot behavior.

consist of fuzzy decision rules, which specify these weights according to goal and sensory information. Each low-level behavior consists of fuzzy control rules, which prescribe motor control inputs that serve to achieve the behavior's designated task.

The functionality of this hierarchical fuzzy-behavior control approach depends on a combined effect of the behavioral functionality of each low-level behavior and the competence of the higher-level behaviors that coordinate them. Perhaps the most difficult aspect of applying the approach is the formulation of fuzzy rules for the higher-level behaviors. This is not entirely intuitive, and expert knowledge on concurrent coordination of fuzzy-behaviors is not readily available. This issue is addressed using GP to computationally evolve rules for composite behaviors. The forthcoming section describes the genetic programming approach to fuzzy rule-base learning.

17.5.5 GP-Fuzzy Approach

The GP paradigm computationally simulates the Darwinian evolution process by applying fitness-based selection and genetic operators to a population of individuals. Each individual represents a computer program of a given programming language and is a candidate solution to a particular problem. The programs are structured as hierarchical compositions of functions (in a set F) and terminals (function arguments in a set T). The population of programs evolves over time in response to selective pressure induced by the relative fitness's of the programs for solving the problem.

For the purpose of evolving fuzzy rule bases, the search space is contained in the set of all possible rule-bases that can be composed recursively from F and T. The set F consists of components of the generic *if-then* rule and common fuzzy logic connectives, i.e., functions for antecedents, consequents, fuzzy intersection, rule inference and fuzzy union. The set T is made up of the input and output linguistic variables and the corresponding membership functions associated with the problem. A rule base that could potentially evolve from F and T can be expressed as a tree data structure with symbolic elements of F occupying internal nodes and symbolic elements of T as leaf nodes of the tree. This tree structure of symbolic elements is the main feature, which distinguishes GP from GAs, which use the numerical string representation.

All rule bases in the initial population are randomly created, but descendant populations are created primarily by reproduction and crossover operations on rule-base tree structures. For the reproduction operation several rule-bases selected on the basis of superior fitness are copied from the current population into the next, i.e., the new generation. The crossover operation starts with two parental rule bases and produces two offsprings that are added to the new generation. The operation begins by independently selecting one random node (using uniform probability distribution) from each parent as the respective crossover point. The subtrees subtending from crossover nodes are then swapped between the parents to produce the two offsprings. GP cycles through the current population perform fitness evaluation and apply genetic operators to create a new population. The cycle repeats on a generation-by-generation basis until satisfaction of termination criteria (e.g. lack of improvement, maximum generation reached, etc.). The GP result is the best-fit rule base that appeared in any generation.

In the GP approach to evolution of fuzzy rule bases, the same fuzzy linguistic terms and operators that comprise the genes and chromosome persist in the phenotype. Thus, the use of GP allows direct manipulation of the actual linguistic rule representation of fuzzy rule-based systems. Furthermore, the dynamic variability of the representation allows for rule bases of various sizes and different numbers of rules. This enhances population diversity, which is important for the success of the GP system, and any evolutionary algorithm for that matter. The dynamic variability also increases the potential for discovering rule bases of smaller sizes than necessary for completeness, but sufficient for realizing desired behavior.

In this section, the soft computing approaches in handling complex models and unstructured environments are studied. Neuro-fuzzy, GA-fuzzy and GP-fuzzy hybrid paradigms can be successfully implemented to solve

the prominent robot control issues, namely, control of direct drive robot motors, control of flexible links and intelligent navigation of mobile robots. This in near future allows us to combine soft computing paradigms for more intelligent and robust control.

17.6 Soft Computing Based Rocket Engine Control

Many of the rocket engine programs initiated by NASA's Marshall Space Flight Center (MSFC) in Huntsville, Alabama, have been successful as evident by success of the Space Shuttle Main Engine, ground testing of the former X-33 engine and Fastrac X-34 engine for the reusable launch vehicle program. As a result, a database of test cases and lessons learned has been created from which improvements to engine control for future engine programs can be made. Such cases include premature engine shutdowns, propellant leaks, and numerous cases of anomalous sensors and data. Such cases are not only costly to the American taxpayer, but also present a risk in social acceptance of current and future space programs.

The Space Transportation Directorate at MSFC has continually expressed an interest in improving engine control and many efforts in various areas for control and anomaly detection and mitigation have been undertaken. Some successful attempts have included nozzle plume analysis and engine vibration analysis. Other efforts, although successful in theory and simulation, have been partially successful in actual engine test firings. It is the harsh engine environment of cryogenics, vibrations, real-time control demands and different engine configurations from test to test that continually encourage researchers to determine alternative solutions or improvements to approaches for engine control and anomaly detection and mitigation.

Current control technologies depend on proven, sometimes archaic, hardware and logical programming techniques which are costly to implement and maintain, and do not account for unforeseen conditions leading to the kinds of problems referenced earlier. The principle goal is to provide another avenue to address MSFC's Space Transportation Directorate's interest in improving overall engine control. An approach for investigating and demonstrating how the application of soft computing technologies can further address presented control issues in rocket engine control is presented in this section as a case study. The testbed engine is shown in Figure 17-21.

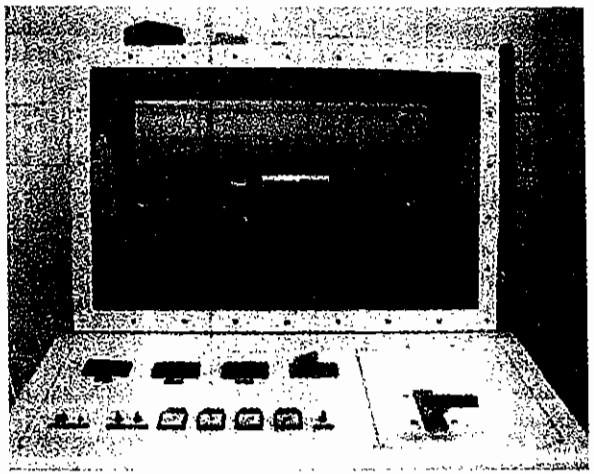


Figure 17-21 Turbine technologies SR-30 turbojet engine.

In this particular work, automation and control of a small-scale turbojet engine is described and some preliminary data obtained using a PID controller has been provided. Turbine technologies turbojet engine is equipped with instrumentation for monitoring the operating conditions of the engine. Some preliminary data obtained to demonstrate the safety of the engine under expected hazardous operating conditions and to demonstrate the applicability of one-dimensional propulsion equations to calculate the thrust induced by the engine are shown. Additional data obtained to determine the system transfer function to design a PID controller are also shown. The PID control algorithm design has been outlined.

The instrumentation includes several thermocouples and pressure transducers and a load cell to measure the thrust generated by the engine. A valve controls the fuel-flow rate. In the present work two separate control approaches were used. First the difference between the desired thrust from the engine and the thrust measured using the load cell was used as the feedback signal to control the fuel-flow rate to the engine. In the second approach the temperature and pressure sensor data were used to calculate the thrust produced by the engine using the aero-thermodynamic equations applying to turbojet engine operations, and the difference between the calculated thrust and the desired thrust was used as the feedback signal. In the present approach, turbojet engine's operation will be automated and several control logics will be trimmed to show their capabilities. In this hardware-in-the-loop control demonstration effort, first a simple PID control algorithm is demonstrated. The testbed development and some preliminary results obtained are presented using the experimental apparatus.

Simply stated, the term "soft computing" here refers to computational mechanisms that can determine suitable relationships (in a system data set) to assess and determine a quantitative opinion(s) based on future conditions. Within MSFC, such computational mechanisms are viewed as a collection of algorithms that can achieve optimal or near-optimal results in the presence of imprecise data, uncertainty, unknown physics and probabilistic outcomes. Such algorithms include automated reasoning, nondeterministic or probabilistic methods. Examples of the latter include Bayesian networks, statistical resampling techniques, chaos theory and parts of learning theory. Other well-known soft computing technologies include fuzzy logic, neural networks and genetic algorithms. The term soft computing is used metaphorically to contrast with hard computing.

Hard computing systems are based on those traditional approaches used commonly in most event-driven systems. Such approaches are often viewed as crisp or binary. For example, in a propulsion system for engine start preparations, if liquid oxygen tank temperature $A < x$ and liquid oxygen bottom tank pressure $A > y$, then open liquid oxygen engine supply valve can be opened. For this example, soft computing would accommodate a region of acceptable temperature and pressure valves as well as observe other conditions such as liquid level and so on. A mechanism (e.g. NNs) for determining when to open the liquid oxygen engine supply valve would be used.

The approach would differ in that it would be tolerant of any imprecision and uncertainty. In essence, one could view soft computing as being similar to the way the human brain works. Humans tend to use heuristic (objective) and subjective knowledge before making decisions based on current states of events. The key features in soft computing stems from addressing any inherent imprecision, uncertainty, partial truths and overall system knowledge. The central goal in soft computing is to attain more robust response. For this effort, the primary technologies to be used are Bayesian belief networks and fuzzy logic. For the engine start-up sequence, the Bayesian belief networks will be used to ascertain the state of the engine prior to proceeding to main stage control. For main stage engine control approach, fuzzy logic will be employed and it is largely dependent on the complexity of the engine control requirements and functions.

17.6.1 Bayesian Belief Networks

For the engine start phase, the primary soft computing technology to be utilized is Bayesian belief networks (BBNs). The sole intent of the BBN is to qualify each of the states during engine start-up prior to reaching main

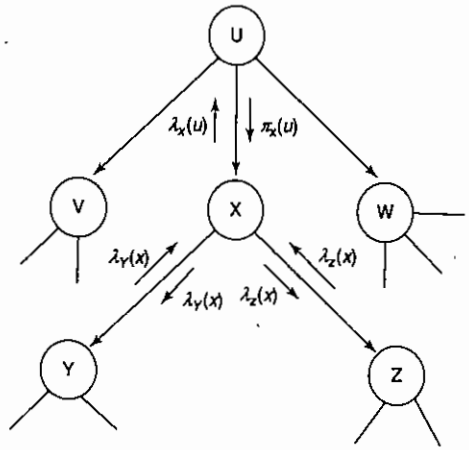


Figure 17-22 Bayesian belief network.

stage. This will further assure certainty in the health of the engine and proceeding into main stage in addition to providing added assurance into preventing any premature engine shutdowns. BBNs have been proven to be good predictive and diagnostic mechanisms for reasoning about the state of events in environments where uncertainty is universal. Suppressing the details, the genealogy of this SCT is strongly rooted in classic statistical Bayesian inference theory where a subjectivist viewpoint is taken. Figure 17-22 shows a Bayesian belief network.

In short, Bayesian inference uses a different interpretation of probability where one's degree of belief in some event is part of the reasoning. BBNs are computational architectures that permit declarative (prior conditional probabilistic values) and subjective opinions (posterior probabilistic values) about world (factual) knowledge to be part of the reasoning and assessment through a visual network representation and a unique syntactic message-passing feature.

1. *Belief updating*: When node X is activated to update its parameters for belief updating, it first inspects all messages transmitted to it by its parents (π) and its children nodes (λ). Then using all input, it updates its belief.
2. *Bottom-up propagation*: Using messages transmitted by Y and Z, compute message to transmit to parent node U.
3. *Top-down propagation*: Node X then computes new messages to be sent to its children nodes Y and Z.

17.6.2 Fuzzy Logic Control

For main stage control, the plan is to use fuzzy logic. The use of fuzzy logic is suitable in that it accommodates the uncertainties associated with control during power. Fuzzy logic is a branch of mathematics that deals with approximate reasoning.

Zadeh of the University of California at Berkeley combines the topics of multivalued logic, probability theory and artificial intelligence for simulation of human thought by using computer software as a medium. The technology of fuzzy logic enables a computer to make decision based on vagueness or imprecision intrinsic

in most physical systems. Fuzzy logic comprises sets and subsets where a set represents an input linguistic variable and its subsets represent the linguistic values.

A cook-book process is followed using fuzzification and defuzzification to determine a suitable response to conditions on any given system to be controlled. The centroid method (or center of gravity method) will be used for fuzzification. The intent in employing fuzzy logic with the SR-30 engine is to utilize all engine test data and conventional PID data to design and develop the fuzzy logic based controller.

Once the soft computing techniques of BBNs and fuzzy logic have been developed and tested and verified (off-line), both will be integrated into the SR-30 engine tested control environment for final integration and verification testing.

17.6.3 Software Engineering in Marshall's Flight Software Group

In every software development organization a set of processes and standards for their base product line are typically adhered too. Such processes and standards generally adhere to a type of software development life cycle. For the Flight Software Group the popular Waterfall Model is used. Furthermore, the Flight Software Group's process is ISO 9001 certified. And more importantly, has recently been certified as a CMM Level 3 organization, a first for any NASA organization. CMM is the Capability Maturity Model for software that was developed by Carnegie Melon's Software Engineering Institute and has become an internationally recognized standard for evaluating software development processes where a level 5 is the highest certification a software development organization can achieve. The principle function of the Flight Software Group is to develop flight critical software for embedded systems, hence requiring all software development processes to be stringent with software quality assurance functions underlying all activities of software development.

The Flight Software Group traditionally views software engineering as the establishment and uses sound engineering processes to develop reliable software, based on human processes and thinking, that works on real machines. Furthermore, software engineering is also viewed as the design and implementation of a set of user requirements into software using sound engineering processes. The emphasis here is that the Flight Software Group uses sound software development processes based on empirically proven and sound practices.

17.6.4 Experimental Apparatus and Facility Turbine Technologies SR-30 Engine

Soft computing technology hardware-in the-loop experiments were conducted using Turbine-Technologies model SR-30 turbojet engine shown in Figure 17-21. The demonstration engine consists of the turbojet engine manufactured by Turbine Technologies Ltd. in its custom enclosure. The enclosure includes a control panel for engine operation and monitoring and a PC-based data acquisition unit for measuring the engine operating conditions.

The SR-30 engine has a single-stage radial flow compressor with a maximum pressure ratio of $PR = 3.4$, single-stage axial-flow turbine, and reverse-flow annular combustion chamber and it operates obeying the Brayton thermodynamic cycle in the same fashion as the large turbojet engines. The engine as produced by the Turbine Technologies includes many pressure and temperature sensors, a load-cell for thrust measurements, a custom motor winding for reading the engine rpm and a fuel flow-rate measurement system to monitor/measure the operating parameters of the engine. The engine generates 20 lbs of thrust at 90,000 rpm while ingesting $m = 1.1$ lb s^{-1} of air. The engine has a length of 10.75 in. and the exit exhaust diameter of Dexit = 2.25 in.

The engine available is instrumented with pressure transducers in the compressor inlet and exit, in the combustor, in the turbine exit and in the thrust nozzle exit, and Krype thermocouples in the compressor inlet and exit, in the turbine inlet and exit and in the thrust nozzle exit. The engine available was also equipped

with a National Instruments (NI) PCI 4351, A/D board with 24 bit resolution for 16 analog inputs with a 60 samples s^{-1} capability, and a NI Virtual Bench Logger data acquisition program for monitoring the measured parameters on a PC.

Starting the engine requires an external source of high-pressure air at minimum 100 psi to spin up the engine to approximately 10,000 rpm. Subsequent fuel injection and ignition starts the engine. The fuel-flow rate is controlled by the person operating the engine with the use of a lever, which basically controls a valve constricting the fuel flow to the engine. Engine idles at approximately 50,000 rpm and the thrust generated increases with the increased rpm. To obtain higher thrust values the engine operator steadily increases the fuel-flow rate from the idle conditions. In order to stop the engine it is brought to the idle conditions and run until the exhaust temperature drops under 100 °C, to minimize engine damage.

17.6.5 System Modifications

Turbine Technologies data acquisition system as purchased and used for classroom demonstrations is not sufficiently fast enough for use with the hardware-in-the-loop control algorithms. Since one of the main scopes of the present work is to implement and demonstrate different control algorithms in controlling a turbojet engine thrust, a new data acquisition system and software has been implemented into the existing system to increase the data acquisition speed and to increase the control capability. Additionally, the available system was designed and used to collect and present data and it did not have provisions to send signals via computer for closed-loop control applications. Changes implemented include the replacement of the data acquisition board, connection panels for the sensors, addition of a low flow-rate fuel-flow rate measurement unit, a fast acting linear servo-controller.

17.6.6 Fuel-Flow Rate Measurement System

The SR-30 engine as produced by the Turbine Technologies uses a pressure transducer together with a calibration curve to determine the fuel-flow rate to the engine. The pressure values read on the fuel line are plotted against the fuel flow spent and also against the rpm of the engine to generate pressure vs. the fuel rate and the pressure vs. the rpm calibration curves for long term operations. However, for the present purposes, since the fuel-flow rate is mainly the only control input to control the desired thrust of the engine a more accurate and faster fuel-flow rate measurement device has to be implemented.

17.6.7 Exit Conditions Monitoring

Turbojet propulsion equations used in calculation of the engine thrust requires the measurement of the exit conditions, namely the exhaust total pressure and the total temperature. Although the existing system available from Turbine Technologies incorporated a pressure transducer and a thermocouple, for this purpose, the response time for the equipment was rather slow. In order to increase the time resolution of the data obtained at the exit conditions a new pressure transducer with a 0.2 ms response time and a 0.1 s response time has been incorporated.

As a result of this effort, new insight has been gained into the behavior and application of soft computing technologies in a rocket engine control environment. The methodology created here will provide a new approach to the area of employing soft computing technologies in rapid response engine control systems for future vision vehicles. It will yield better insight into incorporating soft computing technologies with proven and practical software engineering methods. It is expected that this effort will demonstrate that by

employing soft computing technologies, issues in quality and reliability of the overall scheme of engine controller development can be further improved and thus safety be further insured.

Furthermore, the use of these soft computing technologies is expected to supplement efforts in improving software management, software development time, software maintenance, processor execution, fault tolerance and mitigation and nonlinear control in power level transitions, all of which contribute to a better engine control system. It is projected that the final product will yield a foundation for a path to further development of an alternative low cost engine controller that would be capable of performing in unique vision spacecraft vehicles requiring low cost and advanced avionics architectures for autonomous operations from engine pre-start to engine shutdown.

17.7 Summary

In this chapter we have dealt with the applications of soft computing techniques. The application areas of these soft computing techniques are growing day by day. Neural networks and fuzzy logic are effectively used in various control applications. Genetic algorithm plays a major role in providing solutions for optimizing a problem. The combinations of all these techniques give an accurate solution to complex systems. There are various researches going around the world in the field of soft computing.

17.8 Review Questions

1. State the various applications of neural networks.
2. Mention the application areas of fuzzy logic.
3. In what areas does genetic algorithm gives a best optimized solution?
4. List few applications of hybrid fuzzy GA systems and neurofuzzy systems.
5. Soft computing techniques gives best solution to complex problems. Justify.
6. With suitable case study, explain how neural network best performs its control action.
7. Explain the application of fuzzy logic systems to image processing applications.
8. Describe in detail the application of genetic algorithm to Civil Engineering area.
9. With suitable block diagram, explain the principle involved in a liquid level controller using neurofuzzy technique.
10. With a case study example, describe in detail the application of soft computing.

17.9 Exercise Problems

1. Write a program for implementing genetic algorithm based Internet search technique.
2. Write a program for processing an image of size 16 × 16 using neural networks and fuzzy logic.
3. Build a 3D game using the design issues in genetic algorithm.
4. Implement a water flow management system and water treatment system using soft computing approaches.
5. Construct a neural network training controller for controlling the motion of a satellite.

6. Write a program for analyzing the landing of an aircraft using fuzzy logic methodology.
7. Implement robot motion control using neuro-fuzzy controller.
8. Write a program using genetic algorithm to solve a traveling saleman problem.
9. Implement with any example, the concept involved in parallel genetic algorithm.
10. Write a program for controlling the motion of an inverted pendulum using neural networks and fuzzy logic.

Soft Computing Techniques Using C and C++

18

Learning Objectives

- Gives the source codes for soft computing techniques in C and C++.
- Neural network implementation is performed for perceptron network, Madaline net, BPN, CPN, ART and Kohonen self-organizing feature maps.
- In fuzzy logic, the implementation is carried out for primitive operations of classical sets and fuzzy sets.
- The Cartesian products of two given fuzzy sets, max min composition for fuzzy relations are also implemented in C and C++ to enhance the reading of fuzzy logic concept.
- Few problems of maximizing and minimizing a function, traveling salesman problem, prisonner's dilemma, quadratic equation solving are implemented in the universal language to depict the genetic algorithm operation.

18.1 Introduction

This chapter gives the source codes for implementation of Soft Computing Techniques using the languages C and C++. C is a general-purpose structured programming language that is powerful, efficient and compact. It combines the features of a high-level language with the elements of the assembler and thus is close to man and machine. Programs written in C are very efficient and fast. C++ on the other hand is an object-oriented language that a C programmer can appreciate, especially who is an early age assembly language programmer. C++ orients toward execution performance and then toward flexibility. The name C++ signifies the evolutionary nature of the changes from C. Thus Soft Computing being an approach based on evolutionary strategies and evolutionary programming can be implemented using the structured programming and object programming languages. This chapter discusses few problems solved using C/C++.

18.2 Neural Network Implementation

The various neural networks discussed through Chapters 2–6 are implemented using C and C++ languages in this section. The source code for each network for a specific application is given below.

18.2.1 Perceptron Network

The program for perceptron network is as follows:

```
/*PERCEPTRON*/
#include<stdio.h>
#include<conio.h>
main()
{
```

```
  signed int x[4][2],tar[4];
  float w[2],wc[2],out=0;
  int i,j,k=0,h=0;
  float s=0,b=0,bc=0,alpha=0;
  float theta;
  clrscr();
  printf("Enter the value of theta & alpha");
  scanf("%f%f",&theta,&alpha);
for(i=0;i<=3;i++)
{
  printf("Enter the value of %d Inputrow & Target",i);
  for(j=0;j<=1;j++)
  {
  scanf("%d",&x[i][j]);
  }
  scanf("%d",&tar[i]);
  w[i]=0;
  wc[i]=0;
}
printf("\Net\t Target\tWeight changes\tNew weights\t Bias changes\tBias\n");
printf("-------------------------------------------------------\n");
mew:
  printf("ITERATION %d\n",h);
  printf("-------------------------------------------------------\n");
  for(i=0;i<=3;i++)
  {
    for(j=0;j<=1;j++)
    {
      s+=(float)x[i][j]*w[j];
    }
    s+=b;
    printf("%.2f\t",s);
      if(s>theta)
      out=1;
      else if(s<-theta)
      out=-1;
      else
      {
      out=0;
      }
      printf("%d\t",tar[i]);
    s=0;
    if(out==tar[i])
    {
      for(j=0;j<=1;j++)
      {
      wc[j]=0;
      bc=0;
      printf("%.2f\t",wc[j]);
      }
```



```
      for(j=0;j<=1;j++)
      printf("%.2f\t",w[j]);
      k+=1;
      b+=bc;
      printf("%.2f\t\t",bc);
      printf("%.2f\t",b);
    }
    else
    {
      for(j=0;j<=1;j++)
      {
      wc[j]=x[i][j]*tar[i]*alpha;
      w[j]+=wc[j];
      printf("%.2f\t",wc[j]);
      wc[j]=0;
      }
      for(j=0;j<=1;j++)
      printf("%.2f\t",w[j]);
      bc=tar[i]*alpha;
      b+=bc;
      printf("%.2f\t\t",bc);
      printf("%.2f\t",b);
    }
    printf("\n");
    }
    if(k==4)
    {
    printf("\nFinal weights\n");
    for(j=0;j<=1;j++)
    {
      printf("w[%d]=%.2f\t",j,w[j]);
    }
    printf("Bias b=%.2f",b);
    }
    else
    {
    k=0;
    h=h+1;
    getch();
    goto mew;
    }
    getch();
    }
```

18.2.2 Adaline Network

The program for adaline network is as follows:

```
/*ADALINE*/
#include<stdio.h>
#include<conio.h>
```

```
main()
{
  signed int x[4][4],tar[4];
  float wc[4],w[4],e=0,er=0,yin=0,alp=0.5,b=0,bc=0,t=0;
  int i,j,k,q=1;
  clrscr();
  for(i=0;i<=3;i++)
  {
    printf("\nEnter the %d row and target\t",i);
    for(j=0;j<=3;j++)
    {
      scanf("%d",&x[i][j]);
    }
    scanf("%d",&tar[i]);
    printf("%d",tar[i]);
    w[i]=0.0;
    wc[i]=0.0;
  }
  mew:
  er=0;e=0;
  yin=0;
  printf("\n ITERATION%d",q);
  printf("\n-----------------");
  for(i=0;i<=3;i++)
  {
  t=tar[i];
  for(j=0;j<=3;j++)
  {
  yin=yin+x[i][j]*w[j];
  }
  b=b+bc;
  yin=yin+b;
  bc=0.0;
  printf("\nNet=%f\t",yin);
  e=(float)tar[i]-yin;
  yin=0.0;
  printf("Error=%f\t",e);
  printf("Target=%d\t\n",tar[i]);
  er=er+e*e;
  for(k=0;k<=3;k++)
  {
  wc[k]=x[i][k]*e*alp;
  w[k]+=wc[k];
  wc[k]=0.0;
  }
  printf("Weights \t");
  for(k=0;k<=3;k++)
  {
  printf("%f\t",w[k]);
  }
```



```
  bc=e*alp;
  printf("b=%.2f\t",b);
  getch();
  printf("\n Error Square=%f",er);
  if(er<=1.000)
  {
    printf("\n");
    for(k=0;k<=1;k++)
    printf("%f\t",w[k]);
    getch();
  }
  else
  {
    e=0;
    er=0;
    yin=0;
    q=q+1;
    goto mew;
  }
  getch();
  }
  }
```

18.2.3 Madaline Network for XOR Function

The program is as follows:

```
//XOR function using madaline
#include<stdio.h>
#include<conio.h>
void main()
{
  signed int x[4][2],tar[4];
  float w[2][2],a,o[2];
  float wc[2][2],zin[2],z1=0,z2=0,yin=0,b[2],er=0,b3=0,v1=0,v2=0.5;
  int i,j,c=0,in,d;
  float bc[2];
  float alp=0.5;
  clrscr();
  for(i=0;i<=3;i++)
  {
    printf("Enter the %d row & target:");
    for(j=0;j<=1;j++)
      scanf("%d",&x[i][j]);
    scanf("%d",&tar[i]);
  }
  getch();
  printf("Enter Weights:");
  for(i=0;i<=1;i++)
  {
```

```
  for(j=0;j<=1;j++)
  {
    scanf("%f",&a);
    w[i][j]=a;
    wc[i][j]=0;
  }
  printf("bias");
  scanf("%f",&b[i]);
  zin[i]=0;
}
mew:
printf("Iteration\n");
printf("--------------\n");
for(i=0;i<=3;i++)
{
  for(in=0;in<=1;in++)
  {
    for(j=0;j<=1;j++)
    {
      zin[in]+=x[i][j]*w[j][c];
    }
    zin[in]+=b[in];
    printf("zin%d= %.3f\t",in,zin[in]);
    c+=1;
  }
  c=0;
  d=1;
  if(zin[c]>=0 & zin[d]>=0)
    z1=z2=1;
  else if(zin[c]>=0 & zin[d]<=0)
  {
    z1=1;
    z2=-1;
  }
  else if(zin[c]<=0 & zin[d]>=0)
  {
    z1=-1;
    z2=1;
  }
  else
  {
    z1=z2=-1;
  }
yin=z1*v1+z2*v2+b3;
printf("NET %.3f\t",yin);
  for(in=0;in<=1;in++)
  {
    o[in]=tar[i]-zin[in];
    er+=o[in]*o[in];
    zin[in]=0;
```



```
  }
  if(yin==tar[i])
  {
    for(in=0;in<=1;in++)
    {
      for(j=0;j<=1;j++)
      {
        wc[in][j]=0;
        w[in][j]+=wc[in][j];
      }
    bc[in]=0;
    }
    yin=0;
  }
  else
  {
    for(in=0;in<=1;in++)
    {
      for(j=0;j<=1;j++)
      {
        wc[in][j]=alp*o[j]*x[i][in];
        printf("wc%d%d=%.3f\t",in,j,wc[in][j]);
        w[in][j]+=wc[in][j];
        printf("w=%.3f\t",w[in][j]);
        wc[in][j]=0;
      }
    }
    for(in=0;in<=1;in++)
    {
      bc[in]=alp*o[in];
      b[in]+=bc[in];
      printf("\nb%d=%.3f",in,b[in]);
    }
    for(in=0;i<=1;in++)
 {
   bc[in]=0;
 }
   yin=0;
 }
   printf("\n");
 }
 if(er<=1)
 {
   for(i=0;i<=1;i++)
   {
     for(j=0;j<=1;j++)
     printf("%.3f",w[i][j]);
   }
 }
else
```

```
    {
      yin=0;
      for(in=0;in<=1;in++)
      {
        bc[in]=0;
        zin[in]=0;
      }
      er=0;
      getch();
      goto mew;
      }
    getch();
}
```

18.2.4 Back Propagation Network for XOR Function using Bipolar Inputs and Binary Targets

In this case the assumption is made for the necessary parameters. The initial weights and bias are assumed to be of small random values. The program is as follows:

```
/*BACK PROPAGATION NETWORK*/
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<stdlib.h>
void main()
{
  float v[2][4],w[4][1],vc[2][4],wc[4][1],de,del[4],bl,bia,bc[4],e=0;
  float x[4][2],t[4],zin[4],delin[4],yin=0,y,dy,dz[4],b[4],z[4],
        es,alp=0.02;
  int i,j,k=0,itr=0;
  v[0][0]=0.1970;
  v[0][1]=0.3191;
  v[0][2]=-0.1448;
  v[0][3]=0.3594;
  v[1][0]=0.3099;
  v[1][1]=0.1904;
  v[1][2]=-0.0347;
  v[1][3]=-0.4861;
  w[0][0]=0.4919;
  w[1][0]=-0.2913;
  w[2][0]=-0.3979;
  w[3][0]=0.3581;
  b[0]=-0.3378;
  b[1]=0.2771;
  b[2]=0.2859;
  b[3]=-0.3329;
  bl=-0.141;
  x[0][0]=-1;
```



```
  x[0][1]=-1;
  x[1][0]=-1;
  x[1][1]=1;
  x[2][0]=1;
  x[2][1]=-1;
  x[3][0]=1;
  x[3][1]=1;
  t[0]=0;
  t[1]=1;
  t[2]=1;
  t[3]=0;
  clrscr();
  for(itr=0;itr<=387;itr++)
  {
    e=0;
    es=0;
    for(i=0;i<=3;i++)
    {
      do
      {
        for(j=0;j<=1;j++)
        {
          zin[k]+=x[i][j]*v[j][k];
        }
        zin[k]+=b[k];
        k+=1;
      }while(k<=4);
      for(j=0;j<=3;j++)
      {
        z[j]=(1-exp(-zin[j]))/(1+exp(-zin[j]));
        dz[j]=((1+z[j])*(1-z[j]))*0.5;
      }
      for(j=0;j<=3;j++)
      {
        yin+=z[j]*w[j][0];
      }
      yin+=bl;
      y=(1-exp(-yin))/(1+exp(-yin));
      dy=((1+y)*(1-y))*0.5;
      de=(t[i]-y)*dy;
      e=t[i]-y;
      es+=0.5*(e*e);
    for(j=0;j<=3;j++)
    {
      wc[j][0]=alp*de*z[j];
      delin[j]=de*w[j][0];
      del[j]=delin[j]*dz[j];
    }
    bia=alp*de;
    for(k=0;k<=1;k++)
```

```
    {
      for(j=0;j<=3;j++)
      {
      vc[k][j]=alp*del[j]*x[i][k];
      v[k][j]+=vc[k][j];
      }
    }
    for(j=0;j<=3;j++)
    {
      bc[j]=alp*del[j];
      w[j][0]+=wc[j][0];
      b[j]+=bc[j];
    }
    bl+=bia;
    for(j=0;j<=3;j++)
    {
      zin[j]=0;
      z[j]=0;
      dz[j]=0;
      delin[j]=0;
      del[j]=0;
      bc[j]=0;
    }
    k=0;yin=0;y=0;
    dy=0;bia=0;de=0;
    }
    printf("\nEpoch %d:\n",itr);
    for(k=0;k<=1;k++)
    {
      for(j=0;j<=3;j++)
      {
        printf("%f\t",v[k][j]);
      }
      printf("\n");
    }
    printf("\n");
    for(k=0;k<=3;k++)
    {
      printf("%f\t",w[k][0]);
    }
    printf("\n%f",b1);
    printf("\t");
    for(k=0;k<=3;k++)
    {
      printf("%f\t",b[k]);
    }
    getch();
    }
    getch();
}
```


18.2.5 Kohonen Self-Organizing Feature Map

The user can enter the inputs and initialize the weights of his wish.

```
//A KSOM to cluster four input vectors
#include<stdio.h>
#include<conio.h>
void main()
{
  signed int x[4][2];
  float w[4][2],d1,d2,o,m=0;
  int i,j,k,J;
  float alp=0.6;
  clrscr();
  printf("Enter the input:");
  for(i=0;i<=3;i++)
  {
    for(j=0;j<=3;j++)
    {
      scanf("%d",&x[i][j]);
    }
  }
  printf("Enter the Weight matrix:");
  for(i=0;i<=3;i++)
  {
    for(j=0;j<=1;j++)
    {
      scanf("%f",&o);
      w[i][j]=0;
    }
  }
  mew:
  for(i=0;i<=3;i++)
  {
    for(k=0;k<=1;k++)
    {
      for(j=0;j<=3;j++)
      {
        if(k==0)
          d1+=(w[j][k]-x[i][j])*(w[j][k]-x[i][j]);
        else
          d2+=(w[j][k]-x[i][j])*(w[j][k]-x[i][j]);
      }
    }
    if(d1>d2)
      J=1;
    else
      J=0;
      d1=d2=0;
    for(j=0;j<=3;j++)
```

```
    {
      w[j][J]+=alp*(x[i][j]-w[j][J]);
    }
    getch();
    }
    alp=alp/1.014;
    if(m>=100)
    {
      for(i=0;i<=3;i++)
        {
          for(j=0;j<=1;j++)
          {
            printf("\n%f\t",w[i][j]);
          }
          getch();
        }
    }
    else
    {
      m=m+1;
      for(i=0;i<=3;i++)
      {
      for(j=0;j<=1;j++)
      {
      printf("\n%f\t",w[i][j]);
      }
      printf("\n");
      }
    getch();
    goto mew;
    }
  getch();
}
```

18.2.6 ART 1 Network with Nine Input Units and Two Cluster Units

The program is as follows:

```
/* AN ART1 NET WITH 9 INPUT UNITS AND 2 CLUSTER UNIT */
#include<stdio.h>
#include<conio.h>
main()
  {
  float ro;
  float b[9][3],t[2][9],s[9],x[9],sin=0,y[2],xin=0;
  int i,j,k=0,J,c=0;
  y[0]=0;
  y[1]=0;
  b[0][0]=0.33;b[1][0]=0.0;b[2][0]=0.33;
  b[3][0]=0.0;b[4][0]=0.33;b[5][0]=0.0;
```



```
b[6][0]=0.33;b[7][0]=0.0;b[8][0]=0.33;

b[0][1]=0.1;b[1][1]=0.1;b[2][1]=0.1;
b[3][1]=0.1;b[4][1]=0.1;b[5][1]=0.1;
b[6][1]=0.1;b[7][1]=0.1;b[8][1]=0.1;
t[0][0]=1.0;t[0][1]=0;t[0][2]=1.0;t[0][3]=0;
t[0][4]=1.0;t[0][5]=0;t[0][6]=1.0;t[0][7]=0;
t[0][8]=1.0;

t[1][0]=1;t[1][1]=1;t[1][2]=1;t[1][3]=1;
t[1][4]=1;t[1][5]=1;t[1][6]=1;t[1][7]=1;
t[1][8]=1;

clrscr();
mew:
  printf("Enter the value of ro\n");
  scanf("%f",&ro);
  printf("Enter the input value\n");
  for(i=0;i<=8;i++)
  {
    scanf("%f",&s[i]);
    x[i]=s[i];
  }
  sin=s[0]+s[1]+s[2]+s[3]+s[4]+s[5]+s[6]+s[7]+s[8];
  for(i=0;i<=1;i++)
  {
    do
    {
    y[i]+=s[k]*b[k][i];
    k+=1;
    }while(k<=8);
    k=0;
  }
  for(i=0;i<=1;i++)
    printf("\tyin=%f",y[i]);
    if(y[0]>=y[1])
      J=0;
    else
      J=1;
    printf("J=%d",J);
    me:
    for(i=0;i<=8;i++)
    {
      x[i]=s[i]*t[J][i];
    }
    xin=x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7]+x[8];
      if((xin/sin)>= ro)
      {
        for(i=0;i<=8;i++)
        {
```

```
            b[i][J]=(2*x[i])/(1+xin);
            t[J][i]=x[i];
          }
        }
        else
        {
          y[J]=-1;
          xin=0;
          goto me;
        }
        printf("\nBottom up weights\n");
      for(i=0;i<=8;i++)
      {
      for(j=0;j<=1;j++)
      {
        printf("%f\t",b[i][j]);
      }
        printf("\n");
      }
    printf("\nTop down Weights\n");
    for(i=0;i<=1;i++)
    {
      for(j=0;j<=8;j++)
    {
      printf("%f\t",t[i][j]);
    }
    printf("\n");
  }
  getch();
  y[0]=0;
  y[1]=0;
  y[2]=0;
  sin=xin=0;
  c+=1;
  k=0;
  if(c<=2)
    goto mew;
    getch();
  }
```

18.2.7 ART 1 Network to Cluster Four Vectors

The program is as follows:

```
/* ART1 NETWORK TO CLUSTER FOUR VECTORS */
#include<stdio.h>
#include<conio.h>
main()
{
  float n=4.0,m=3.0,o=0.4,l=2.0;
```



```
float b[4][3],t[3][4],s[4],x[4],sin=0,y[3],xin=0;
int i,j,k=0,J,c=0;
y[0]=0,y[1]=0,y[2]=0;
clrscr();
for(i=0;i<=3;i++)
{
  for(j=0;j<=2;j++)
  {
    b[i][j]=0.2;
  }
}
for(i=0;i<=2;i++)
{
  for(j=0;j<=3;j++)
  {
    t[i][j]=1.0;
  }
}
mew:
printf("Enter the input value:\n");
for(i=0;i<=3;i++)
{
  scanf("%f",&s[i]);
  x[i]=s[i];
  sin+=s[i];
}
for(i=0;i<=2;i++)
{
  printf("\nY");
  do
    {
      y[i]+=s[k]*b[k][i];
      k+=1;
    }while(k<=3);
  if(y[0]>=y[1])
  {
    if(y[0]>=y[2])
      J=0;
    else
      J=2;
    }
    else
    {
      if(y[1]>=y[2])
        J=1;
      else
        J=2;
    }
    for(i=0;i<=3;i++)
    {
```

```
        x[i]=s[i]*t[J][i];
        xin+=x[i];
      }
      if(xin/sin>=0.4)
      {
        for(i=0;i<=3;i++)
      {
          b[i][J]=(2*x[i])/(1+xin);
          t[J][i]=x[i];
        }
      }
      else
      {
      y[J]=-1;
      }
      printf("\n");
      for(i=0;i<=3;i++)
      {
        for(j=0;j<=2;j++)
        {
          printf("%f\t",b[i][j]);
        }
        printf("\n");
      }
      for(i=0;i<=2;i++)
      {
        for(j=0;j<=3;j++)
        {
          printf("%f\t",t[i][j]);
        }
      printf("\n");
    }
    getch();
    y[0]=y[1]=y[2]=0;
    sin=xin=0;
    c+=1;
    k=0;
    if(c<=3)
    goto mew;
    }
  getch();
  }
```

18.2.8 Full Counterpropagation Network

The program is as follows:

```
/* FULL COUNTER*/
#include<stdio.h>
#include<conio.h>
```



```
void main ()
{
  float alp=0.6,x=0.1,n[10],v[1][10],d[10],p,w[1][10],y,bet=0.6;
  float u[10][1],t[10][1],a=0.6,b=0.6;
  int i,j,J,k=0,m;
  clrscr();
  v[0][0]=0.1;v[0][1]=0.15;v[0][2]=0.2;
  v[0][3]=0.3;v[0][4]=0.5;v[0][5]=1.5;
  v[0][6]=3.0;v[0][7]=5.0;v[0][8]=7.0;
  v[0][9]=9.0;
  w[0][0]=9.0;w[0][1]=7.0;w[0][2]=5.0;
  w[0][3]=3.0;w[0][4]=1.5;w[0][5]=0.5;
  w[0][6]=0.2;w[0][7]=0.2;w[0][8]=0.15;
  w[0][9]=0.1;
  u[0][0]=0.1;u[1][0]=0.15;u[2][0]=0.2;
  u[3][0]=0.3;u[4][0]=0.5;u[5][0]=1.5;
  u[6][0]=3.0;u[7][0]=5.0;u[8][0]=7.0;
  u[9][0]=9.0;
  t[0][0]=9.0;t[1][0]=7.0;t[2][0]=5.0;
  t[3][0]=3.0;t[4][0]=1.5;t[5][0]=0.5;
  t[6][0]=0.3;t[7][0]=0.2;t[8][0]=0.15;
  t[9][0]=0.1;
  do
  {
    y=1/x;
    printf("\n");
    for(j=0;j<=9;j++)
    {
    n[j]=(x-v[0][j])*(x-v[0][j])+(y-w[0][j])*(y-w[0][j]);
    d[j]=n[j];
    }
    for(m=0;m<=9;m++)
    {
      for(j=m;j<=9;++j)
      {
        if(d[k]>d[j])
        {
          p=d[j];
          d[j]=d[k];
          d[k]=p;
        }
      }
        k+=1;
  }
  for(j=0;j<=9;j++)
  {
    if(d[0]==n[j])
    {
      J=j;
    }
```

```
}
v[0][J]+=alp*(x-v[0][J]);
w[0][J]+=bet*(y-w[0][J]);
printf("\nInput X=%f",x);
printf("\nUpdated weights: v");
for(j=0;j<=9;j++)
{
  printf("%f\t",v[0][j]);
  n[j]=0;
  d[j]=0;
}
printf("\n Updated weights: w");
for(j=0;j<=9;j++)
{
  printf("%f\t",w[0][j]);
}
x=x+0.5;
alp=alp/1.014;
bet=bet/1.014;
J=0;
k=0;
getch();
}
while(x<=10.50);
x=0.1;
do
{
  for(j=0;j<=9;j++)
  {
    n[j]=(x-v[0][j])*(x-v[0][j])+(y-w[0][j])*(y-w[0][j]);
    d[j]=n[j];
  }
  for(m=0;m<=9;m++)
  {
    for(j=m;j<=9;j++)
    {
      if(d[k]>d[j])
      {
        p=d[j];
        d[j]=d[k];
        d[k]=p;
      }
    }
    k+=1;
  }
  for(j=0;j<=9;j++)
  {
    if(d[0]==n[j])
    {
      J=j;
```



```
    }
  }
  u[J][0]+=a*(y-u[J][0]);
  t[J][0]+=b*(x-t[J][0]);
  printf("\n Input=%f",x);
  printf("\n Updated wights u:");
  for(j=0;j<=9;j++)
  {
  printf("%f\t",u[j][0]);
  n[j]=0;
  d[j]=0;
  }
  printf("\nUpdated weights t:");
  for(j=0;j<=9;j++)
  {
    printf("%f\t",t[j][0]);
  }
  k=0;
  J=0;
  a=a/1.014;
  b=b/1.014;
  x=x+0.5;
  y=1/x;
  getch();
}while(x<=10.5);
getch();
}
```

18.3 Fuzzy Logic Implementation

The various concepts of fuzzy logic through chapters Chapters 7–14 are implemented using C and C++ languages in this section. The source code for the same is as given below.

18.3.1 Implement the Various Primitive Operations of Classical Sets

The program is as follows:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<alloc.h>
struct SET
{
  char *elts;
  int n;
};
typedef struct SET set;
set s;

void getval(set m,char x)
{
  int i;
```

```
    printf("\n Enter the %c:\n",x);
    for(i=0;i2*s.n)
    {
      printf("\n Invalid values");
      getch();
      exit(0);
    }
    a.elts=(char *) malloc(a.n);
    b.elts=(char *) malloc(b.n);
    getval(s,'S');
    getval(a,'A');
    getval(b,'B');

    while(1)
    {
      printf("\n Menu:\n 1.AUB\n 2.A B\n 3.A~\n 4.B~   \n 5.Print
          S,A,B\n 6.Exit");
      switch((ch=getch()))
      {
        case '1':
          ans=unionset(a,b);
          printval(ans,'U');
          getch();
          break;
        case '2':
          ans=interset(a,b);
          printval(ans,'^');
          getch();
          break;
        case '3':
          ans=complement(a);
          printval(ans,'a');
          getch();
          break;
        case '4':
          ans=complement(b);
          printval(ans,'b');
          getch();
          break;
        case '5':
          printval(s,'S');
          printval(a,'A');
          printval(b,'B');
          getch();
          break;
        case '6':
          exit(0);
      }
    }
}
```


Output

```
Enter the no of elts in sample space:5.
Enter the no of elts in A:3
Enter the no of elts in B:2

Enter the S:
Element 1:1
Element 2:2
Element 3:3
Element 4:4
Element 5:5

Enter the A:
Element 1:1
Element 2:2
Element 3:3

Enter the B:
Element 1:3
Element 2:4

Menu:
1.AUB
2.A^B
3.A~
4.B~
5.Print S,A,B
6.Exit

 U={1,2,3,4}
^ ={3}
a={4,5}
b={1,2,5}
S={1,2,3,4,5}
A={1,2,3}
B={3,4}
```

18.3.2 To Verify Various Laws Associated with Classical Sets

The program is as follows:

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<stdlib.h>

struct SET
{
  char *elts;
  int n;
};
```

```
  typedef struct SET set;
  set s;

void getval(set m,char *x)
{
  int i;
  printf("\n Enter the %s:\n",x);
  for(i=0;i3*s.n)
  {
    printf("\n Invalid values");
    getch();
    exit(0);
  }
  a.elts=(char *) malloc(a.n);
  b.elts=(char *) malloc(b.n);
  c.elts=(char *) malloc(a.n);
  getval(s,"S");
  getval(a,"A");
  getval(b,"B");
  getval(c,"C");
  clrscr();
    printf("\n Menu: \n 1.DeMorgan's Law\
      \n 2.Associative Law\
      \n 3.Distributive Law\
      \n 4.Commutative Law\
      \n 5.Exit");
while(1)
{
  switch((ch=getch()))
  {
    case '1':
      clrscr();
      printf("\n DeMorgan's Law:(A B)~=A~UB~");
      t1=intersect(a,b);
      printval(t1,"A^B");
      t2=complement(t1);
      printval(t2,"(A^B)~");

      t1=complement(a);
      printval(t1,"A~");
      t4=complement(a);
      t2=complement(b);
      printval(t2,"B~");
      ans=unionset(t1,t2);
      printval(ans,"A~UB~");

      printf("\n DeMorgan's Law:(AUB)~=A~ ^ B~");
      t1=unionset(a,b);
      printval(t1,"AUB");
      ans=complement(t1);
```



```
      printval(t2,"(AUB)~");

      t1=complement(a);
      printval(t1,"A~");
      t2=complement(b);
      printval(t2,"B~");
      ans=intersect(t1,t2);
      printval(ans,"A~ ^ B~");
      break;
    case '2':
      clrscr();
      printf("\n Associative Law:(A^B)^C = A^(B^C)");
      t1=intersect(a,b);
      printval(t1,"A^B");
      t2=intersect(t1,c);
      printval(t2,"(A^B)^C");

      t1=intersect(b,c);
      printval(t1,"B^C");
      t2=intersect(t1,a);
      printval(t2,"A^(B^C)");
      printf("\n Associative Law:(AUB)UC = AU(BUC)");
      t1=unionset(a,b);
      printval(t1,"AUB");
      t2=unionset(t1,c);
      printval(t2,"(AUB)UC");

      t1=unionset(b,c);
      printval(t1,"BUC");
      t2=unionset(t1,a);
      printval(t2,"AU(BUC)");
      break;
    case '3':
      clrscr();
      printf("\n Distributive Law:(AUB)^C = (A^B) U (A^C)");
      t1=unionset(a,b);
      printval(t1,"AUB");
      t2=intersect(t1,c);
      printval(t2,"(AUB)^C");

      t1=intersect(a,b);
      printval(t1,"A^B");
      t2=intersect(a,c);
      printval(t1,"A^C");
      ans=unionset(t1,t2);
      printval(ans,"(A^B)U(A^C)");

      printf("\n Distributive Law:(A^B)U C=(AUB)^(AUC)");
      t1=intersect(a,b);
      printval(t1,"A^B");
```

```
        t2=unionset(t1,c);
        printval(t2,"(A^B)UC");

        t1=unionset(a,b);
        printval(t1,"AUB");
        t2=unionset(a,c);
        printval(t1,"AUC");
        t2=intersect(t1,t2);
        printval(t2,"(AUB)^(AUC)");
        break;
      case '4':
        printf("\n Commutative Law: AUB=BUA");
        t1=unionset(a,b);
        printval(t1,"AUB");
        t1=unionset(b,a);
        printval(t1,"BUA");
        printf("\n Commutative Law: A^B=B^A");
        t1=intersect(a,b);
        printval(t1,"A^B");
        t1=intersect(b,a);
        printval(t1,"B^A");
        break;
      case '5':
        exit(0);
      default:
        putch('\a');
      }
      putch('\n');
      printval(s,"S");
      printval(a,"A");
      printval(b,"B");
      printval(c,"C");
      getch();
    }
}
```

Output

```
Enter the no of elts in sample space:3
Enter the no of elts in A:2
Enter the no of elts in B:2
Enter the no of elts in C:2
Enter the S:
Element 1:1
Element 2:2
Element 3:3

Enter the A:
Element 1:1
Element 2:2
```



```
Enter the B:
Element 1:2
Element 2:3

Enter the C:
Element 1:1
Element 2:3

Menu:
1.DeMorgan's Law
2.Associative Law
3.Distributive Law
4.Commutative Law
5.Exit

DeMorgan's Law:(A^B)~=A~UB~
A^B ={2}
(A^B)~ ={1,3}
A~ ={3}
B~ ={1}
A~UB~ ={1}
DeMorgan's Law:(AUB)~=A~ ^ B~
AUB ={1,2,3}
(AUB)~ ={1}
A~ ={3}
B~ ={1}
A~ ^ B~ ={1}

Associative Law:(A^B)^C = A^(B^C)
A^B ={2}
(A^B)^C ={}
B^C ={3}
A^(B^C) ={}

Associative Law:(AUB)UC = AU(BUC)
AUB ={1,2,3}
(AUB)UC ={1,2,3}
BUC ={2,3,1}
AU(BUC) ={2,3,1}

Distributive Law:(AUB)^C = (A^B) U (A^C)
AUB ={1,2,3}
(AUB)^C ={1,3}
231 ={2}
A^C ={1}
(A^B)U(A^C) ={1}

Distributive Law:(A^B)U C=(AUB)^(AUC)
A^B ={2}
(A^B)UC ={2,1,3}
```

```
AUB ={1,2,3}
AUC ={1,2,3}
(AUB)^(AUC) ={1,2,3}

Commutative Law: AUB=BUA
AUB ={1,2,3}
BUA ={2,3,1}

Commutative Law: A^B=B^A
A^B ={2}
B^A ={2}
S ={1,2,3}
A ={1,2}
B ={2,3}
C ={1,3}
```

18.3.3 To Perform Various Primitive Operations on Fuzzy Sets with Dynamic Components

The program is as follows:

```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>

struct SET
{
  float nr[5];
  float dr[5];
  int n;
};

typedef struct SET fuzzy;

void getval(fuzzy *m,char *x)
{
  int i;
  float f;
  clrscr();
  printf("\n Enter the %s:\n",x);
  for(i=0;i<m->n;i++)
  {
    printf(" Numerator Element %d :",i+1);
    scanf("%f",&f);
    m->nr[i]=f;
    fflush(stdin);
    printf("Denominator Element %d:",i+1);
    scanf("%f",&f);
    m->dr[i]=f;
  }
```



```
}
void printval(fuzzy *m,char *x)
{
  int i;
  printf("\n %s={",x);
  for(i=0;i<m->n;i++)
  {
    printf("%6.2f / %6.2f",m->nr[i],m->dr[i]);
    if(i!=m->n-1) putch('+');
  }
  printf(")");
}
fuzzy unionset(fuzzy a,fuzzy b)

{
  fuzzy temp;
  char ch;
  int i;
  temp.n=a.n;
  for(i=0;i<a.n;i++)
  {
    if(a.dr[i]!=b.dr[i])
      {
        printf("\n Denominators not equal");
        getch();
        exit(0);
      }
    if(a.nr[i]<b.nr[i])
      temp.nr[i]=b.nr[i];
    else
      temp.nr[i]=a.nr[i];
    temp.dr[i]=a.dr[i];
  }
  return temp;
}

fuzzy intersect(fuzzy a,fuzzy b)
{
  fuzzy temp;
  int i;
  temp.n=a.n;
  for(i=0;i<a.n;i++)
  {
    if(a.dr[i]!=b.dr[i])
      {
        printf("\n Denominators not equal");
        getch();
        exit(0);
      }
    if(a.nr[i]>b.nr[i])
```

```
        temp.nr[i]=b.nr[i];
      else
        temp.nr[i]=a.nr[i];
      temp.dr[i]=a.dr[i];
    }
    return temp;
}

fuzzy complement(fuzzy a)
{
  fuzzy temp;
  int i;
  temp.n=a.n;
  for(i=0;i<a.n;i++)
  {
    temp.nr[i]=1-a.nr[i];
    temp.dr[i]=a.dr[i];
  }
  return temp;
}

void main()
{
  fuzzy a,b,ans;
  char ch;
  clrscr();
  printf("\n Enter the no of componets:");
  scanf("%d",&a.n);
  b.n=a.n;
  getval(&a,"A");
  getval(&b,"B");
  clrscr();
  printval(&a,"A");
  printval(&b,"B");
  getch();
  while(1)
  {
    clrscr();
    printf("\n Menu:\n 1.AUB\n 2.A^B\n 3.A~ \n 4.B~   \n 5.Print
        S,A,B\n 6.Exit");
    switch((ch=getch()))
    {
      case '1':
        ans=unionset(a,b);
        printval(&ans,"AUB");
        getch();
        break;
      case '2':
        ans=intersect(a,b);
        printval(&ans,"A^B");
```



```
        getch();
        break;
      case '3':
        ans=complement(a);
        printval(&ans,"A~");
        getch();
        break;
      case '4':
        ans=complement(b);
        printval(&ans,"B~");
        getch();
        break;
      case '5':
        printval(&a,"A");
        printval(&b,"B");
        getch();
        break;
      case '6':
        exit(0);
    }
  }
}
```

Output

```
Enter the no of componets:3

Enter the A:
Numerator Element 1 :0.4
Denominator Element 1:1
Numerator Element 2 :0.2
Denominator Element 2:2
Numerator Element 3 :0.7
Denominator Element 3:3

Enter the B:
Numerator Element 1 :0.4
Denominator Element 1:1
Numerator Element 2 :0.8
Denominator Element 2:2
Numerator Element 3 :0.2
Denominator Element 3:3

A={ 0.40 / 1.00+ 0.20 / 2.00+ 0.70 / 3.00 }
B={ 0.40 / 1.00+ 0.80 / 2.00+ 0.20 / 3.00 }

Menu:
1.AUB
2.A^B
3.A~
```

```
4.B~
5.Print S,A,B
6.Exit

AUB={ 0.40 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00 }
A^B={ 0.40 / 1.00+ 0.20 / 2.00+ 0.20 / 3.00 }
A~={ 0.60 / 1.00+ 0.80 / 2.00+ 0.30 / 3.00 }
B~={ 0.60 / 1.00+ 0.20 / 2.00+ 0.80 / 3.00}
```

18.3.4 To Verify the Various Laws Associated with Fuzzy Set

The program is as follows:

```
#include<stdio.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>

struct SET
{
  float nr[5];
  float dr[5];
  int n;
};

typedef struct SET fuzzy;

void printval(fuzzy *m,char *x)
{
  int i;
  printf("\n %s={",x);
  for(i=0;i<m->n;i++)
  {
    printf(" %6.2f / %6.2f",m->nr[i],m->dr[i]);
    if(i!=m->n-1) putch('+');
  }
  printf(" }");
}
fuzzy unionset(fuzzy a,fuzzy b)
{
  fuzzy temp;
  char ch;
  int i;
  temp.n=a.n;
  for(i=0;i<a.n;i++)
  {
    if(a.dr[i]!=b.dr[i])
    {
      printf("\n Denominators not equal");
      getch();
```



```
      exit(0);
    }
    if(a.nr[i]<b.nr[i])
      temp.nr[i]=b.nr[i];
    else
      temp.nr[i]=a.nr[i];
    temp.dr[i]=a.dr[i];
  }
  return temp;
}

fuzzy intersect(fuzzy a,fuzzy b)
{
  fuzzy temp;
  int i;
  temp.n=a.n;
  for(i=0;i<a.n;i++)
  {
    if(a.dr[i]!=b.dr[i])
      {
        printf("\n Denominators not equal");
        getch();
        exit(0);
      }
    if(a.nr[i]>b.nr[i])
      temp.nr[i]=b.nr[i];
    else
      temp.nr[i]=a.nr[i];
    temp.dr[i]=a.dr[i];
  }
 return temp;
 }

 fuzzy complement(fuzzy a)
 {
   fuzzy temp;
   int i;
   temp.n=a.n;
   for(i=0;i<a.n;i++)
   {
     temp.nr[i]=1-a.nr[i];
     temp.dr[i]=a.dr[i];
   }
   return temp;
 }

 void main()
 {
   fuzzy a,b,temp1,temp2,ans;
   char ch;
```

```
clrscr();
a.n=b.n=3;
a.nr[0]=0.1; a.dr[0]=1;
a.nr[1]=0.2; a.dr[1]=2;
a.nr[2]=0.3; a.dr[2]=3;
b.nr[0]=0.4; b.dr[0]=1;
b.nr[1]=0.3; b.dr[1]=2;
b.nr[2]=0.2; b.dr[2]=3;
printval(&a,"A");
printval(&b,"B");
getch();
printf("\n Menu:\n 1.Difference A/B \n 2.Difference B/A\n
  3.DeMorgan's law -1\n 4.DeMorgan's law -2\n 5.Excluded Middle
  laws\n 6.Print S,A,B\n 7.Exit");
while(1)
{
  switch((ch=getch()))
  {
    case '1':
      temp1=complement(b);
      printval(&temp1,"1.B~");
      printval(&a,"A");
      ans=intersect(a,temp1);
      printval(&ans,"A/B = A^B~");
      break;
    case '2':
      temp1=complement(a);
      printval(&temp1,"2.A~");
      printval(&b,"B");
      ans=unionset(a,temp1);
      printval(&ans,"A/B = B^A~");
      break;
    case '3':
      ans=unionset(a,b);
      ans=complement(ans);
      printval(&ans,"3.(AUB)~");
      temp1=complement(a);
      temp2=complement(b);
      printval(&temp1,"A~");
      printval(&temp2,"B~");
      ans=intersect(temp1,temp2);
      printval(&ans,"A~^B~");
      break;
    case '4':
      ans=intersect(a,b);
      ans=complement(ans);
      printval(&ans,"4.(A^B)~");
      temp1=complement(a);
      temp2=complement(b);
      printval(&temp1,"A~");
```



```
      printval(&temp2,"B~");
      ans=unionset(temp1,temp2);
      printval(&ans,"A~ U B~");
      break;
    case '5':
      ans=complement(a);
      ans=unionset(ans,a);
      printval(&ans,"5.A~^A");
      ans=complement(b);
      ans=unionset(ans,b);
      printval(&ans,"B~B");
      break;
    case '6':
      printval(&a,"A");
      printval(&b,"B");
      break;
    case '7':
      exit(0);
  }
 }
}
```

Output

```
Menu:
1.Difference A/B
2.Difference B/A
3.DeMorgan's law -1
4.DeMorgan's law -2
5.Excluded Middle laws
6.Print S,A,B
7.Exit
1.B~={0.60 / 1.00+ 0.70 / 2.00+ 0.80 / 3.00}
A={0.10 / 1.00+ 0.20 / 2.00+ 0.30 / 3.00}
A/B = A^B~={0.10 / 1.00+ 0.20 / 2.00+ 0.30 / 3.00}
2.A~={0.90 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00}
B={0.40 / 1.00+ 0.30 / 2.00+ 0.20 / 3.00}
A/B = B^A~={0.90 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00}
3.(AUB)~={0.60 / 1.00+ 0.70 / 2.00+ 0.70 / 3.00}
A~={0.90 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00}
B~={0.60 / 1.00+ 0.70 / 2.00+ 0.80 / 3.00}
A~ ^B~={0.60 / 1.00+ 0.70 / 2.00+ 0.70 / 3.00}
4.(A^B)~={0.90 / 1.00+ 0.80 / 2.00+ 0.80 / 3.00}
A~={0.90 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00}
B~={0.60 / 1.00+ 0.70 / 2.00+ 0.80 / 3.00}
A~ U B~={0.90 / 1.00+ 0.80 / 2.00+ 0.80 / 3.00}
5.A~^A={0.90 / 1.00+ 0.80 / 2.00+ 0.70 / 3.00}
B~ B={0.60 / 1.00+ 0.70 / 2.00+ 0.80 / 3.00}
A={0.10 / 1.00+ 0.20 / 2.00+ 0.30 / 3.00}
B={0.40 / 1.00+ 0.30 / 2.00+ 0.20 / 3.00}
```

18.3.5 To Perform Cartesian Product Over Two Given Fuzzy Sets

```
#include<stdio.h>
#include<limits.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>
#define min(x,y) (x<y ? x : y)

struct SET
{
  float nr[5];
  float dr[5];
  int n;
};
typedef struct SET fuzzy;

void printval(fuzzy *m,char *x)
{
  int i;
  printf("\n %s={",x);
  for(i=0;i<m->n;i++)
  {
    printf(" %5.2f /%5.2f ",m->nr[i],m->dr[i]);
    if(i!=m->n-1) putch('+');
  }
  printf(" }");
}

void main()
{
  fuzzy V,I;
  int i,j;
  float P[6][6];
  clrscr();

  V.n=I.n=5;
  V.nr[0]=0.2;    V.dr[0]=30;
  V.nr[1]=0.8;    V.dr[1]=45;
  V.nr[2]=1;      V.dr[2]=60;
  V.nr[3]=0.9;    V.dr[3]=75;
  V.nr[4]=0.7;    V.dr[4]=90;

  I.nr[0]=0.4;    I.dr[0]=0.8;
  I.nr[1]=0.7;    I.dr[1]=0.9;
  I.nr[2]=1;      I.dr[2]=1;
  I.nr[3]=0.8;    I.dr[3]=1.1;
  I.nr[4]=0.6;    I.dr[4]=1.2;

  printval(&V,"V");
```



```
  printval(&I,"I");
  printf("\n");
  for(i=0;i<=V.n;i++)
    for(j=0;j<=I.n;j++)
    {
      if(i==0 && j>0)
        P[i][j]=I.dr[j-1];
      else if(j==0 && i>0)
        P[i][j]=V.dr[i-1];
      else if(i>0 && j>0)
        P[i][j]=min(V.nr[i-1],I.nr[j-1]);
    }
   for(i=0;i<=V.n;i++)
   {
     for(j=0;j<=I.n;j++)
     {
       if(i==0 && j==0)
         printf(" ");
       else
         printf(" %6.2f ",P[i][j]);
     }
     printf("\n");
   }
   getch();
}
```

Output

```
V={0.20/30.00 + 0.80/45.00 + 1.00/60.00 + 0.90/75.00 + 0.70/90.00}
I={0.40/0.80 + 0.70/0.90 + 1.00/1.00 + 0.80/1.10 + 0.60/1.20}
VxI=
          0.80   0.90   1.00   1.10   1.20
30.00     0.20   0.20   0.20   0.20   0.20
45.00     0.40   0.70   0.80   0.80   0.60
60.00     0.40   0.70   1.00   0.80   0.60
75.00     0.40   0.70   0.90   0.80   0.60
90.00     0.40   0.70   0.70   0.70   0.60
```

18.3.6 To Perform Max-Min Composition of Two Matrices Obtained from Cartesian Product

The program is as follows:

```
#include<stdio.h>
#include<limits.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>

#define min(x,y) (x<y ? x : y)
```

```
struct SET
{
  float nr[5];
  float dr[5];
  int n;
};

typedef struct SET fuzzy;
void printval(fuzzy *m,char *x)
{
  int i;
  printf("\n %s={",x);
  for(i=0;i<m->n;i++)
  {
    printf(" %5.2f /%5.2f ",m->nr[i],m->dr[i]);
    if(i!=m->n-1) putch('+');
  }
  printf(" }");
}

void main()
{
  fuzzy V,I,C;
  int i,j,k,prows,pcols,trows,tcols;
  float P[6][6],T[6][4],E[6][4],max;

  clrscr();

  V.n=I.n=5;        C.n=3;
  V.nr[0]=0.2;      V.dr[0]=30;
  V.nr[1]=0.8;      V.dr[1]=45;
  V.nr[2]=1;        V.dr[2]=60;
  V.nr[3]=0.9;      V.dr[3]=75;
  V.nr[4]=0.7;      V.dr[4]=90;

  I.nr[0]=0.4;      I.dr[0]=0.8;
  I.nr[1]=0.7;      I.dr[1]=0.9;
  I.nr[2]=1;        I.dr[2]=1;
  I.nr[3]=0.8;      I.dr[3]=1.1;
  I.nr[4]=0.6;      I.dr[4]=1.2;

  C.nr[0]=0.4;      C.dr[0]=0.5;
  C.nr[1]=1;        C.dr[1]=0.6;
  C.nr[2]=0.5;      C.dr[2]=0.7;

  printval(&V,"V");
  printval(&I,"I");
  printval(&C,"C");
  printf("\n M=VxI=");

  for(i=0;i<=V.n;i++)
```



```
    for(j=0;j<=I.n;j++)
    {
      if(i==0 && j>0)
        P[i][j]=I.dr[j-1];
      else if(j==0 && i>0)
        P[i][j]=V.dr[i-1];
      else if(i>0 && j>0)
        P[i][j]=min(V.nr[i-1],I.nr[j-1]);
    }

  for(i=0;i<=I.n;i++)
    for(j=0;j<=C.n;j++)
    {
      if(i==0 && j>0)
        T[i][j]=C.dr[j-1];
      else if(j==0 && i>0)
        T[i][j]=I.dr[i-1];
      else if(i>0 && j>0)
        T[i][j]=min(I.nr[i-1],C.nr[j-1]);
    }
  for(i=0;i<=V.n;i++)
  {
    for(j=0;j<=I.n;j++)
    {
      if(i==0 && j==0)
        printf(" ");
      else
        printf(" %6.2f ",P[i][j]);
    }
    printf("\n");
  }
  printf("\n N=IxC=");
  for(i=0;i<=I.n;i++)
  {
    for(j=0;j<=C.n;j++)
    {
      if(i==0 && j==0)
        printf(" ");
      else
        printf(" %6.2f ",T[i][j]);
    }
    printf("\n");
  }

  prows=6,pcols=6;
  trows=6,tcols=4;
  for(i=0;i<prows;i++)
  {
    for(j=0;j<tcols;j++)
    {
```

```
        if(i==0 && j==0)
          E[i][j]=0;
        else if(i==0 && j>0)
          E[i][j]=T[i][j];
        else if(i>0 && j==0)
          E[i][j]=P[i][j];
        else
        {
          max=0;
          for(k=1;k<pcols;k++)
          {
            if(i>0 && j>0)
              if(max < min(P[i][k],T[k][j]))
                max=min(P[i][k],T[k][j]);
          }
          if(i>0 && j>0)
            E[i][j]=max;
        }
      }
    }

    getch();

    printf("\n M o N");
    for(i=0;i<prows;i++)
    {
      for(j=0;j<tcols;j++)
      {
        if(i==0 && j==0)
          printf(" ");
        else
          printf(" %6.2f ",E[i][j]);
      }
      printf("\n");
    }
    getch();
}
```

Output

```
V={0.20/30.00 + 0.80/45.00 + 1.00/60.00 + 0.90/75.00 + 0.70/90.00}
I={0.40/0.80 + 0.70/0.90 + 1.00/1.00 + 0.80/1.10 + 0.60/1.20}
C={0.40/0.50 + 1.00/0.60 + 0.50/0.70}

M=VxI=
        0.80   0.90   1.00   1.10   1.20
30.00   0.20   0.20   0.20   0.20   0.20
45.00   0.40   0.70   0.80   0.80   0.60
60.00   0.40   0.70   1.00   0.80   0.60
75.00   0.40   0.70.  0.90   0.80   0.60
90.00   0.40   0.70   0.70   0.70   0.60
```



```
N=IxC=
        0.50   0.60   0.70
0.80    0.40   0.40   0.40
0.90    0.40   0.70   0.50
1.00    0.40   1.00   0.50
1.10    0.40   0.80   0.50
1.20    0.40   0.60   0.50

M o N
        0.50   0.60   0.70
30.00   0.20   0.20   0.20
45.00   0.40   0.80   0.50
60.00   0.40   1.00   0.50
75.00   0.40   0.90   0.50
90.00   0.40   0.70   0.50
```

18.3.7 To Perform Max-Product Composition of Two Matrices Obtained from Cartesian Product

The program is as follows:

```
#include<stdio.h>
#include<limits.h>
#include<alloc.h>
#include<conio.h>
#include<stdlib.h>

#define product(x,y) ((x)*(y))

struct SET
{
  float nr[5];
  float dr[5];
  int n;
};

typedef struct SET fuzzy;
void printval(fuzzy *m,char *x)
{
  int i;
  printf("\n %s={",x);
  for(i=0;i<m->n;i++)
  {
    printf(" %5.2f /%5.2f ",m->nr[i],m->dr[i]);
    if(i!=m->n-1) putch('+');
  }
  printf(" }");
}

void main()
{
```

```
fuzzy V,I,C;
int i,j,k,prows,pcols,trows,tcols;
float P[6][6],T[6][4],E[6][4],max;

clrscr();

V.n=I.n=5;        C.n=3;
V.nr[0]=0.2;      V.dr[0]=30;
V.nr[1]=0.8;      V.dr[1]=45;
V.nr[2]=1;        V.dr[2]=60;
V.nr[3]=0.9;      V.dr[3]=75;
V.nr[4]=0.7;      V.dr[4]=90;

I.nr[0]=0.4;      I.dr[0]=0.8;
I.nr[1]=0.7;      I.dr[1]=0.9;
I.nr[2]=1;        I.dr[2]=1;
I.nr[3]=0.8;      I.dr[3]=1.1;
I.nr[4]=0.6;      I.dr[4]=1.2;

C.nr[0]=0.4;      C.dr[0]=0.5;
C.nr[1]=1;        C.dr[1]=0.6;
C.nr[2]=0.5;      C.dr[2]=0.7;

printval(&V,"V");
printval(&I,"I");
printval(&C,"C");
printf("\n");

for(i=0;i<=V.n;i++)
for(j=0;j<=I.n;j++)
  {
    if(i==0 && j>0)
      P[i][j]=I.dr[j-1];
    else if(j==0 && i>0)
      P[i][j]=V.dr[i-1];
    else if(i>0 && j>0)
      P[i][j]=min(V.nr[i-1],I.nr[j-1]);
  }

for(i=0;i<=I.n;i++)
  for(j=0;j<=C.n;j++)
  {
    if(i==0 && j>0)
      T[i][j]=C.dr[j-1];
    else if(j==0 && i>0)
      T[i][j]=I.dr[i-1];
    else if(i>0 && j>0)
      T[i][j]=min(I.nr[i-1],C.nr[j-1]);
  }
```



```
for(i=0;i<=V.n;i++)
{
  for(j=0;j<=I.n;j++)
  {
    if(i==0 && j==0)
      printf(" ");
    else
      printf(" %6.2f ",P[i][j]);
  }
  printf("\n");
}
printf("\n");
for(i=0;i<=I.n;i++)
{
  for(j=0;j<=C.n;j++)
  {
    if(i==0 && j==0)
      printf(" ");
    else
      printf(" %6.2f ",T[i][j]);
  }
  printf("\n");
}

prows=6,pcols=6;
trows=6,tcols=4;
for(i=0;i<prows;i++)
  {
    for(j=0;j<tcols;j++)
    {
      if(i==0 && j==0)
        E[i][j]=0;
      else if(i==0 && j>0)
        E[i][j]=T[i][j];
      else if(i>0 && j==0)
        E[i][j]=P[i][j];
      else
      {
        max=0;
        for(k=1;k<pcols;k++)
        {
          if(i>0 && j>0)
            if(max < product(P[i][k],T[k][j]))
              max=product(P[i][k],T[k][j]);
        }
      if(i>0 && j>0)
        E[i][j]=max;
      }
    }
  }
```

```
    getch();

    printf("\n");
    for(i=0;i<prows;i++)
    {
      for(j=0;j<tcols;j++)
      {
        if(i==0 && j==0)
          printf(" ");
        else
          printf(" %6.2f ",E[i][j]);
      }
      printf("\n");
    }
    getch();
}
```

Output

```
V={0.20/30.00 + 0.80/45.00 + 1.00/60.00 + 0.90/75.00 + 0.70/90.00}
I={0.40/0.80 + 0.70/0.90 + 1.00/1.00 + 0.80/1.10 + 0.60/1.20}
C={0.40/0.50 + 1.00/0.60 + 0.50/0.70}

M=VxI
        0.80   0.90   1.00   1.10   1.20
30.00   0.20   0.20   0.20   0.20   0.20
45.00   0.40   0.70   0.80   0.80   0.60
60.00   0.40   0.70   1.00   0.80   0.60
75.00   0.40   0.70   0.90   0.80   0.60
90.00   0.40   0.70   0.70   0.70   0.60

N=IxC
        0.50   0.60   0.70
0.80    0.40   0.40   0.40
0.90    0.40   0.70   0.50
1.00    0.40   1.00   0.50
1.10    0.40   0.80   0.50
1.20    0.40   0.60   0.50

M o N
        0.50   0.60   0.70
30.00   0.08   0.20   0.10
45.00   0.32   0.80   0.40
60.00   0.40   1.00   0.50
75.00   0.36   0.90   0.45
90.00   0.28   0.70   0.35
```

18.4 Genetic Algorithm Implementation

The genetic algorithm concept discussed in Chapter 15 is brought for various applications using C and C++ in this section. The source code for each application is given below.

18.4.1 To Maximize $F(x_1, x_2) = 4x_1 + 3x_2$

To find the solution of the function Max $F(x_1, x_2) = 4x_1 + 3x_2$, given the constraints

$$2 * x_1 + 3 * x_2 \leq 6$$
$$-3 * x_1 + 2 * x_2 \leq 3$$
$$2 * x_1 + x_2 \leq 4$$
$$0 \leq x \leq 2$$

using genetic algorithm.

Steps involved

Step 1: Generate initial population by using random number generation.

Step 2: Use the tournament selection method to select any two parents.

Step 3: Generate the offsprings by using the following arithmetic crossover operator:

$$x_1 = a * x_1 + (1 - a) * x_2$$
$$x_2 = a * x_2 + (1 - a) * x_1$$

Step 4: Calculate the maximum fitness value by applying this operation separately for each iteration.

Step 5: To print the output of the function.

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<stdlib.h>
#include<math.h>
float mutation(float ,int );
      //Main Program
void main()
{
float x1[10],x2[10],sum[10],max_val=0.0,a,max_x1,max_x2;
int flag=0,j,i,k;
clrscr();
randomize();
      //Initial Population Generation
printf("\tInitial Population \n");
for(i=0;i<4;i++)
{
x1[i]=(float)(random(1000.0))/500.0;
x2[i]=(float)(random(1000.0))/500.0;
printf("\t%d\tx1 : %f\tx2 : %f\n",i,x1[i],x2[i]);
}

for(i=0;i<10;i++)
{
for(j=0;j<4;j++)
```

```
{
flag=0;
      //Constraints
if(((2*x1[j])+(3*x2[j]))<=6)        //2x1+3x2<=6
{
  if(((-3*x1[j])+(2*x2[j]))<=3) //-3x1+2x2<=3
    {
    if(((2*x1[j])+(x2[j]))<=4) //2x1+x2<=4
    {
      flag=1;
      sum[j]=(4*x1[j])+(3*x2[j]);
    }
  }
}
if(flag==0)
sum[j]=0;
}
  printf("\t After %d generation\n",i);
  for(k=0;k<4;k++)
  {
    printf("\t%d\tx1 : %f\tx2 : %f\tsum : %f\n",k,x1[k],x2[k],sum[k]);
  }
  for(k=0;k<4;k++)
  {
    if(max_val<sum[k])
    {
      max_val=sum[k];
      max_x1=x1[k];
      max_x2=x2[k];
    }
  }
  printf("\tx1 : %f\tx2 : %f\tMax : %f\n",max_x1,max_x2,max_val);
  getch();
  for(k=0;k<3;k++)
  {
      //cross over operation
    a=(float)(random(1000.0))/1000.0;
    x1[k]=((a*x1[k])+(1-a)*x2[k+1]);
    x2[k]=((a*x2[k])+(1-a)*x1[k+1]);
  }

  a=(float)(random(1000.0))/1000.0;
  x1[4]=((a*x1[4])+(1-a)*x2[1]);
  x2[4]=((a*x2[4])+(1-a)*x1[1]);

      //Mutation Operation
  for(k=0;k<4;k++)
  {
    if(sum[k]==0)
    {
```



```
      x1[k]=(x1[0]+x1[1]+x1[2]+x1[3])/4.0;
      x1[k]=(x2[0]+x2[1]+x2[2]+x2[3])/4.0;
      }
      if(sum[k]==sum[k+1])
      {
      x1[k]=(float)(random(1000.0))/500.0;
      x2[k]=(float)(random(1000.0))/500.0;
      }
    }
  }
  clrscr();
  printf("\tTHE SOLUTION OF THE FOLLOWING PROBLEM IS\n");
  printf("\n\tSUM:\tMAXIMIZE\tF(x1,x2) : 4x1+3x2");
  printf("\n\n\t2*x1+3*x2<=5\n\t-3*x1+2*x2<=3\n\t2*x1+x2<=4\n");
  printf("\n\tx1 : %f\tx2 : %f\tMax : %f\n",max_x1,max_x2,max_val);
  getch();
  }

  void cross_over(float x1,float x2)
  {
    int a;
    a=(float)(random(1000))/1000;
    x1=((a*x1)+(1-a)*x2);
    x2=((a*x2)+(1-a)*x1);
  }

  float mutation(float x,int i)
  {
  }
  float max(float sum)
  {
  int i;
  for(i=0;i<3;i++)
  {
  if(sum[i]<sum[i+1])
  max=sum[i+1];
  else
  max=sum[i];
  return max;
  }
```

Output

Initial Population

```
0  x1 : 0.432000  x2 : 1.366000
1  x1 : 1.452000  x2 : 0.468000
2  x1 : 0.928000  x2 : 0.854000
3  x1 : 0.860000  x2 : 1.342000
```

After 0 generation

```
0  x1 : 0.432000  x2 : 1.366000  sum : 5.826000
```

```
1  x1 : 1.452000  x2 : 0.468000  sum : 7.212000
2  x1 : 0.928000  x2 : 0.854000  sum : 6.274000
3  x1 : 0.860000  x2 : 1.342000  sum : 7.466000
   x1 : 0.860000  x2 : 1.342000  Max : 7.466000
```

After 1 generation

```
0  x1 : 1.477638  x2 : 1.397484  sum : 0.000000
1  x1 : 1.364000  x2 : 0.960000  sum : 8.336000
2  x1 : 0.820292  x2 : 0.933684  sum : 6.082220
3  x1 : 0.524000  x2 : 0.990000  sum : 5.066000
   x1 : 1.364000  x2 : 0.960000  Max : 8.336000
```

After 2 generation

```
0  x1 : 0.979173  x2 : 1.364904  sum : 0.000000
1  x1 : 1.039542  x2 : 0.854660  sum : 6.722147
2  x1 : 0.914141  x2 : 0.707129  sum : 5.777948
3  x1 : 0.524000  x2 : 0.990000  sum : 5.066000
   x1 : 1.364000  x2 : 0.960000  Max : 8.336000
```

After 3 generation

```
0  x1 : 1.342291  x2 : 1.076486  sum : 8.598619
1  x1 : 0.922836  x2 : 1.088421  sum : 6.956608
2  x1 : 0.784797  x2 : 0.836438  sum : 5.648500
3  x1 : 0.782000  x2 : 0.690000  sum : 5.198000
   x1 : 1.342291  x2 : 1.076486  Max : 8.598619
```

After 4 generation

```
0  x1 : 1.180576  x2 : 0.978611  sum : 7.658136
1  x1 : 0.858469  x2 : 0.862221  sum : 6.020540
2  x1 : 0.774369  x2 : 0.830450  sum : 5.588825
3  x1 : 0.782000  x2 : 0.690000  sum : 5.198000
   x1 : 1.342291  x2 : 1.076486  Max : 8.598619
```

After 5 generation

```
0  x1 : 1.106081  x2 : 0.950498  sum : 7.275816
1  x1 : 0.842442  x2 : 0.811970  sum : 5.805677
2  x1 : 0.757664  x2 : 0.820857  sum : 5.493226
3  x1 : 0.782000  x2 : 0.690000  sum : 5.198000
   x1 : 1.342291  x2 : 1.076486  Max : 8.598619
```

After 6 generation

```
0  x1 : 1.071964  x2 : 0.937963  sum : 7.101746
1  x1 : 0.839420  x2 : 0.804367  sum : 5.770781
2  x1 : 0.695954  x2 : 0.785419  sum : 5.140076
3  x1 : 0.782000  x2 : 0.690000  sum : 5.198000
   x1 : 1.342291  x2 : 1.076486  Max : 8.598619
```


THE SOLUTION OF THE FOLLOWING PROBLEM IS

SUM: MAXIMIZE $F(x1, x2) = 4x1 + 3x2$

$$2 * x1 + 3 * x2 <= 6$$
$$-3 * x1 + 2 * x2 <= 3$$
$$2 * x1 + x2 <= 4$$

x1 : 1.342291 x2 : 1.076486 Max : 8.598619

18.4.2 To Minimize a Function $F(x) = x^2$

To write a program to minimize $F(x) = x^2$ using genetic algorithm.

Steps involved

Step 1: Generate the random number as *n*.

Step 2: Initialize *i*, *j* to *n* and *m* respectively.

Step 3: Max ← 1000, x[*i*] ← 0, sum ← 0, m_max ← 1000

Step 4: Compute x[i] ← x[i]+(pp[i][j]*pow(2,p-1-j)) and

$$fx[i] = x[i] * x[i]$$
$$\text{sum} = \text{sum} + fx[i]$$

Step 5: If (max>=fx[i])

Step 6: max ← fx[i];

Step 7: until m_max>max

Step 8: Compute minimum value

Program

```
#include<stdio.h>
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

int pop[10][10],npop[10][10],tpop[10][10],x[10],fx[10],m_max=961,
    ico=0,ico1,it=0;
void iter(int [10][10],int,int);
int u_rand(int);
void tour_sel(int,int);
void cross_ov(int,int);
void mutat(int,int);
void main()
{
int k,m,j,i,p[10],n=0,a[10],nit;
clrscr();
```

```
randomize();
cout<<"\t\tENTER THE NUMBER OF POPULATION IN EACH ITERATION : ";
cin>>n;
cout<<"\n\t\tENTER THE NUMBER OF ITERATION : ";
cin>>nit;
m=5;
for(i=0;i<n;i++)
{
for(j=m-1;j>=0;j--)
{
  pop[i][j]=u_rand(2);
}
}
cout<<"\nITERATION "<<it<<" IS :\n";
iter(pop,n,m);
it++;
getch();
do
{
it++;
cout<<"\nITERATION "<<it<<" IS :\n";
tour_sel(n,m);
iter(pop,n,m);
getch();
}while(it<nit);
cout<<"\n\nAFTER THE "<<ico1<<" ITERATION, THE MINIMUM VALUE IS :
    "<<(int) sqrt(m_max);
getch();
}
void iter(int pp[10][10],int o, int p)
{
int i,j,sum,avg,max=961;
for(i=0;i<o;i++)
{
x[i]=0;
for(j=0;j<p;j++)
{
x[i]=x[i]+(pp[i][j]*pow(2,p-1-j));
}
fx[i]=x[i]*x[i];
sum=sum+fx[i];
if (max>=fx[i])
  max=fx[i];
}
avg=sum/o;
cout<<"\n\nS.NO.\tPOPULATION\tX\tF(X)\n\n";
for(i=0;i<o;i++)
{
cout<<ico<<"\t";
ico++;
```



```
for(j=0;j<p;j++)
  cout<<pp[i][j];
cout<<"\t\t"<<x[i]<<"\t"<<fx[i]<<"\n";
}
cout<<"\n\t SUM : "<<sum<<"\tAVERAGE : "<<avg<<"\tMINIMUM : "<<max<<"\n";
if (m_max>max)
  {
  m_max=max;
  ico1=it;
  }
}
int u_rand(int x)
{
int y;
y=rand()%x;
return(y);
}
void tour_sel(int np,int mb)
{
int i,j,k,l,co=0,cc;
do
{
k=u_rand(np);
do
{
cc=0;
l=u_rand(np);
if (k==l)
  cc++;
}while(cc!=0);
if (fx[k]>fx[l])
{
  for(j=0;j<mb;j++)
  npop[co][j]=pop[k][j];
}
else if (fx[k]<fx[l])
  {
  for(j=0;j<mb;j++)
    npop[co][j]=pop[l][j];
  }
co++;
}while(co<np);
getch();
cross_ov(np,mb);
getch();
}
void cross_ov(int np1,int mb1)
{
int i,j,k,l,co,temp;
i=0;
```

```
do
{
k=rand()%2;
do
{
co=0;
l=u_rand(mb1);
if (((k==0) && (l==0)) || ((k==1) && (l==mb1)))
  co++;
}while(co!=0);

if ((k==0) && (l!=0))
{
  for(j=0;j<l;j++)
  {
    temp=npop[i][j];
    npop[i][j]=npop[i+1][j];
    npop[i+1][j]=temp;
  }
}
else if ((k==1) && (l!=mb1))
  {
    for(j=1;j<mb1;j++)
    {
      temp=npop[i][j];
      npop[i][j]=npop[i+1][j];
      npop[i+1][j]=temp;
    }
  }
i=i+2;
}while(i<np1);

for(i=0;i<np1;i++)
{
for(j=0;j<mb1;j++)
{
tpop[i][j]=npop[i][j];
}
}
mutat(np1,mb1);
}
void mutat(int np2,int mb2)
{
int i,j,r,temp,k,z;
i=0;
do
{
  for(k=0;k<np2;k++)
  {
  r=0;
```



```
    if (i!=k)
    {
    for(j=0;j<mb2;j++)
    {
    if (tpop[i][j]==tpop[k][j])
      r++;
    }
    if (r!=mb2-1)
    {
      z=u_rand(mb2);
      if (tpop[i][z]==0)
        tpop[i][z]=1;
      else
        tpop[i][z]=0;
      if (npop[k][u_rand(mb2)]==0)
        npop[k][u_rand(mb2)]=1;
      else
        npop[k][u_rand(mb2)]=0;
      mutat(k,mb2);
    }
    }
    }
  i++;
  }while(i<np2);
  for(i=0;i<np2;i++)
  {
  for(j=0;j<mb2;j++)
  {
  pop[i][j]=tpop[i][j];
  }
  }
  }
```

Output

```
ENTER THE NUMBER OF POPULATION IN EACH ITERATION: 5
ENTER THE NUMBER OF ITERATION: 8
ITERATION 1 IS :
```

S.NO.	POPULATION	X	F(X)
0	00001	1	1
1	11011	27	729
2	11011	27	729
3	01001	9	81
4	00111	7	49

```
SUM : 1589  AVERAGE : 317  MINIMUM : 1
ITERATION 2 IS :
```

S.NO.	POPULATION	X	F(X)
5	10101	21	441
6	11011	27	729
7	01011	11	121

```
8        10010          18    324
9        11110          30    900
  SUM : 2517  AVERAGE : 503  MINIMUM : 121
  ITERATION 3 IS :
S.NO.    POPULATION     X     F(X)
10       10010          18    324
11       10110          22    484
12       11011          27    729
13       01111          15    225
14       01011          11    121
  SUM : 1886  AVERAGE : 377  MINIMUM : 121
  ITERATION 4 IS :
S.NO.    POPULATION     X     F(X)
15       10111          23    529
16       10101          21    441
17       11001          25    625
18       10101          21    441
19       10100          20    400
  SUM : 2440  AVERAGE : 488  MINIMUM : 400
  ITERATION 5 IS :
S.NO.    POPULATION     X     F(X)
20       10110          22    484
21       11111          31    961
22       01111          15    225
23       10111          23    529
24       10110          22    484
  SUM : 2685  AVERAGE : 537  MINIMUM : 225
  ITERATION 6 IS :
S.NO.    POPULATION     X     F(X)
25       01001          9     81
26       01111          15    225
27       01011          11    121
28       01111          15    225
29       11100          28    784
  SUM : 1436  AVERAGE : 287  MINIMUM : 81
  ITERATION 7 IS :
S.NO.    POPULATION     X     F(X)
30       00000          0     0
31       00100          4     16
32       01111          15    225
33       11101          29    841
34       11010          26    676
  SUM : 1760  AVERAGE : 352  MINIMUM : 0
  ITERATION 8 IS :
S.NO.    POPULATION     X     F(X)
35       10000          16    256
36       00000          0     0
37       01011          11    121
```



```
38       01010          10    100
39       00001          1     1
  SUM : 481  AVERAGE : 96  MINIMUM : 0

  AFTER THE 7 ITERATION, THE MINIMUM VALUE IS : 0
```

18.4.3 Traveling Salesman Problem (TSP)

In TSP, salesman travels n cities and returns to the starting city with the minimal cost; he is not allowed to cross the city more than once. In this problem we are taking the assumption that all the n cities are interconnected. The cost indicates the distance between two cities.To solve this problem we make use of GA because the cities are randomly selected. Also the initial population for this problem is randomly selected cities. Fitness function is nothing but the minimum cost. Initially the fitness function is set to the maximum value and for each travel, the cost is calculated and compared with the fitness function. The new fitness value is assigned to the minimum cost. Initial population is randomly chosen and taken as the parent. For the next generation, the cyclic crossover is applied over the parent.

Cyclic Crossover

Let P1 and P2 are two parents

P1 : 2 8 0 1 3 4 5 7 9 6
P2 : 1 0 5 4 6 8 9 7 2 3

Select the first city P1 make it as the first city of offspring1(01)

O1 : 2 - - - - - - - - -

To find the next city of offspring O1 search current city, which is selected from P1 in P2. Find the location of city in P2 and select the city which is in the same location in P1. O1: 2 - - - - - - - 9 -

Continue the same procedure, we will get O1 as

O1 : 2 8 0 1 - 4 5 - 9 -

In the next step we will get the city 2 which is already present in O1 and then stop the procedure. Copy the cities from parent P2 in the corresponding locations

O1 : 2 8 0 1 6 4 5 7 9 3

For the generation offspring O2 the initial selection is from the parent P2, and repeat the procedure with P1

O2 : 1 5 4 3 8 9 7 2 6

If the initial population contain N parents it will generate $N(N-1)/2$ offsprings. The next generation the offsprings are considered as parent. The procedure is continued for N number of generation to find the minimum cost.

Source Code

```
#include<stdio.h>
#include<conio.h>
int tsp[10][10]={{999,10,3,2,5,6,7,2,5,4},
                 {20,999,3,5,10,2,8,1,15,6},
```

```
                {10,5,999,7,8,3,11,12,3,2},
                {3,4,5,999,6,4,10,6,1,8},
                {1,2,3,4,999,5,10,20,11,2},
                {8,5,3,10,2,999,6,9,20,1},
                {3,8,5,2,20,21,999,3,5,6},
                {5,2,1,25,15,10,6,999,8,1},
                {10,11,6,8,3,4,2,15,999,1},
                {5,10,6,4,15,1,3,5,2,999}
                };
int pa[1000][10]= {{0,1,2,3,4,5,6,7,8,9},
                   {9,8,6,3,2,1,0,4,5,7},
                   {2,3,5,0,1,4,9,8,6,7},
                   {4,8,9,0,1,3,2,5,6,7}
                   };
int i,j,k,l,m,y,loc,flag,row,col,it,x=3,y=3;
int count,row=0,res[1][10],row1,col1,z;
int numoff=4;
int offspring[1000][10];
int mincost=9999,mc;
main()
{
  int gen;
  clrscr();
  printf("Number of Generation : ");
  scanf("%d",&gen);
  offcal1(pa);
  offcal2(pa);
  printf(" \n\t\t First Generation\n");
  for(i=0;i<count;i++)
  {
    for(j=0;j<10;j++)
      printf("%d ",offspring[i][j]);
      printf("\n");
  }
  for(y=1;y<=gen-1;y++)
  {
      getch();
      clrscr();
      for(i=0;i<count;i++)
        for(j=0;j<10;j++)
          pa[i][j]=offspring[i][j];
      numoff=count;
      offcal1(pa);
      offcal2(pa);
      printf(" \n\t\t %d Generation\n",y+1);
      for(i=0;i<count;i++)
      {
        for(j=0;j<10;j++)
        printf("%d ",offspring[i][j]);
        printf("\n");
```



```
      }
      getch();
      clrscr();
    }
    printf("\n\nMinimum Cost Path\n");
    for(z=0;z<10;z++)
      printf("%d ",res[0][z]);
    printf("\nMinimum Cost %d \n",mincost);
}
/* finding the offspring using cyclic crossover */
offcal1(pa)
int pa[1000][10];
{
    count=0;
    for(i=0;i<1000;i++)
    for(j=0;j<10;j++)
      offspring[i][j]=-1;

    for(k=0;k<numoff;k++)
    {
      for(l=k+1;l<numoff;l++)
      {
        offspring[row][0]=pa[k][0];
        loc=pa[l][0];
        flag=1;
        while(flag != 0)
        {
          for(j=0;j<10;j++)
          {
            if(pa[k][j] == loc )
            {
              if (offspring[row][j]==-1)
              {
                offspring[row][j]=loc;
                loc=pa[l][j];
              }
              else
                flag=0;
            }
          }
        }/* end while*/
        for(m=0;m<10;m++)
        {
          if(offspring[row][m] == -1)
            offspring[row][m]=pa[l][m];
        }
        for(z=0;z<10;z++)
        {
          if(z<9)
          {
```

```
            row1=offspring[row][z];
            col1=offspring[row][z+1];
            mc=mc+tsp[row1][col1];
          }
          else
          {
            row1=offspring[row][z];
            col1=offspring[row][0];
            mc=mc+tsp[row1][col1];
          }
        }
        if(mc < mincost)
        {
          for(z=0;z<10;z++)
            res[0][z]=offspring[row][z];
          mincost=mc;
        }
        count++;
        row++;
      }/* end 1*/
    }
}
offcal2(pa)
int pa[1000][10];
{
  for(k=0;k<numoff;k++)
  {
    for(l=k+1;l<numoff;l++)
    {
      offspring[row][0]=pa[l][0];
      loc=pa[k][0];
      flag=1;
      while(flag != 0)
      {
        for(j=0;j<10;j++)
        {
          if(pa[l][j] == loc )
          {
            if (offspring[row][j]==-1)
            {
              offspring[row][j]=loc;
              loc=pa[k][j];
            }
            else
              flag=0;
          }
        }
      }/* end while*/
      for(m=0;m<10;m++)
      {
```



```
        if(offspring[row][m] == -1)
          offspring[row][m]=pa[k][m];
        }
        for(z=0;z<10;z++)
        {
      if(z<9)
      {
        row1=offspring[row][z];
        col1=offspring[row][z+1];
        mc=mc+tsp[row1][col1];
      }
      else
      {
        row1=offspring[row][z];
        col1=offspring[row][0];
        mc=mc+tsp[row1][col1];
      }
    }
        row++;
        if(mc < mincost)
        {
          for(z=0;z<10;z++)
            res[0][z]=offspring[row][z];
          mincost=mc;
        }
        count++;
      }/* end 1*/
    }
  }
```

Output

```
Number of Generation : 2
First Generation
0  8  2  3  4  1  6  7  5  9
0  1  2  3  4  5  9  8  6  7
0  1  2  3  4  5  6  7  8  9
9  8  5  3  2  1  0  4  6  7
9  8  6  0  1  3  2  4  5  7
2  3  5  0  1  4  9  8  6  7
9  1  6  3  2  5  0  4  8  7
2  3  5  0  1  4  6  7  8  9
4  8  9  0  1  3  2  5  6  7
2  3  6  0  1  4  9  8  5  7
4  8  9  3  2  1  0  5  6  7
4  8  9  0  1  3  2  5  6  7

2 Generation
0  1  2  3  4  5  9  8  6  7
0  1  2  3  4  5  6  7  8  9
0  8  2  3  4  1  6  7  5  9
```

```
0  8  6  3  4  1  2  7  5  9
0  8  2  3  1  4  6  7  5  9
0  1  2  3  4  5  6  7  8  9
0  8  2  3  1  4  6  7  5  9
0  8  9  3  4  1  2  5  6  7
0  8  2  3  1  4  6  7  5  9
0  8  2  3  4  1  6  7  5  9
0  8  9  3  4  1  2  5  6  7
0  1  2  3  4  5  6  7  8  9
Minimum Cost Path
0  8  2  3  4  1  6  7  5  9
Minimum Cost 53
```

18.4.4 Prisoner's Dilemma

Cooperation is usually analyzed in game theory by means of a non-zero-sum game called the "Prisoner's Dilemma." The two players in the game can choose between two moves, either "cooperate" or "defect." The idea is that each player gains when both cooperate, but if only one of them cooperates, the other one, who defects, will gain more. If both defect, both lose (or gain very little) but not as much as the "cheated" cooperator whose cooperation is not returned. The whole game situation and its different outcomes can be summarized by the following table where hypothetical "points" are given as an example of how the differences in result might be quantified.

Action of A/Action B	Cooperate	Defect
Cooperate	Fairly good [+5]	Bad [−10]
Defect	Good [+10]	Mediocre [0]

The type of crossover that is performed is a "single point crossover" where the point of crossover is randomly selected. The mutation is expected to happen every 2000 generation. It is easy to change the mutation as it is implemented as a separate function.

Source Code

```
#include<stdlib.h>
#include<stdio.h>
#include<conio.h>

int calculate(int*);
int* select(int *);
void crossover(int*,int*);
void sort_select(void);
//THESE ARE SOME GLOBAL VARIABLE USED
int best_score[20];
int score[9];
int index[6];

void main()
{
  int a[10][70],select_string[5][70];
  int best_string[20][70],max,ind=0;
```



```
  int p,counter=1;
  int i,n,j,temp[10];
  randomize();
  clrscr();
  for(j=0;j<10;j++)
  for(i=0;i<70;i++)
    a[j][i]=random(2);
  //THE NUMBER OF GENERATION TO BE SCANED IN
  printf(" Enter the no of generation ");
  scanf("%d",&n);
  for(i=0;i<10;i++)
    score[i]=calculate(&a[i][0]);
//function for sorting the score array and finding the index of
    best score
  sort_select();
  for(i=0;i<7;i++)
  {
    p=index[i]; //THE ORDER OF BEST SCORE STORED IN INDEX.
    for(j=0;j<70;j++)
      select_string[i][j]=a[p][j];
  }
  best_score[0]=score[0];
  for(i=0;i<70;i++)
    best_string[0][i]=select_string[0][i];

  while(counter < n)
  {
    for(i=0;i<7;i=i+2)
      crossover(&a[i][0],&a[i+1][0]);
      for(i=0;i<9;i++)
        score[i]=0;
        for(i=0;i<7;i++)
        score[i]=calculate(&a[i][0]);
//CALCULATE FUNCTION RETURNS SCORE OF EACH STRING
        sort_select();
        best_score[counter]=score[0];
        p=index[0];
       for(j=0;j<70;j++)
        best_string[counter][j]=a[p][j];
       counter++;
  }
  //OUTPUT THE BEST SCORES.
  for(p=0;p<n;p++)
  {
    printf("The best score in the generation %d :",p+1);
    printf(" %d \n", best_score[p]);
  }
  //OUTPUT THE BEST STRINGS.
  for(i=0;i<n;i++)
  {
```

```
    printf("\n\nTHE BEST STRNG IN GENERATION %d :\n\n",i+1);
    for(j=0;j<70;j++)
    {
      if(j%2==0&&j!=0)
      printf(" ");
      if(best_string[i][j] ==1)
        printf("d");
      //COVERTING 1'S AND 0'S TO d AND c
      else
        printf("c");
    }
  }
//CALCULATING THE BEST OF THE BEST
  for(i=0;i<n;i++)
    temp[i]=best_score[i];
  max=temp[0];
  for(i=1;i<n;i++)
  {
    if(max<temp[i])
    {
      max=temp[i];
      ind=i;
    }
  }
//CALCULATING THE BEST FROM THE SELECTED.
  printf("\n\n");
  printf("\nTHE BEST STRING IN ALL GENERATION IS \n\n");
  for(i=0;i<70;i++)
  {
    if(i%2==0&&i!=0)
    printf(" ");
    if(best_string[ind][i]==1)
      printf("d");
    else
    printf("c");
  }
printf("\n\nTHE CORRESPONDING BEST SCORE IS: %d ",best_score[ind]);
getch();
}
int calculate(int* ptr)
{
  int *a;
  int p1,p2,i;
  a=ptr;
  p1=0; p2=0;
  for(i=0;i<70;i=i+2) //calculating the values according to truth
      table.
  {
    if(a[i]==1 && a[i+1]==1)
    {
```



```
      p1=p1+3; p2=p2+3;
    }
    if(a[i]==1 && a[i+1]==0)
    {
      p1=p1+5; p2=p2+0;
    }
    if(a[i]==0 && a[i+1]==1)
    {
      p1=p1+0; p2=p2+5;
    }
    if(a[i]==0 && a[i+1]==0)
    {
      p1=p1+1; p2=p2+1;
    }
  }
  return(p1+p2); //RETRUN THE TOTAL SCORE OF THE STRING.
}
void sort_select() //ORDINARY SORTING PROCEDURE
{
  int temp[9],i,j,t;
  for(i=0;i<10;i++)
  temp[i]=score[i];

  for(i=0;i<10;i++)
    for(j=9;j>=i;j--)
    {
      if(temp[i]<temp[j]) //USUSAL SWAPPING PROCEDURE.
      {
        t=temp[j];
        temp[j]=temp[i];
        temp[i]=t;
      }
    }
  for(i=0;i<7;i++)
  for(j=0;j<10;j++)
    if(temp[i]==score[j])
      index[i]=j;
    score[0]=temp[0];
}
void crossover(int *ptr1,int *ptr2)
{
  int temp,i,j;
  int ind=random(60);  //RANDOM POINT OF CROSSOVER
  for(i=ind;i<70;i++)
  {
    temp=ptr1[i];
    ptr1[i]=ptr2[i];
    ptr2[i]=temp;
  }
}
```

Output

```
Enter the no of generation 5
The best score in the generation 1: 171
The best score in the generation 2: 160
The best score in the generation 3: 170
The best score in the generation 4: 166
The best score in the generation 5: 169
The best string in generation 1:
dd dc cd dc dc cd cd dd cc dc dc dc dd dc dd cd dc cd cd cc dc cd cc cd dd cd
cd dd cd dc cc cd dc dd dd
The best string in generation 2:
cd cc cd cc cd cd dd dc cd cc dc cc dd cd dd dd cc cc dc dd dc cd cd dd dc dd
dd cc cd dd dc dc cd dc cc
The best string in generation 3:
cd cc cd cc cd cd dc dd cd dc dd cc cd cd cc dd cd dd dc cd dc dc dd cd dc dc
dc cd cd cd dc dc dd dc dd
The best string in generation 4:
cd cc cd cc cd cd dc dd cd dc dd cc cd cd cc dd cd dd dc cd dc dc dd dd dc dd
dd cc cd dd dc dc cd dc cc
The best string in generation 5:
Cd dd cc cd dd dc cd cc dd cd dd dd dc cd cd cc dc cd cd dc cc dd dd dc dc dc
dd dc dc cd dc cc dc cd dd
The best string in all generation is
dd dc cd dc dc cd cd dd cc dc dc dc dd dc dd cd dc cd cd cc dc cd cc cd dd cd
cd dd cd dc cc cd dc dd dd
The corresponding best score is : 171
```

18.4.5 Quadratic Equation Solving

To find the roots of the quadratic equation using genetic algorithm. To solve the above problem for the quadratic equation $x*x+5*x+6$ using following procedure. It could be used for solving any quadratic equation by changing fitness function $f(x)$ and changing length of chromosome.

Steps involved

Step 1: Initial population size is 10 and chromosome length is set to 5. Selecting initial population, i.e. random approximate solution to the problem, which are 10 different 5-bit binary strings. Here initial population consists of 10 chromosomes. Chromosomes are generated by using random number generator.

Step 2: Converting the chromosome's genotypes to its phenotype (i.e. binary string into decimal value). In the binary string the most significant bit is sign bit. Its weight is $-2*(n-1)$ and other bits are magnitude bits their weights are $2*(n-1)$.

Step 3: Evaluate the objective function $f(x)=x*x+5*x+6$. For each chromosome:

- Convert the value of the objective function into fitness. Here for this problem fitness is simply equal to the value of the objective function.
- If $f(x)==0$ for a particular chromosome, that chromosome is required accurate solution. Now display the value of chromosome and stop. Otherwise perform next generation by continuing following steps.

Step 4: Implementation of selection operation. For this problem the tournament selection is adopted. The tournament selection is implemented as follows: Take any two chromosomes randomly and select one with min. Fitness for next generation. This process has to be repeated till we get 10 chromosomes.

Step 5: Implementation of crossover operation on new population. Take chromosome 1 and 2 randomly fix the cut-point position and randomly decide left or right crossover and interchange the bits and the resulting chromosomes are used in the next generation. Repeat the above process for chromosome pair (3,4), pair (5,6), pair (7,8) and pair (9,10). This crossover operation generates 10 new chromosomes for the next generation.

Step 6: Jump to Step 2 (i.e. perform next generation).

Source Code

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
int f(int);

void main()
{
  struct c{
    int chromosome[5];
    int decimal_val;
    int fittness;
  };
  struct c ipop[10], newpop[10];
  int i,j,cut,gen,t,flag,num,s1,s2;
  clrscr();

  /* generating Initial population */
  randomize();
  for(i=0;i<10; ++i)
    for(j=0; j<5; ++j)
      ipop[i].chromosome[j] = rand()%2;

  /* start of the next generation */
  gen=1;
  while(1)
  {
  /* Converting a binary string into decimal value */
  for(i=0;i<10; ++i)
  {
    num=0;
    for(j=0;j<4;++j)
      num = num+ (ipop[i].chromosome[j] * pow(2,j));
    num = num-(ipop[i].chromosome[4]*pow(2,4));
    ipop[i].decimal_val = num;
  }
```

```
/* Calculating fittness value */
for(i=0;i<10;++i)
  ipop[i].fittness = f(ipop[i].decimal_val);
printf("Generation- %1d\n", gen);
printf("Initial population- output\n");
for(i=0;i<10;++i)
{
  for(j=4; j>=0; --j)
    printf("%1d", ipop[i].chromosome[j]);
  printf(" %d", ipop[i].decimal_val);
  printf(" %d", ipop[i].fittness);
  printf("\n");
}
for(i=0;i<10; ++i)
{
  if(ipop[i].fittness ==0)
  {
    printf("stop generations\n");
    printf("result = %d\n", ipop[i].decimal_val);
    goto l1;
  }
}

/* tournament selection */
printf("tournament selection\n ");
i=0;
while(i<=9)
{
  s1 = rand()%10;
  s2 = rand()%10;
  printf("%d %d %d %d\n", s1,s2,ipop[s1].fittness, ipop[s2].
      fittness);
  getche();
  if( ipop[s1].fittness < ipop[s2].fittness)
  {
    for(j=0;j<5;++j)
    newpop[i].chromosome[j] = ipop[s1].chromosome[j];
  }
  else
  {
    for(j=0;j<5;++j)
    newpop[i].chromosome[j] = ipop[s2].chromosome[j];
  }
  i++;
}
getche();
printf("new population -output\n");
for(i=0;i<10;++i)
{
  for(j=4; j>=0; --j)
```



```
    printf("%1d", newpop[i].chromosome[j]);
  printf("\n");
}
getche();

/*crossover operation */
printf("crossover operation\n");
printf("left/right cut-point position\n");
for(i=0;i<=4;++i)
{
  flag= rand()%2;
  cut= rand()%5;
  printf("%1d %1d\n", flag, cut);
  if(flag==0) /* crossover to left of cutpoint position*/
    for(j=0;j<=cut-1;++j)
    {
      t=newpop[2*i].chromosome[j];
      newpop[2*i].chromosome[j]= newpop[(2*i+1)].chromosome[j];
      newpop[(2*i+1)].chromosome[j]= t;
    }
  else /*  crossover to the right of cutpoint position*/
    for(j=cut+1;j<=4;++j)
    {
      t=newpop[2*i].chromosome[j];
      newpop[2*i].chromosome[j]= newpop[(2*i+1)].chromosome[j];
      newpop[(2*i+1)].chromosome[j]= t;
    }
    for(j=4; j>=0; --j)
      printf("%1d", newpop[2*i].chromosome[j]);
      printf("\n");
    for(j=4; j>=0; --j)
      printf("%1d", newpop[2*i+1].chromosome[j]);
      printf("\n");
  }

  /* copy newpopulation to initial population*/
  for(i=0; i<10; ++i)
  {
    for(j=0; j<5;++j)
      ipop[i].chromosome[j] = newpop[i].chromosome[j];
    }
    gen=gen+1;
  }
  l1:
    printf("end\n");
}
int f(int x)
{
  return ( x*x + 5*x + 6);
}
```

Output

```
At the end of fifth generation, the output is
Generation- 5
Chromosome      decimalvalue  Fittnessvalue
00001                  1            12
11100                 -4             2
11100                 -4             2
00000                  0             6
00001                  1            12
00001                  1            12
10001                -15           156
01100                 12           210
11101                 -3             0
00000                  0             6
stop generations
result = -3
```

18.5 Summary

Thus in this chapter the implementation of soft computing concept using C/C++ has been dealt. The concepts of neural networks, fuzzy logic and genetic algorithm discussed in various chapters have been implemented here. C being a universal language helps in evolving the soft computing techniques and since it is portable, soft computing programs written in C for one computer can be run on another with little or no modification. With the availability of large number of functions, the programming task becomes simple. C++, an evolution of C, has helped soft computing to run in an object oriented programming environment.

18.6 Exercise Problems

1. Implement the AND function using perceptron network using a C program.
2. Write a C++ program to apply back propagation network for a pattern recognition problem.
3. Implement OR function with bipolar inputs and targets with a MADALINE neural net.
4. Write a program to create an ART 1 network to cluster seven input units and three cluster units.
5. Develop a Kohonen self-organizing feature map for a image recognition problem using a C program.
6. Write a program to implement various operations of fuzzy sets.
7. Implement the properties of fuzzy sets using a C++ program.
8. Develop a C program to perform compositional operations in fuzzy relations.
9. Maximize Rosenbrock's function using a C++ program.
10. Minimize Rastrigin's function using structure-oriented programming language.
11. Given a polynomial equation of the form $f(x) = 4x^4 + 3x^3 + 2x^2 + x + 7$. Find the roots of this polynomial using GA approach.
12. Consider a hyperbolic tangent function. Maximize it within the range $0 < x < 22/7$ using a C program. Apply two-point crossover and tournament selection process.
13. Find the roots of the quadratic equation using GA. The quadratic equation is $f(x) = 6x^2 + 5x + 3$.
14. Find the solution of the function $f(x) = \sin(7\pi x) + 10$ with the constraint $-3 < x < 3$ by using genetic algorithm.
15. Write a program to minimize "cosine" function.

MATLAB Environment for Soft Computing Techniques

19

Learning Objectives

- Discusses how soft computing techniques are implemented using MATLAB software.
- Gives brief note on the development of MATLAB software.
- Details how basic operations are carried out using MATLAB.
- An introduction SIMULINK which is a branch in MATLAB package is discussed.
- The various soft computing toolboxes in MATLAB – Neural network toolbox, Fuzzy logic toolbox, Genetic algorithm toolbox – are included with commands for ready reference of the user.
- The chapter apart from the command line functions also discusses the implementation of soft computing techniques using SIMULINK blocks and graphical user interface (GUI) toolbox.
- The chapter provides the reader several problems solved using MATLAB software for soft computing techniques.

19.1 Introduction

MATLAB (Matrix Laboratory), a product of Mathworks, is a scientific software package designed to provide integrated numeric computation and graphics visualization in high-level programming language. Cleve Moler, Chief Scientist at MathWorks, Inc., originally wrote MATLAB to provide easy access to matrix software developed in the LINPACK and EISPACK projects. The very first version was written in the late 1970s for use in courses in matrix theory, linear algebra and numerical analysis. MATLAB is therefore built upon a foundation of sophisticated matrix software, in which the basic data element is a matrix that does not require predimensioning.

MATLAB program consists of standard and specialized toolboxes allowing users to take advantage of the matrix-algorithm-based projects. MATLAB offers interactive features allowing the users a great flexibility in the manipulation of data and in the form of matrix arrays for computation and visualization. MATLAB inputs can be entered at the "command line" or from "mfiles," which contains a programming-like set of instructions to be executed by MATLAB. In the aspect of programming, MATLAB works differently from FOTRAN, C, or Basic; for example, no dimensioning required for matrix arrays and no object code file generated. MATLAB offers some standard toolboxes and many optional (at extra charges) toolboxes such as Financial Toolbox and Statistics Toolbox. Users may create their own toolboxes consisting of "mfiles" written for specific applications. MATLAB is a high-performance language for technical computing. It integrates computation, visualization and programming in an easy-to-use environment where problems and solutions are expressed in familiar mathematical notation. Typical use includes:

1. math and computation;
2. algorithm development;

3. modeling, simulation and prototyping;
4. data analysis, exploration and visualization;
5. scientific and engineering graphics;
6. application development, including graphical-user interface building.

MATLAB's two- and three-dimensional graphics are object-oriented. MATLAB is thus both an environment and a matrix/vector-oriented programming language, which enables the use to build own required tools. The main features of MATLAB are:

1. Advance algorithms for high-performance numerical computations, especially in the field of matrix algebra.
2. A large collection of predefined mathematical functions and the ability to define one's own functions.
3. Two- and three-dimensional graphics for plotting and displaying data.
4. A complete help system online.
5. Powerful matrix/vector-oriented high-level programming language for individual applications.
6. Ability to cooperate with programs written in other languages and for importing and exporting formatted data.
7. Toolboxes available for solving advanced problems in several application areas.

19.2 Getting Started with MATLAB

Clicking on the program icon on a Windows/Mac machine can start MATLAB. The command window can be used to interactively issue commands and evaluate expressions. The extensive help files are essential for obtaining information about the various commands and functions available. Typing help at the command prompt generates a list of the various function categories and toolboxes with a brief description of each. Typing help *topic* generates help on the specified topic, which is generally a MATLAB command or toolbox name. This can be used to get the syntax for different commands. A more user-friendly graphical help system is also available via the help menu.

MATLAB uses conventional notation for real numbers. Scientific notation is also accepted in the form of a real number followed by the letter e and an integer exponent. Imaginary numbers are obtained by using either i or j as a suffix. Examples of valid numbers: 5, 4.55, 1.945e–20, i, $10+15i$.

The basic mathematical operators (+, −, *, /, ^) can be used directly at the command prompt to perform calculations, as can various elementary mathematical functions. **help elfun** gives a list of the elementary mathematical functions. Remember that angles are specified in radians to functions like sin() and cos(). Examples of usage:

```
>> 8 + 3i + 5
ans = 13.0000 + 3.0000i
>> sqrt(-4)
ans = 0 + 1.4142i
```

19.2.1 Matrices and Vectors

Construction: The simplest way to construct a matrix in MATLAB is to enumerate its elements row by row within square brackets, the rows separated by semi-colons, the elements of each row separated by spaces.

These elements can be real/complex numbers or other vectors and matrices, as long as the dimensions match up. For example,

```
>> A = [1 2 3 4; 5 6 7 8]
A =
    1 2 3 4
    5 6 7 8
```

Row vectors are simply matrices with one row of elements:

```
>> x = [4 5 6 7]
x =
    4 5 6 7
```

Vectors with equally spaced elements can be constructed using the colon notation:

x = starting value: increment: maximum value

A default increment of 1 is used if the increment is omitted. If the vector has to have a specific last element, we use the linspace command:

x = linspace(*first_element, last_element, number_of_elements*)

Column vectors are constructed as matrices with several rows of one element each:

```
>> y = [1; 2; 3]
y =
    1
    2
    3
```

Equally spaced column vectors are obtained by first generating a row vector using the colon notation or the linspace command, and then applying the transpose operator. There are other built-in functions to generate specific types of matrices:

1. **eye**(m, n) generates an $m \times n$ matrix with ones on the main diagonal and zeros elsewhere. If $m = n$, **eye**(n) can be used instead.
2. **zeros**(m, n) is an $m \times n$ matrix whose elements are all zeros.
3. **ones**(m, n) is an $m \times n$ matrix whose elements are all ones.
4. **diag**(v) is a square diagonal matrix with vector v on the main diagonal.
5. **diag**(A) is a column vector formed from the main diagonal of A.

Addressing elements: The element in the ith row and jth column of a matrix A is $A(i, j)$. The subscripts i and j cannot be negative or zero. In the case of a vector x, the ith element of the vector is addressed as $x(i)$:

$$\gg A = [1\ 2\ 3;\ 7\ 8\ 9]$$
$$A(2, 1) = 7$$
$$\gg x = [4\ 5\ 6]$$
$$x(2) = 5$$

Matrix operations: Matrix addition, subtraction and multiplication are implemented using the traditional operators $+$, $-$ and $*$. The inverse of a square matrix A is given by **inv**(A). There are two matrix division operators, \ and /. If A is a non-singular matrix, then $A\backslash B$ and B/A correspond to the left and right multiplication of B by **inv**(A). The transpose of a matrix A is given by A'. The expression A' produces the conjugate transpose of A, that is it transposes A and replaces its elements by their complex conjugates. If the elements of A are real, then A' and A.' are equivalent.

Array operations: It is also possible to work with the matrix as an array, that is to perform uniform elementwise operations. The array operators ($+$, $-$, $.*$, $./$, $.^$) are used for this purpose. For example, if A and B are of the same dimensions, the expression $A. * B$ would multiply each element of A with the corresponding element of B to produce a matrix of the same dimension as A and B.

Scripts: Instead of executing individual commands at the prompt, it is possible to make a text file with a sequence of commands, that is a MATLAB script, and execute all the commands in sequence. The script must be a plain ASCII file with a ".m"' extension. When this file is in the working directory, typing the name of the file without the extension is sufficient to execute the script.

Plotting: The general form of the two-dimensional plot command is **plot**(x, y, S) where x and y are vectors of the same type and dimension and S is a string of characters within quotes which specifies plot attributes like color, line style, etc. Use **help plot** to find out options and related commands.

19.3 Introduction to Simulink

Simulink (Simulation and Link) is an extension of MATLAB by Mathworks. It works with MATLAB to offer modeling, simulating and analyzing of dynamical systems under a graphical user interface (GUI) environment. The construction of a model is simplified with click-and-drag mouse operations. Simulink includes a comprehensive block library of toolboxes for both linear and nonlinear analyses. Models are hierarchical, which allow using both top-down and bottom-up approaches. As Simulink is an integral part of MATLAB; it is easy to switch back and forth during the analysis process, and thus, the user may take full advantage of features offered in both environments. MATLAB is an interactive package for numerical analysis, matrix computation, control system design and linear system analysis and design available on most CAEN (Computer Aided Engineering Network) platforms (Macintosh, PC, Sun and Hewlett-Packard). In addition to the standard functions provided by MATLAB, there exist large set of toolboxes, or collections of functions and procedures, available as part of the MATLAB package. The toolboxes are:

1. *Control system*: It provides several features for advanced control system design and analysis.
2. *Communications*: It provides functions to model the components of a communication system's physical layer.
3. *Signal processing*: It contains functions to design analog and digital filters and apply these filters to data and analyze the results.

4. *System identification*: It provides features to build mathematical models of dynamical systems based on observed system data.
5. *Robust control*: It allows users to create robust multivariable feedback control system designs based on the concept of the singular-value Bode plot.
6. *Simulink*: It allows you to model dynamic systems graphically.
7. *Neural network*: It allows you to simulate neural networks.
8. *Fuzzy logic*: It allows for manipulation of fuzzy systems and membership functions.
9. *Image processing*: It provides access to a wide variety of functions for reading, writing, and filtering images of various kinds in different ways.
10. *Analysis*: It includes a wide variety of system analysis tools for varying matrices.
11. *Optimization*: It contains basic tools for use in constrained and unconstrained optimization problems.
12. *Spline*: It can be used to find approximate functional representations of data sets.
13. *Symbolic*: It allows for symbolic (rather than purely numeric) manipulation of functions.
14. *User interface utilities*: It includes tools for creating dialog boxes, menu utilities and other user interaction for script files.

In MATLAB command window, enter: > > **simulink** and press ENTER to invoke Simulink. A Simulink library browser window would appear as shown in Figure 19-1.

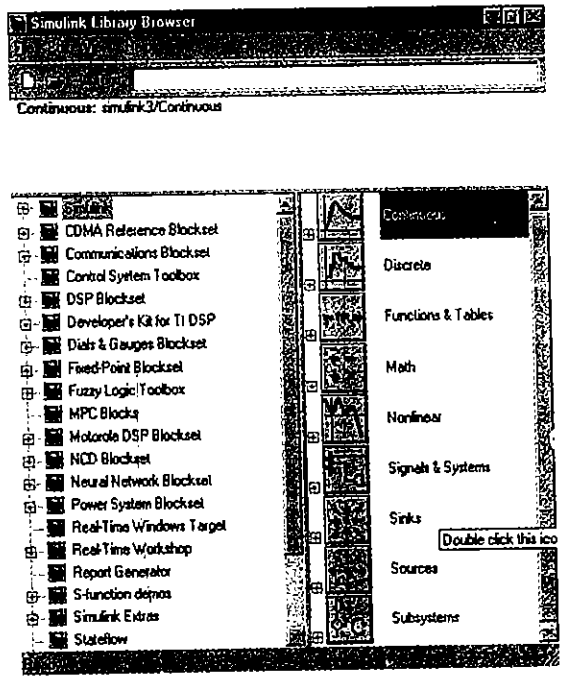


Figure 19-1 Simulink library browser.

19.4 MATLAB Neural Network Toolbox

The MATLAB neural network toolbox provides a complete set of functions and a GUI for the design, implementation, visualization and simulation of neural networks. It supports the most commonly used supervised and unsupervised network architectures and a comprehensive set of training and learning functions. The neural network toolbox extends the MATLAB computing environment to provide tools for the design, implementation, visualization and simulation of neural networks. Neural networks are uniquely powerful tools that are used in applications where formal analysis would be difficult or impossible, such as pattern recognition and nonlinear system identification and control.

Salient Features:

1. GUI for creating, training and simulation of neural networks.
2. Set of training and learning functions.
3. Automatic generation of Simulink models from neural network objects.
4. Pre- and post-processing functions for improving network training and assessing network performance.
5. Routines for improving generalization.
6. Visualization functions for viewing network performance.

19.4.1 Creating a Custom Neural Network

The command NETWORK creates a custom neural network.

Synopsis:

net = network

net = network(numInputs,numLayers,biasConnect,
inputConnect, layerConnect,outputConnect,targetConnect)

Description:

NETWORK creates new custom networks. It is used to create networks that are then customized by functions such as NEWP, NEWLIN, NEWFF, etc.

NETWORK takes the following optional arguments (shown with default values):

numInputs: Number of inputs, 0.

numLayers: Number of layers, 0.

biasConnect: numLayers-by-1 Boolean vector, zeros.

inputConnect: numLayers-by-numInputs Boolean matrix, zeros.

layerConnect: numLayers-by-numLayers Boolean matrix, zeros.

outputConnect: 1-by-numLayers Boolean vector, zeros.

targetConnect: 1-by-numLayers Boolean vector, zeros.

and returns,

NET: New network with the given property values.

Properties

Architecture properties

net.numInputs: 0 or a positive integer.
Number of inputs.

net.numLayers: 0 or a positive integer.
Number of layers.

net.biasConnect: numLayer-by-1 Boolean vector.

If net.biasConnect(i) is 1 then the layer i has a bias and
net.biases{i} is a structure describing that bias.

net.inputConnect: numLayer-by-numInputs Boolean vector.

If net.inputConnect(i,j) is 1 then layer i has a weight coming from
input j and net.inputWeights{i,j} is a structure describing that weight.

net.layerConnect: numLayer-by-numLayers Boolean vector.

If net.layerConnect(i,j) is 1 then layer i has a weight coming from
layer j and net.layerWeights{i,j} is a structure describing that weight.

net.outputConnect: 1-by-numLayers Boolean vector.

If net.outputConnect(i) is 1 then the network has an output from layer i and net.outputs{i} is a structure describing that output.

net.targetConnect: 1-by-numLayers Boolean vector.

If net.outputConnect(i) is 1 then the network has a target from layer i and net.targets{i} is a structure describing that target.

net.numOutputs: 0 or a positive integer. Read only.
Number of network outputs according to net.outputConnect.

net.numTargets: 0 or a positive integer. Read only.
Number of targets according to net.targetConnect.

net.numInputDelays: 0 or a positive integer. Read only.
Maximum input delay according to all

net.inputWeight{i,j}.delays.

net.numLayerDelays: 0 or a positive number. Read only.
Maximum layer delay according to all Net.layerWeight{i,j}.delays.

Subobject structure properties

net.inputs: numInputs-by-1 cell array.

net.inputs{i} is a structure defining input i:

net.layers: numLayers-by-1 cell array.

net.layers{i} is a structure defining layer i:

net.biases: numLayers-by-1 cell array.

if net.biasConnect(i) is 1, then net.biases{i} is a structure defining the bias for layer i.

net.inputWeights: numLayers-by-numInputs cell array.

if net.inputConnect(i,j) is 1, then net.inputWeights{i,j} is a structure defining the weight to layer i from input j.

net.layerWeights: numLayers-by-numLayers cell array.

if net.layerConnect(i,j) is 1, then net.layerWeights{i,j} is a structure defining the weight to layer i from layer j.

net.outputs: 1-by-numLayers cell array.

if net.outputConnect(i) is 1, then net.outputs{i} is a structure defining the network output from layer i.

net.targets: 1-by-numLayers cell array.

if net.targetConnect(i) is 1, then net.targets{i} is a structure defining the network target to layer i.

Function properties:

net.adaptFcn: name of a network adaption function

net.initFcn: name of a network initialization function

net.performFcn: name of a network performance function

net.trainFcn: name of a network training function or

Parameter properties:

net.adaptParam: network adaption parameters.

net.initParam: network initialization parameters.

net.performParam: network performance parameters.

net.trainParam: network training parameters.

Weight and bias value properties:

net.IW: numLayers-by-numInputs cell array of input weight values.

net.LW: numLayers-by-numLayers cell array of layer weight values.

net.b: numLayers-by-1 cell array of bias values.

Other properties:

net.userdata: structure you can use to store useful values.

19.4.2 Commands in Neural Network Toolbox

The various commands used in the neural network toolbox are as follows:

Graphical user interface functions:
nntool: Neural network toolbox graphical user interface.

Analysis functions:
errsurf: Error surface of single input neuron.
maxlinlr: Maximum learning rate for a linear layer.

Distance functions:
boxdist: Box distance function.
dist: Euclidean distance weight function.
mandist: Manhattan distance weight function.
linkdist: Link distance function.

Layer initialization functions:
initnw: Nguyen-Widrow layer initialization function.
initwb: By-weight-and-bias layer initialization function.

Learning functions:
learncon: Conscience bias learning function.
learngd: Gradient descent weight/bias learning function.
learngdm: Gradient descent w/momentum weight/bias learning function.
learnh: Hebb weight learning function.
learnhd: Hebb with decay weight learning function.
learnis: Instar weight learning function.
learnk: Kohonen weight learning function.
learnlv1: LVQ1 weight learning function.
learnlv2: LVQ2 weight learning function.
learnos: Outstar weight learning function.
learnp: Perceptron weight/bias learning function.
learnpn: Normalized perceptron weight/bias learning function.
learnsom: Self-organizing map weight learning function.
learnwh: Widrow-Hoff weight/bias learning rule.

Line search functions:
srchbac: Backtracking search.
srchbre: Brent's combination golden section/quadratic interpolation.
srchcha: Charalambous' cubic interpolation.
srchgol: Golden section search.
srchhyb: Hybrid bisection/cubic search.

New networks:
network: Create a custom neural network.
newc: Create a competitive layer.
newcf: Create a cascade-forward backpropagation network.
newelm: Create an Elman backpropagation network.
newff: Create a feed-forward backpropagation network.
newfftd: Create a feed-forward input-delay backprop network.
newgrnn: Design a generalized regression neural network.
newhop: Create a Hopfield recurrent network.
newlin: Create a linear layer.
newlind: Design a linear layer.
newlvq: Create a learning vector quantization network.
newp: Create a perceptron.
newpnn: Design a probabilistic neural network.
newrb: Design a radial basis network.
newrbe: Design an exact radial basis network.
newsom: Create a self-organizing map.

Net input functions:
netprod: Product net input function.
netsum: Sum net input function.

Net input derivative functions:
dnetprod: Product net input derivative function.
dnetsum: Sum net input derivative function.

Network initialization functions:
initlay: Layer-by-layer network initialization function.

Performance functions:
mae: Mean absolute error performance function.
mse: Mean squared error performance function.
msereg: Mean squared error with regularization performance function.
sse: Sum squared error performance function.

Performance derivative functions:
dmae: Mean absolute error performance derivatives function.
dmse: Mean squared error performance derivatives function.
dmsereg: Mean squared error w/reg performance derivative function.
dsse: Sum squared error performance derivative function.

Plotting functions:
hintonw: Hinton graph of weight matrix.
hintonwb: Hinton graph of weight matrix and bias vector.
plotbr: Plot network performance for Bayesian regularization training.
plotes: Plot an error surface of a single input neuron.
plotpc: Plot classification line on perceptron vector plot.
plotpv: Plot perceptron input/target vectors.
plotep: Plot a weight-bias position on an error surface.
plotperf: Plot network performance.
plotsom: Plot self-organizing map.
plotv: Plot vectors as lines from the origin.
plotvec: Plot vectors with different colors.

Pre- and post-processing:
prestd: Normalize data for unity standard deviation and zero mean.
poststd: Unnormalize data which has been normalized by PRESTD.
trastd: Transform data with precalculated mean and standard deviation.
premnmx: Normalize data for maximum of 1 and minimum of −1.
postmnmx: Unnormalize data which has been normalized by PREMNMX.
tramnmx: Transform data with precalculated minimum and maximum.
prepca: Principal component analysis on input data.
trapca: Transform data with PCA matrix computed by PREPCA.
postreg: Post-training regression analysis.

Simulink support:
gensim: Generate a Simulink block to simulate a neural network.

Topology functions:
gridtop: Grid layer topology function.
hextop: Hexagonal layer topology function.
randtop: Random layer topology function.

Transfer functions:
compet: Competitive transfer function.
hardlim: Hard limit transfer function.

hardlims: Symmetric hard limit transfer function.
logsig: Log sigmoid transfer function.
poslin: Positive linear transfer function.
purelin: Linear transfer function.
radbas: Radial basis transfer function.
satlin: Saturating linear transfer function.
satlins: Symmetric saturating linear transfer function.
softmax: Soft max transfer function.
tansig: Hyperbolic tangent sigmoid transfer function.
tribas: Triangular basis transfer function.

Training functions:
trainb: Batch training with weight and bias learning rules.
trainbfg: BFGS quasi-Newton backpropagation.
trainbr: Bayesian regularization.
trainc: Cyclical order incremental training w/learning functions.
traincgb: Powell-Beale conjugate gradient backpropagation.
traincgf: Fletcher-Powell conjugate gradient backpropagation.
traincgp: Polak-Ribiere conjugate gradient backpropagation.
traingd: Gradient descent backpropagation.
traingdm: Gradient descent with momentum backpropagation.
traingda: Gradient descent with adaptive lr backpropagation.
traingdx: Gradient descent w/momentum and adaptive lr backpropagation.
trainlm: Levenberg-Marquardt backpropagation.
trainoss: One step secant backpropagation.
trainr: Random order incremental training w/learning functions.
trainrp Resilient backpropagation (Rprop).
trains: Sequential order incremental training w/learning functions.
trainscg: Scaled conjugate gradient backpropagation.

Transfer derivative functions:
dhardlim: Hard limit transfer derivative function.
dhardlms: Symmetric hard limit transfer derivative function.
dlogsig: Log sigmoid transfer derivative function.
dposlin: Positive linear transfer derivative function.
dpurelin: Hard limit transfer derivative function.
dradbas: Radial basis transfer derivative function.
dsatlin: Saturating linear transfer derivative function.
dsatlins: Symmetric saturating linear transfer derivative function.
dtansig: Hyperbolic tangent sigmoid transfer derivative function.
dtribas: Triangular basis transfer derivative function.

Update networks from previous versions:
nnt2c: Update NNT 2.0 competitive layer.
nnt2elm: Update NNT 2.0 Elman backpropagation network.
nnt2ff: Update NNT 2.0 feed-forward network.
nnt2hop: Update NNT 2.0 Hopfield recurrent network.

nnt2lin: Update NNT 2.0 linear layer.
nnt2lvq: Update NNT 2.0 learning vector quantization network.
nnt2p: Update NNT 2.0 perceptron.
nnt2rb: Update NNT 2.0 radial basis network.
nnt2som: Update NNT 2.0 self-organizing map.

Using networks:
sim: Simulate a neural network.
init: Initialize a neural network.
adapt: Allow a neural network to adapt.
train: Train a neural network.
disp: Display a neural network's properties.
display: Display the name and properties of a neural network variable.

Vectors:
cell2mat: Combine cell array of matrices into one matrix.
concur: Create concurrent bias vectors.
con2seq: Convert concurrent vectors to sequential vectors.
combvec: Create all combinations of vectors.
ind2vec: Convert indices to vectors.
mat2cell: Break matrix up into cell array of matrices.
minmax: Ranges of matrix rows.
nncopy: Copy matrix or cell array.
normc: Normalize columns of a matrix.
normr: Normalize rows of a matrix.
pnormc: Pseudo-normalize columns of a matrix.
quant: Discretize values as multiples of a quantity.
seq2con: Convert sequential vectors to concurrent vectors.
sumsqr: Sum squared elements of matrix.
vec2ind: Convert vectors to indices.

Weight functions:
dist: Euclidean distance weight function.
dotprod: Dot product weight function.
mandist: Manhattan distance weight function.
negdist: Dot product weight function.
normprod: Normalized dot product weight function.

Weight and bias initialization functions:
initcon: Conscience bias initialization function.
initzero: Zero weight/bias initialization function.
midpoint: Midpoint weight initialization function.
randnc: Normalized column weight initialization function.
randnr: Normalized row weight initialization function.
rands: Symmetric random weight/bias initialization function.

Weight derivative functions:
ddotprod: Dot product weight derivative function.

Train and adapt
There are two types of training that are given below.

1. *Incremental training*: updating the weights after the presentation of each single training sample.
2. *Batch training*: updating the weights after each presenting the complete data set.

When using **adapt**, both incremental and batch training can be used. When using **train**, on the other hand, only batch training can be used, regardless of the format of the data. The big plus point of **train** is that it gives you a lot more choice in training functions (gradient descent, gradient descent w/momentum, Levenberg-Marquardt, etc.) which are implemented very efficiently.

The difference between **train** and **adapt** is similar as *the difference between passes and epochs*. When using **adapt**, the property that determines how many times the complete training data set is used for training the network is called **net.adaptParam.passes**. But, when using **train**, the same property is called **net.trainParam.epochs**.

```
>> net.trainFcn = 'traingdm';
>> net.trainParam.epochs = 1000;
>> net.adaptFcn = 'adaptwb';
>> net.adaptParam.passes = 10;
```

19.4.3 Neural Network Graphical User Interface Toolbox

A GUI can be used to

1. create networks;
2. create data;
3. train the networks;
4. export the networks;
5. export the data to the command line workspace.

The salient features of GUI are the following:

1. It is designed to be simple and user-friendly. This tool lets you import potentially large and complex data sets.
2. It also enables you to create, initialize, train, simulate and manage the networks. It has the GUI **Network/Data Manager** window.
3. The window has its own work area, separate from the more familiar command line workspace. Thus, when using the GUI, one might "export" the GUI results to the (command line) workspace. Similarly, one might "import" results from the command line workspace to the GUI.
4. Once the **Network/Data Manager** is up and running, create a network, view it, train it, simulate it and export the final results to the workspace. Similarly, import data from the workspace for use in the GUI.

This tool lets you import potentially large and complex data sets. The GUI also enables you to create, initialize, train, simulate and manage your networks. Simple graphical representations allow you to visualize and understand network architecture (see Figure 19-2).

The following example deals with a perceptron network. It gives a step-by-step procedure of creating a network.

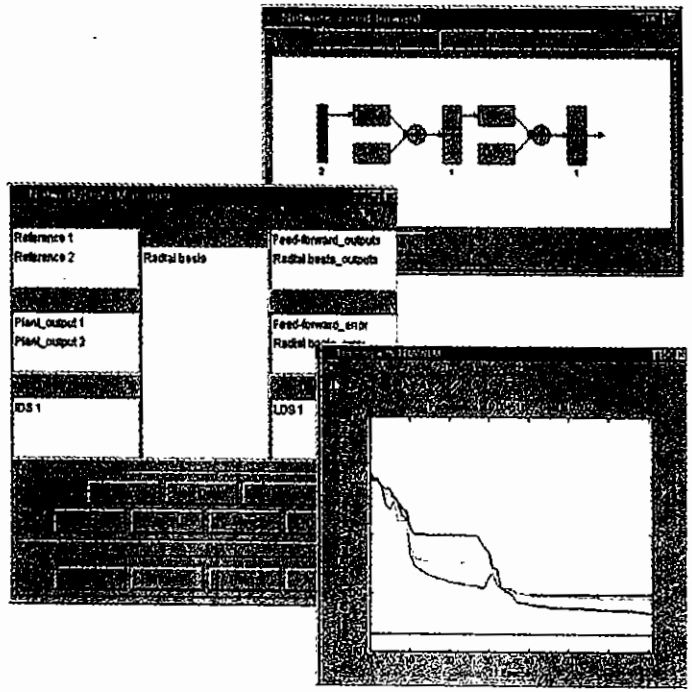


Figure 19-2 This window displays portions of the neural network GUI. Dialogs and panes allow you to visualize your network (top), evaluate training results (bottom), and manage your networks (center).

Create a Perceptron Network (NN Tool)

Create a perceptron network to perform the OR function in this example. It has an input vector data 1 = [0 0 1 1; 0 1 0 1] and a target vector data 2 = [0 1 1 1]. The network can be called OR Net. Once created, the network will be trained. Then save the network, its output, etc. by "exporting" it to the command line.

Input and target:

To start, type NN tool. The window shown in Figure 19-3 appears.

First step is to define the network input, data 1, as having the particular value [0 0 1 1; 0 1 0 1]. Thus, the network had a two-element input and four sets of such two-element vectors are presented to it in training. Create New Data appears. Set the Name to data 1, the Value to [0 0 1 1; 0 1 0 1], and make sure that Data Type is set to Inputs. The Create New Data window will then look like the one shown in Figure 19-4.

Now click Create to actually create an input file data 1. The Network/Data Manager window comes up and data 1 is shown as an input.

Step 2 is to create a network target. Click on New Data again, and this time enter the variable name data 2, specify the value [0 1 1 1] and click on Target under data type. The window will look the one given in Figure 19-5.

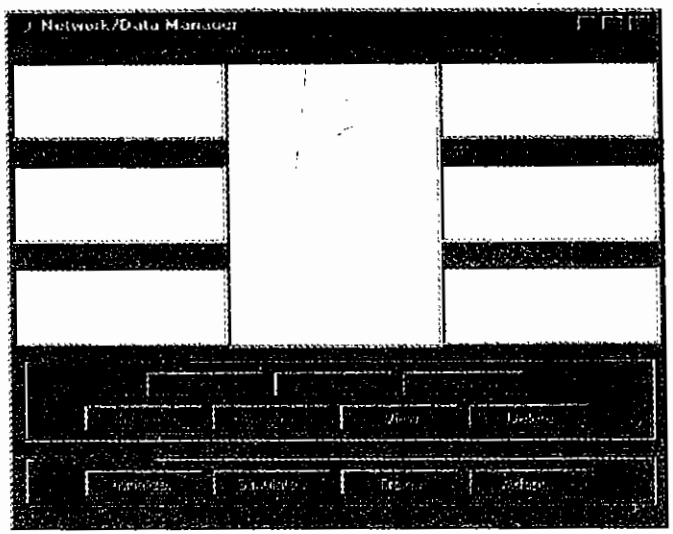


Figure 19-3 Neural network launch pad window network manager.

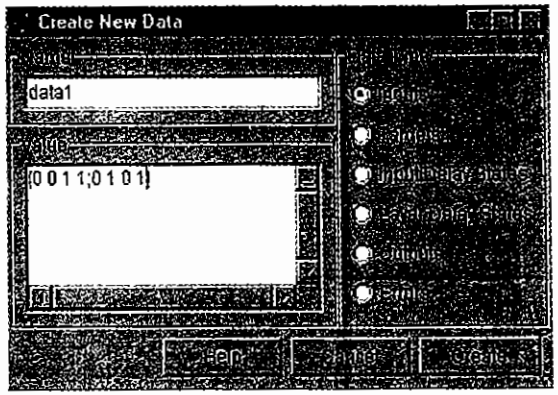


Figure 19-4 Window for creating new data (inputs).

Again click on Create and see the resulting Network/Data Manager window that has data 2 as a target as well as the previous data 1 as an input.

Create network:

Now we want to create a new network, which is OR NET. To do this, click on New Network, and a Create New Network window appears. Enter OR NET under Network Name. Set the Network Type to Perceptron, because that is the kind of network to create. The input ranges can be set by entering numbers in that field, but it is easier to get them from the particular input data that you want to use. To do this, click on the down

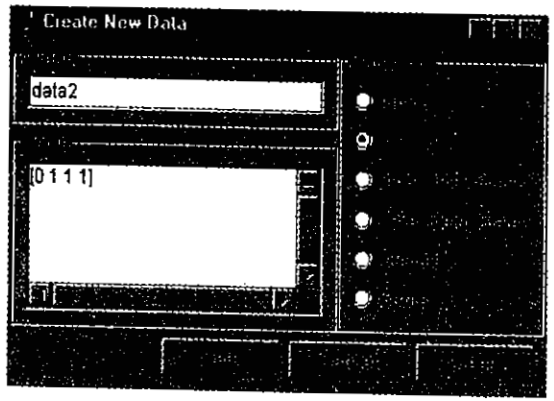


Figure 19-5 Window for creating new data (targets).

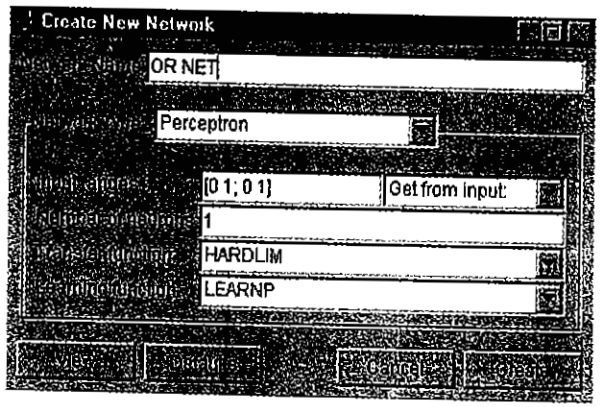


Figure 19-6 Window for creating new network.

arrow at the right side of **Input Range**. This pull-down menu shows that you can get the input ranges from the file p. This should lead to input ranges [0 1; 0 1]. We want to use a hardlim transfer function and a learnp learning function, so set those values using the arrows for **Transfer function** and **Learning function**, respectively. By now your **Create New Network** window should look like the one given in Figure 19-6.

Next you might look at the network by clicking on **View** (Figure 19-7). Figure 19-7 shows a network with a single input (composed of two elements), a hardlim transfer function and a single output which is going to be created. This is the perceptron network. Now click **Create** to generate the network. Now go back to the **Network/Data Manager** window. Note that OR NET is now listed as a network.

Train the perceptron:

To train the network, click on OR NET to highlight it. Then click on **Train**. This leads to a new window labeled **Network:OR NET** (Figure 19-8). At this point you can view the network again by clicking on the top tab **Train**. You can also check on the initialization by clicking on the top tab **Initialize**. Now click on the

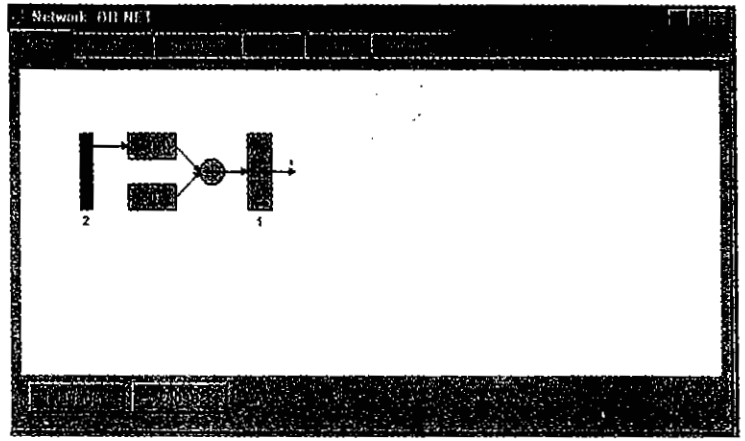


Figure 19-7 Window to view the new architecture created.

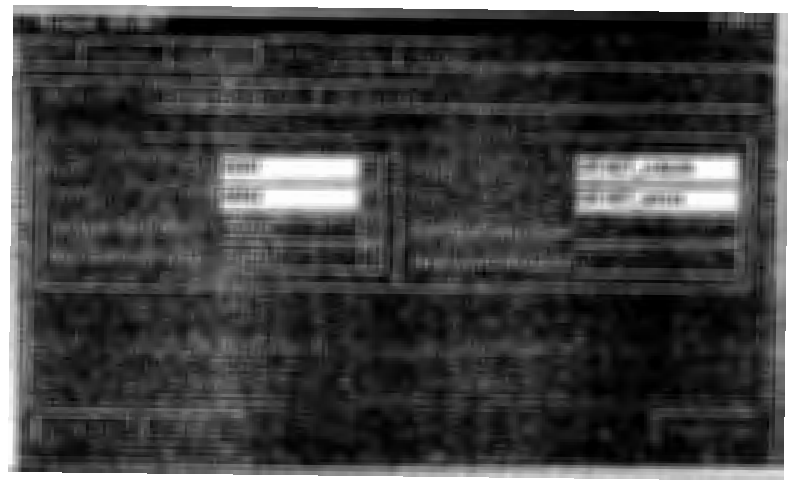


Figure 19-8 Perceptron net for AND function.

top tab **Train**. Specify the inputs and outputs by clicking on the left tab **Training Info** and selecting data1 from the pop-down list of inputs and data2 from the pull-down list of targets (Figure 19-8).

On clicking the **Training Parameters** tab, it shows parameters such as the epochs and error goal. These parameters can be changed at this point if desired. Now click **Train Network** to train the perceptron network and see the following training results (Figure 19-9).

Thus, the network was trained to zero error in four epochs. (Note that other kinds of networks commonly do not train to zero error and their errors commonly cover a much larger range. On that account, plot their errors on a log scale rather than on a linear scale such as that used above for perceptrons.)

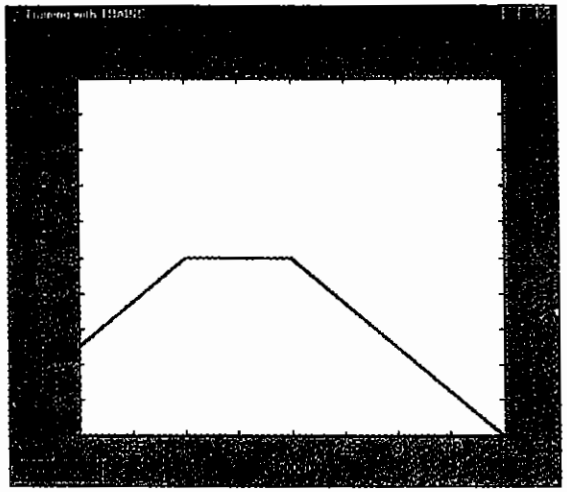


Figure 19-9 Training results of OR function.

To check that the trained network does indeed give zero error by using the input p and simulating the network, go to the **Network/Data Manager** window and click on **Network Only: Simulate**. This will bring up the **Network: OR NET** window. Click there on **Simulate**. Now use the **Input** pull-down menu to specify data 1 as the input, and label the output as OR NET_outputsSim to distinguish it from the training output. Now click **Simulate Network** in the lower-right corner. Look at the **Network/Data Manager**. It will show a new variable in the output: OR NET_outputsSim. Double-click on it and a small window Data:OR NET_outputsSim appears with the value [0 1 1 1]. Thus, the network does perform the OR of the inputs, giving 0 as an output only in this first case, when both inputs are 0.

19.5 Fuzzy Logic MATLAB Toolbox

Fuzzy logic in MATLAB can be dealt very easily because of the existing new Fuzzy Logic Toolbox. This provides a complete set of functions to design and implement various fuzzy logic processes. The major fuzzy logic operations include fuzzification, defuzzification and the fuzzy inference. These all are performed by means of various functions and can be even implemented using the GUI. Many of the applications can be simulated using the "fuzzy logic controller" simulink block present in Matlab Simulink toolbox. The features are the following.

1. It provides tools to create and edit Fuzzy Inference Systems (FIS).
2. It allows integrating fuzzy systems into simulation with SIMULINK.
3. It is possible to create stand-alone C programs that call on fuzzy systems built with MATLAB.

The Toolbox provides three categories of tools:

1. command line functions;
2. graphical or interactive tools;
3. SIMULINK blocks.

19.5.1 Commands in Fuzzy Logic Toolbox

The various commands in Fuzzy Logic Toolbox to be operated in command line are as follows:

GUI editors:

anfisedit:	ANFIS training and testing UI tool
findcluster:	Clustering UI tool.
fuzzy:	Basic FIS editor.
mfedit:	Membership function editor.
ruleedit:	Rule editor and parser.
ruleview:	Rule viewer and fuzzy inference diagram.
surfview:	Output surface viewer.

Membership functions:

dsigmf:	Difference of two sigmoid membership functions.
gauss2mf:	Two-sided Gaussian curve membership function.
gaussmf:	Gaussian curve membership function.
gbellmf:	Generalized bell-shaped curve membership function.
pimf:	Pi-shaped curve membership function.
psigmf:	Product of two sigmoid membership functions.
smf:	S-shaped curve membership function.
sigmf:	Sigmoid curve membership function.
trapmf:	Trapezoidal membership function.
trimf:	Triangular membership function.
zmf:	Z-shaped curve membership function.

Command line FIS functions:

addmf:	Add membership function to FIS.
addrule:	Add rule to FIS.
addvar:	Add variable to FIS.
defuzz:	Defuzzify membership function.
evalfis:	Perform fuzzy inference calculation.
evalmf:	Generic membership function evaluation.
gensurf:	Generate FIS output surface.
getfis:	Get fuzzy system properties.
mf2mf:	Translate parameters between functions.
newfis:	Create new FIS.
parsrule:	Parse fuzzy rules.
plotfis:	Display FIS input-output diagram.
plotmf:	Display all membership functions for one variable.
readfis:	Load FIS from disk.
rmmf:	Remove membership function from FIS.
rmvar:	Remove variable from FIS.
setfis:	Set fuzzy system properties.
showfis:	Display annotated FIS.
showrule:	Display FIS rules.
writefis:	Save FIS to disk.

Advanced techniques:

anfis:	Training routine for Sugeno-type FIS (MEX only).
fcm:	Find clusters with fuzzy c-means clustering.
genfis1:	Generate FIS matrix using generic method.
genfis2:	Generate FIS matrix using subtractive clustering.
subclust:	Estimate cluster centers with subtractive clustering.

Miscellaneous functions:

convertfis:	Convert v1.0 fuzzy matrix to v2.0 fuzzy structure.
discfis:	Discretize a fuzzy inference system.
evalmmf:	For multiple membership functions evaluation.
fstrvcat:	Concatenate matrices of varying size.
fuzarith:	Fuzzy arithmatic function.
findrow:	Find the rows of a matrix that match the input string.
genparam:	Generates initial premise parameters for ANFIS learning.
probor:	Probabilistic OR.
sugmax:	Maximum output range for a Sugeno system.

GUI helper files:

cmfdlg:	Add customized membership function dialog.
cmthdlg:	Add customized inference method dialog.
fisgui:	Generic GUI handling for the Fuzzy Logic Toolbox.
gfmfdlg:	Generate FIS using grid partition method dialog.
mfdlg:	Add membership function dialog.
mfdrag:	Drag membership functions using mouse.
popundo:	Pull the last change off the undo stack.
pushundo:	Push the current FIS data onto the undo stack.
savedlg:	Save before closing dialog.
statmsg:	Display messages in a status field.
updtfis:	Update Fuzzy Logic Toolbox GUI tools.
wsdlg:	Open from/save to workspace dialog.

19.5.2 Simulink Blocks in Fuzzy Logic Toolbox

Once fuzzy system is created using GUI tools or some other method, it can be directly embedded into SIMULINK using the fuzzy logic controller (FLC) block as shown in Figure 19-10.

Make sure that the FIS matrix corresponding to the fuzzy system is both in the MATLAB workspace and referred to by name in the dialog box associated with this FLC. Although it is possible to use the Fuzzy

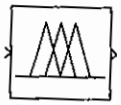


Figure 19-10 Fuzzy logic controller Simulink block.

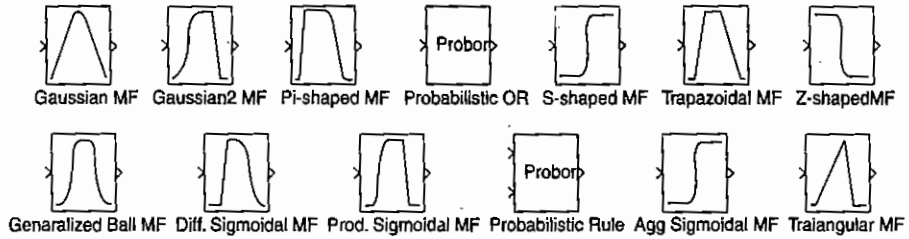


Figure 19-11 Membership functions.

Logic Toolbox by working strictly from the command line, in general it is much easier to build a system up graphically so that GUI tools are commonly used for building, editing and observing FIS.

The process of mapping from a given input to an output using fuzzy logic involves membership functions, fuzzy logic operators and If-Then rules.

Membership functions: This toolbox includes 11 built-in membership function types, built from several basic functions: piecewise linear functions *(triangular and trapezoidal)*, the Gaussian distribution function (*Gaussian curves and generalized bell)*, the sigmoid curve, and quadratic and cubic polynomial curves (*Z, S, and Pi curves*) (Figure 19-11).

Fuzzy logic operators: According to the fuzzy logical operations, any number of well-defined methods can fill in for the AND operation or the OR operation. In the Fuzzy Logic Toolbox, two built-in *AND methods* are supported: *min* (minimum) and *prod* (algebraic product). Two built-in *OR methods* are also supported: *max* (maximum) and the *probor* (probabilistic OR, also known as algebraic sum).

Based on implication method, two built-in methods are supported. These are the same functions that are used by the AND method, so that, *min* method truncates the output fuzzy set and *prod* scales the output fuzzy set.

Based on aggregation method, three built-in methods are supported: *max* (maximum), *probor* (probabilistic OR) and *sum* (simply the sum of each rule's output set).

Although centroid calculation is the most popular defuzzification method, there are five built-in methods supported: *centroid, bisector, middle of maximum, largest of maximum and smallest of maximum.*

If-Then rules: Since rules can be edited in three different formats (*verbose, symbolic* and *indexed*), verbose format makes the system easier to interpret. Every rule has a *weight* (a number between 0 and 1) which is applied to the number given by the antecedent. Generally this weight is 1 and so it has no effect at all on the implication process. For example, let us enter a sample rule (rule number one):

Verbose format: 1. if Temperature is warm then Sky is grey (1)

Symbolic format: 1. (Temperature = = warm) =>Sky = grey (1)

Indexed format: 1, 1 (1): 1

Here the first "1" corresponds to the input variable, the second corresponds to the output variable, the third displays the *weight* applied to each rule and the fourth is shorthand that indicates whether this is an OR (2) rule or an AND (1) rule. So a literal interpretation of rule number one is: "if input1 is MF1 (the first membership function associated with input 1) then output1 should be MF1 (the first membership function associated with output 1) with the weight 1". Note that as long as the aggregation method is commutative, the order in which the rules are executed is not important.

Once an FLC is created, it can be saved on a disk (FIS-file is created, i.e., juggler.fis) as an ASCII text format so that it can be edited and modified. An FLC can also be saved into MATLAB workspace as a matrix

variable (FIS matrix) so that it can be modified; however, its representation is extremely different from FIS-file representation.

19.5.3 Fuzzy Logic GUI Toolbox

The fuzzy logic can be simulated in MATLAB using GUI. On typing "fuzzy" in the command prompt, the fuzzy GUI toolbox opens up. The main windows corresponding to **Fuzzy GUI tools** are shown in Figures 19-12–19-17.

In Figure 19-12 the FIS editor – Mamdani or Sugeno model – is selected. The inference mechanism can be selected at this step. The various mechanisms to be selected in the FIS editor are AND method, OR method, implication, aggregation and defuzzification. Here, the number of input and output variables can be specified.

The membership function editor is shown in Figure 19-13. In this editor, for the input variable and the corresponding output variable, the membership functions using linguistic variables along with their range are defined. Figure 19-13 shows the membership function editor for the input variable and Figure 19-14 shows the membership function editor for the output variable.

The rules to be formed based on the input variables to get the output are defined in the rule editor. The inference of these rules gives the fuzzified output of the problem under consideration. For rule definition either AND connective or OR connective can be used. Figure 19-15 shows a rule editor with 3 rules formulated. The formulated rules can be viewed in the rule viewer as shown in Figure 19-16. On viewing these rules, information about the output can be obtained.

Figure 19-17 shows the surface view of the defined fuzzy inference editor. The fuzzy logic GUI toolbox helps us in designing a suitable FLC module for any application.

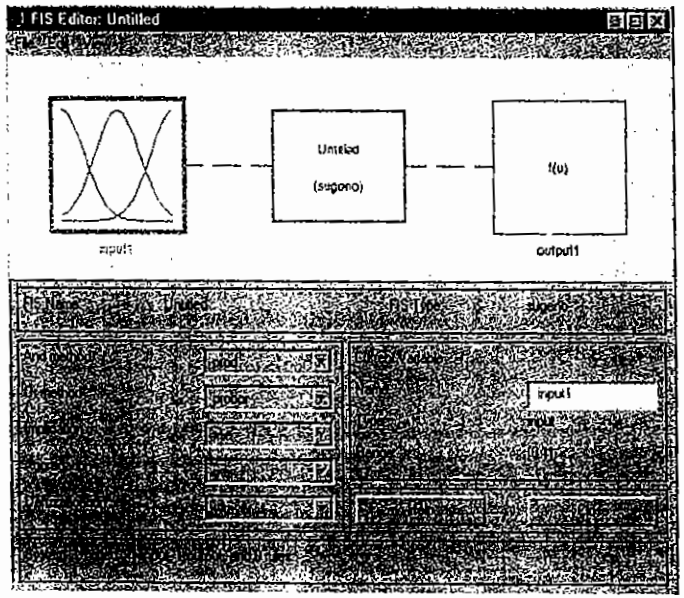


Figure 19-12 Basic FIS editor.

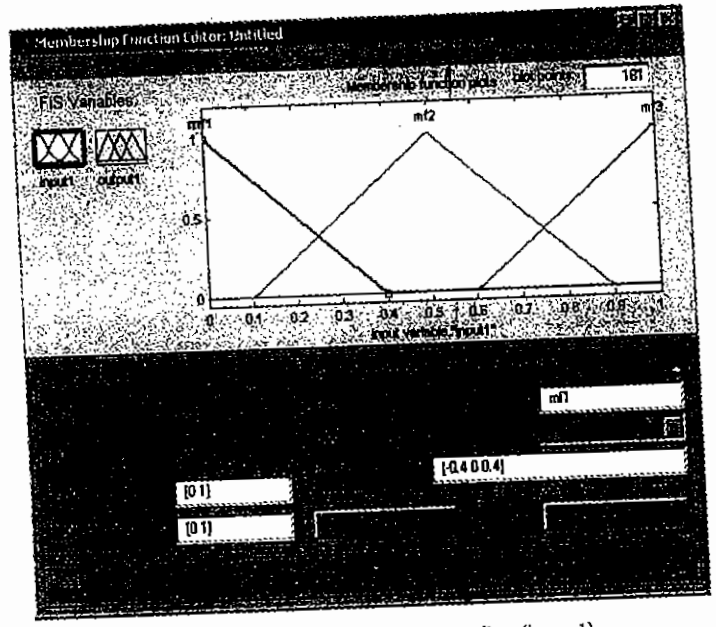


Figure 19-13 Membership function editor (input 1).

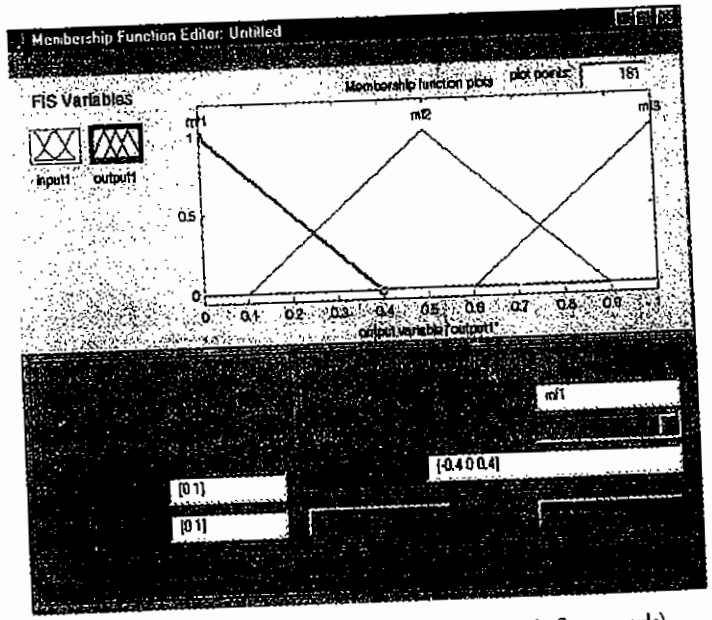


Figure 19-14 Membership function editor (output 1, in Sugeno-style).

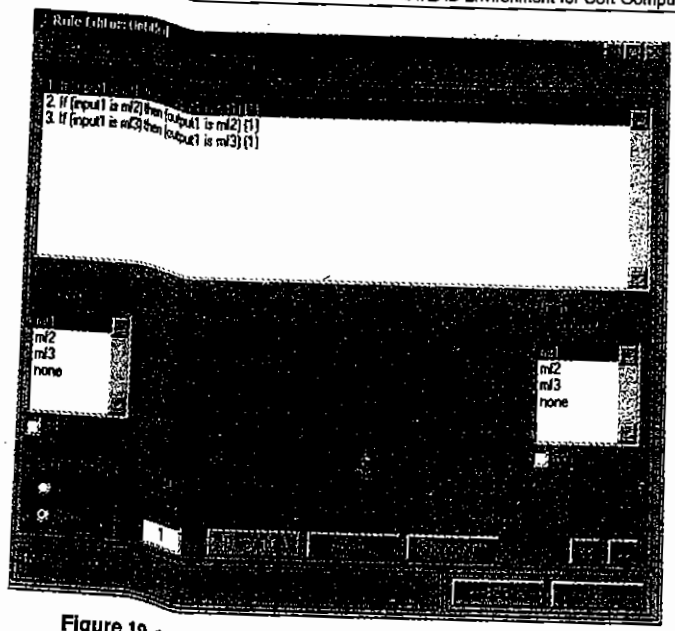


Figure 19-15 Rule editor (sample with 3 rules in verbose format).

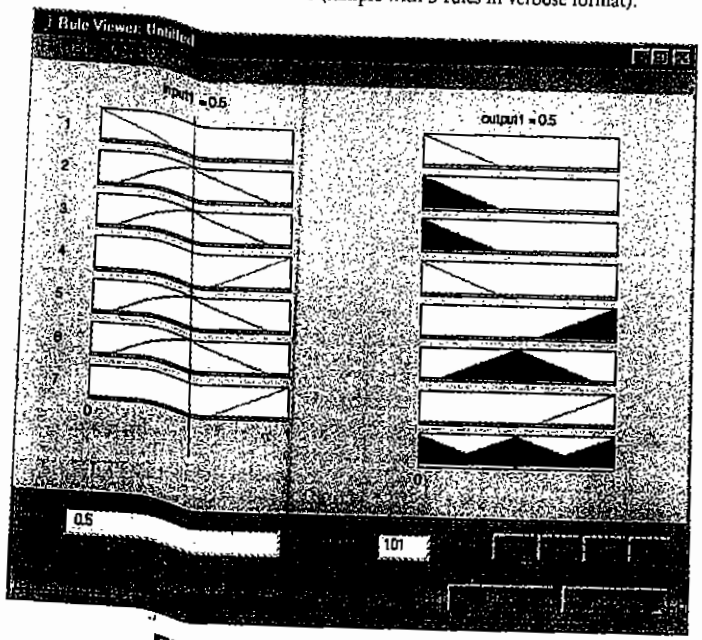


Figure 19-16 Rule viewer (in Sugeno-style).

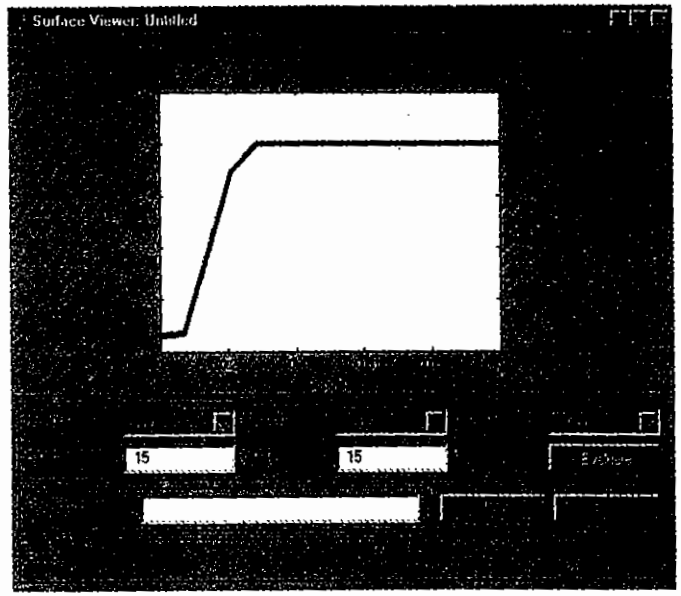


Figure 19-17 Surface viewer (input 1 and input 2 versus output 1).

19.6 Genetic Algorithm MATLAB Toolbox

The genetic algorithm (GA) is a method for solving both constrained and unconstrained optimization problems that are based on natural selection, the process that drives biological evolution. The GA repeatedly modifies a population of individual solutions. At each step, the GA selects individuals at random from the current population to be parents and uses them to produce the children for the next generation. Over successive generations, the population "evolves" toward an optimal solution. One can apply the GA to solve a variety of optimization problems that are not well suited for standard optimization algorithms, including problems in which the objective function is discontinuous, nondifferentiable, stochastic or highly nonlinear.

The GA and direct search toolbox are a collection of functions that extend the capabilities of the optimization toolbox and the MATLAB numeric computing environment. The GA and direct search toolbox include routines for solving optimization problems using

1. genetic algorithm;
2. direct search.

These algorithms enable you to solve a variety of optimization problems that lie outside the scope of the Optimization Toolbox.

The GA uses three main types of rules at each step to create the next generation from the current population:

1. *Selection rules* select the individuals, called *parents*, that contribute to the population at the next generation.
2. *Crossover rules* combine two parents to form children for the next generation.

3. *Mutation rules* apply random changes to individual parents to form children.

The GA at the command line calls the GA function ga with the syntax

$$[x\ \text{fval}] = \text{ga(@fitnessfun, nvars, options)}$$

where @fitnessfun is a handle to the fitness function; nvars is the number of independent variables for the fitness function; options is a structure containing options for the GA. If you do not pass in this argument, ga uses its default options.
The results are given by

x: Point at which the final value is attained.

fval: Final value of the fitness function.

The GA tool is a GUI that enables one to use the GA without working at the command line. To open the GA tool, enter

Gatool

at the MATLAB command prompt.

The Optimization Toolbox extends the MATLAB technical computing environment with tools and widely used algorithms for standard and large-scale optimization. These algorithms solve constrained and unconstrained continuous and discrete problems. The toolbox includes functions for linear programming, quadratic programming, nonlinear optimization, nonlinear least squares, nonlinear equations, multi-objective optimization and binary integer programming.

19.6.1 MATLAB Genetic Algorithm Commands

The various commands used in GA MATLAB toolbox are as follows.

bin2int:	BINary string to INTeger string conversion.
bin2real:	BINary string to REAL vector conversion.
bindecod:	BINary DECODing to binary, integer or real numbers.
Compdiv:	COMPute DIVerse things of GEA Toolbox.
compdiv2:	COMPute DIVerse things of GEA Toolbox.
compete:	COMPETition between subpopulations.
Comploc:	COMPute LOCal model things of toolbox.
Compplot:	COMPute PLOT things of GEA toolbox.
geamain2:	MAIN function for Genetic and Evolutionary Algorithm toolbox for matlab.
Initbp:	CReaTe an initial Binary Population.
Initip:	CReaTe an initial (Integer value) Population.
Initpop:	INITialization of POPulation (including innoculation).
Initpp:	Create an INITial Permutation Population.
Initrp:	INITialize a Real value Population.
Migrate:	MIGRATion of individuals between subpopulations.
Mutate:	high level MUTATion function.
Mutbin:	MUTation for BINary representation.
Mutbmd:	real value Mutation like Discrete Breeder genetic algorithm.
mutcomb:	MUTation for combinatorial problems.
mutes1:	MUTation by Evolutionary Strategies 1, derandomized self adaption.
mutes2:	MUTation by Evolutionary Strategies 2, derandomized self adaption.

mutexch:	MUTation by eXCHange.
mutint:	MUTation for INTeger representation.
Mutinvert:	MUTation by INVERTing variables.
mutmove:	MUTation by MOVEing variables.
Mutrand:	MUTation RANDom.
Mutrandbin:	MUTation RANDom of binary variables.
Mutrandint:	MUTation RANDom of integer variables.
Mutrandperm:	MUTation RANDom of binary variables.
mutrandreal:	MUTation RANDom of real variables.
Mutreal:	real value Mutation like Discrete Breeder genetic algorithm.
Mutswap:	MUTation by SWAPping variables.
mutswaptyp:	MUTation by SWAPping variables of identical type.
rankgoal:	perform goal prefereuce calculation between multiple objective values.
Ranking:	RANK-based fitness assignment, single and multi objective, linear and nonlinear.
rankplt:	RANK two multi objective values Partially Less Than.
rankshare:	SHARing between individuals.
recdis:	RECombination DIScrete.
recdp:	RECombination Double Point.
recdprs:	RECombination Double Point with Reduced Surrogate.
recgp:	RECombination Generalized Position.
recint:	RECombination extended INTermediate.
reclin:	RECombination extended LINe.
reclinex:	EXtended LINe RECombination.
recmp:	RECombination Multi-Point, low level function.
recombin:	high level RECOMBINatiou function.
recpm:	RECombination Partial Matching.
recsh:	RECombination Shuffle.
recshrs:	RECombination SHuffle with Reduced Surrogate.
recsp:	RECombination Single Point.
recsprs:	RECombination Single Point with Reduced Surrogate.
reins:	high-level RE-INSertion function.
reinsloc:	RE-INSertion of offspring in population replacing parents LOCal.
reinsreg:	REINSertion of offspring in REGional population model replacing parents.
selection:	high level SELECTion function.
sellocal:	SELection in a LOCAL neighborhood.
selrws:	SELection by Rouletre Wheel Selection.
selsus	SELection by Stochastic Universal Sampling.
seltour:	SELection by TOURnament.
seltrunc:	SELection by TRUNCation.
tbx3bin:	ToolBoX function to define parameters for optimization of binary variables.
tbx3comp:	ToolBoX function to define parameters for COMPeting subpopulation.
tbx3es1:	ToolBoX function to define parameters for local oriented optimization of real variables.
tbx3guifun1:	ToolBoX function to define parameters for optimization, test of gui.
tbx3int:	ToolBoX function to define parameters for optimization of integer variables.
tbx3perm:	ToolBoX function to define parameters for optimization of permutation variables.

tbx3plot1: ToolBoX function to define parameters for grafical display options.
tbx3real: ToolBoX function to define parameters for optimization of real variables.
terminat: TERMINATion function.

Objective functions:

initdopi: INITialzation function for DOuble Integrator objdopi.
initfun1: INITialzation function for de jong's FUNction 1.
mopfonseca1: MultiObjective Problem: FONSECA's function 1.
mopfonseca2: MultiObjective Problem: FONSECA's function 1.
moptest: MultiObjective function TESTing.
obj4wings: OBJective function FOUR-WINGS.
objbran: OBJective function for BRANin rcos function.
objdopi: OBJective function for DOuble Integrator.
objeaso: OBJective function for EASom function.
objfletwell: OBJective function after FLETcher and PoWELL.
objfractal: OBJective function Fractal Mandelbrot.
objfun1: OBJective function for de jong's FUNction 1.
objfun10: OBJective function for ackley's path FUNction 10.
objfun11: OBJective function for langermann's FUNction 11.
objfun12: OBJective function for michalewicz's FUNction 12.
objfun1a: OBJective function for axis parallel hyper-ellipsoid.
objfun1b: OBJective function for rotated hyper-ellipsoid.
objfun1c: OBJective function for moved axis parallel hyper ellipsoid 1c.
objfun2: OBJective function for rosenbrock's FUNction.
objfun6: OBJective function for rastrigins FUNction 6.
objfun7: OBJective function for schwefel's FUNction.
objfun8: OBJective function for griewangk's FUNction.
objfun9: OBJective function for sum of different power FUNction 9.
objgold: OBJective function for GOLDstein-price function.
objharv: OBJective function for HARVest problem.
objint1: OBJective function for INT function 1.
objint2: OBJective function for INT function 2.
objint3: OBJective function for INT function 3.
objint4: OBJective function for INT function 4.
objlinq: OBJective function for discrete LINear Quadratic problem.
objlinq2: OBJective function for LINear Quadratic problem 2.
objone1: OBJective function for ONEmax function 1.
objpush: OBJective function for PUSH-cart problem.
objridge: OBJective function RIDGE.
objsixh: OBJective function for SIX Hump camelback function
objsoland: OBJective function for SOLAND function.
objtsp1: OBJective function for the traveling salesman example.
objtsplib: OBJective function for the traveling salesman library.
plotdopi: PLOTing of DO(Ppel)uble Integration results.
plottsplib: PLOTing of results of TSP optimization (TSPLIB examples).
simdopi1: M-file description of the SIMULINK system named SIMDOPI1.
simdopiv: SIMulation Modell of DOPpelIntegrator, s-function, Vectorized.

simlinq1: M-file description of the SIMULINK system named SIMLINQ1.
simlinq2: Modell of Linear Quadratic Problem, s-function.
tsp_readlib: TSP utility function, reads TSPLIB data files.
tsp_uscity: TSP utility function, reads US City definitions.

Plot functions:

Fitdistc: FITness DISTance Correlation computation.
meshvar: create grafics of objective functions with plotmesh.
plotmesh: PLOT of objective functions as MESH Plot.
plotmop: PLOT properties of MultiObjective functions.
reslook: LOOK at saved RESults.
resplot: RESult PLOTing of GEA Toolbox optimization.
samdata: sammon mapping: data examples.
samgrad: Sammon mapping gradient calculation.
sammon: Multidimensional scaling (SAMMON mapping).
samobj: Sammon mapping objective function.
samplot: Plot function for Multidimensional scaling (SAMMON mapping).

19.6.2 Genetic Algorithm Graphical User Interface

The GA tool is a GUI that enables you to use the GA without working at the command line. To open the GA tool, enter

```
gatool
```

at the MATLAB command prompt. This opens the tool as shown in Figure 19-18. To use the GA tool, you must first enter the following information.

1. *Fitness function*: The objective function you want to minimize. Enter the fitness function in the form @fitnessfun, where fitnessfun.m which is an M-file that computes the fitness function.
2. *Number of variables*: The number of variables in the given fitness function should be given.

The *plot* options

1. best fitness;
2. best individual;
3. distance;
4. expectation;
5. genealogy;
6. range;
7. score diversity;
8. scores;
9. selection;
10. stopping.

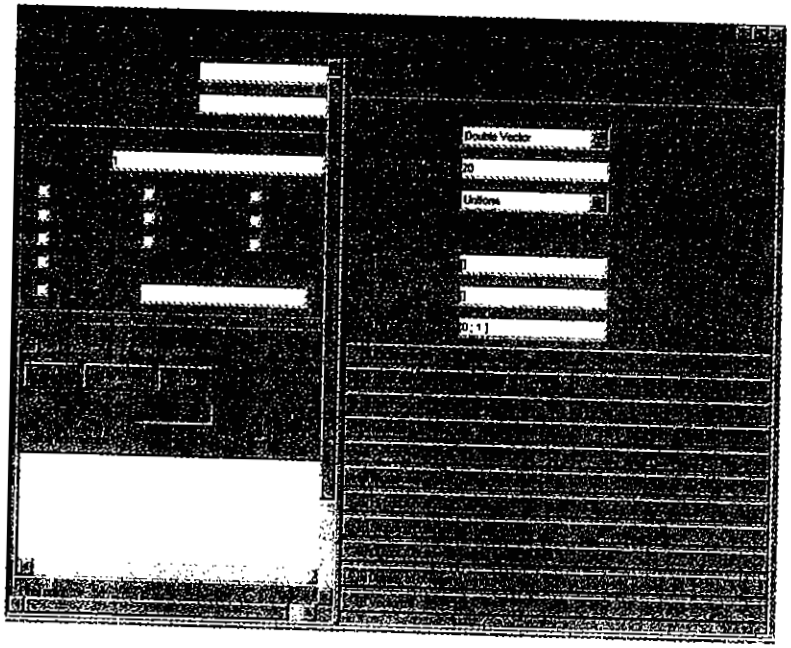


Figure 19-18 Genetic algorithm tool.

On the basis of the problem, custom function may also be built. The various parameters essential for running GA tool should be specified appropriately. The parameters appear on the right-hand side of the GA tool. The description is as follows:

1. *Population*: In this case population type, population size and creation function may be selected. The initial population and initial score may be specified, if not, the "GA tool" creates them. The initial range should be given.
2. *Fitness scaling*: It should be any of the following:
   - rank;
   - proportional;
   - top;
   - shift linear;
   - custom.
3. *Selection*: The selection is made on any one of the methods shown in Figure 19-19.
4. *Reproduction*: In reproduction the elite count and crossover fraction should be given. If elite count is not specified, it is taken as 2 (Figure 19-20).
5. *Mutation*: Generally Gaussian or uniform mutation is carried out. The user may define own customized mutation operation (Figure 19-21).

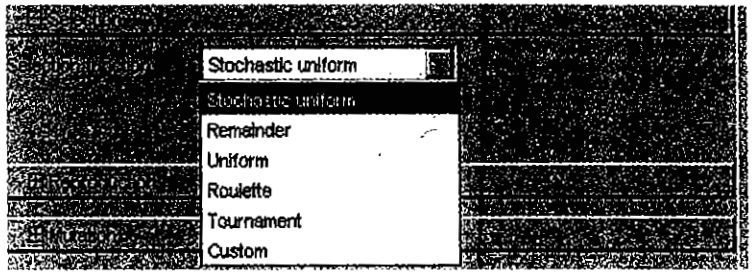


Figure 19-19 Selection.

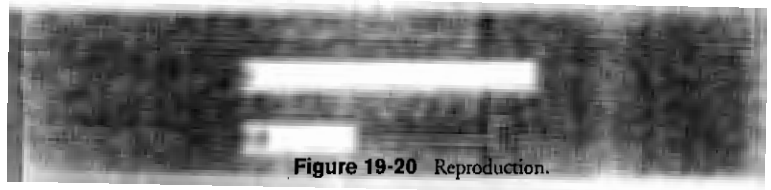


Figure 19-20 Reproduction.

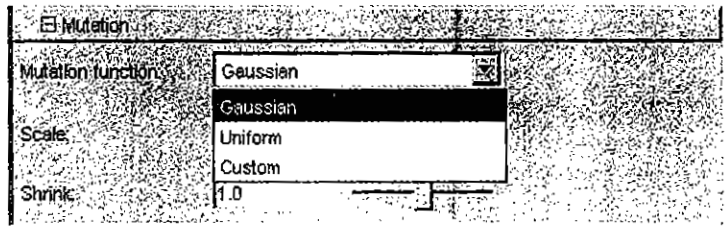


Figure 19-21 Mutation.

6. *Crossover*: The various crossover techniques are shown in Figure 19-22.
7. *Migration*: The parameter for migration should be defined as in Figure 19-23.
8. *Hybrid function*: Any one of the hybrid functions shown in Figure 19-24 may be selected.
9. *Stopping criteria*: The stopping criteria play a major role in simulation. They are shown in Figure 19-25.

The other parameters **Output function, Display to command window** and **Vectorize** may be suitably defined by the user.

Running and Simulation

The menu shown in Figure 19-26 helps the user for running the GA tool.

The running process may be temporarily stopped using "Pause" option and permanently stopped using "Stop" option. The "current generation" will be displayed during the iteration. Once the iterations are completed, the status and results will be displayed. Also the "final point" for the fitness function will be displayed.

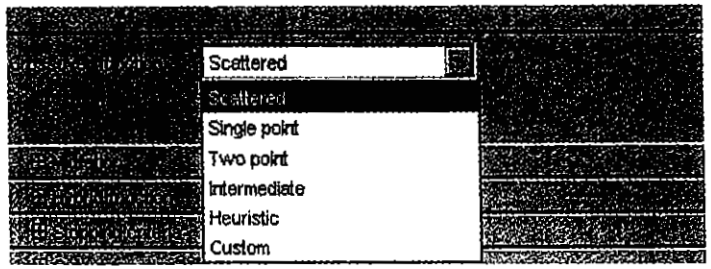


Figure 19-22 Crossover.

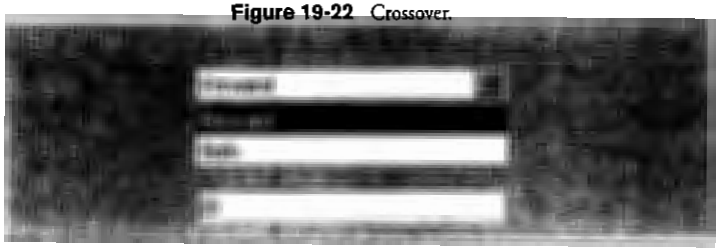


Figure 19-23 Migration.

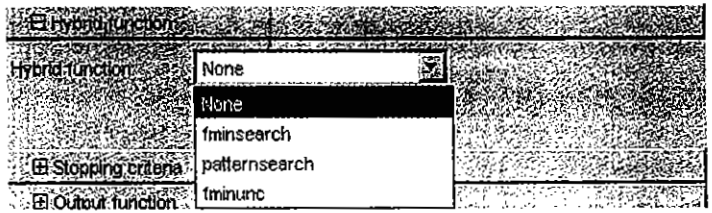


Figure 19-24 Hybrid function.

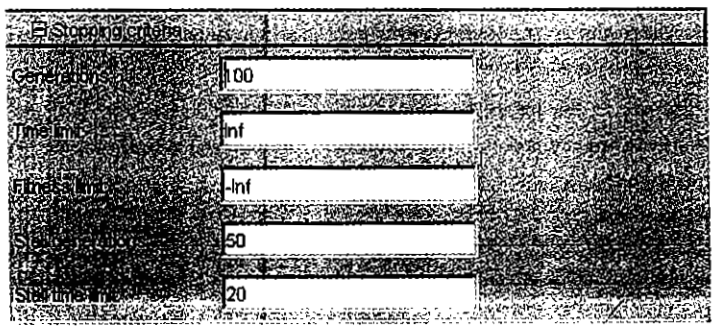


Figure 19-25 Stopping criteria.

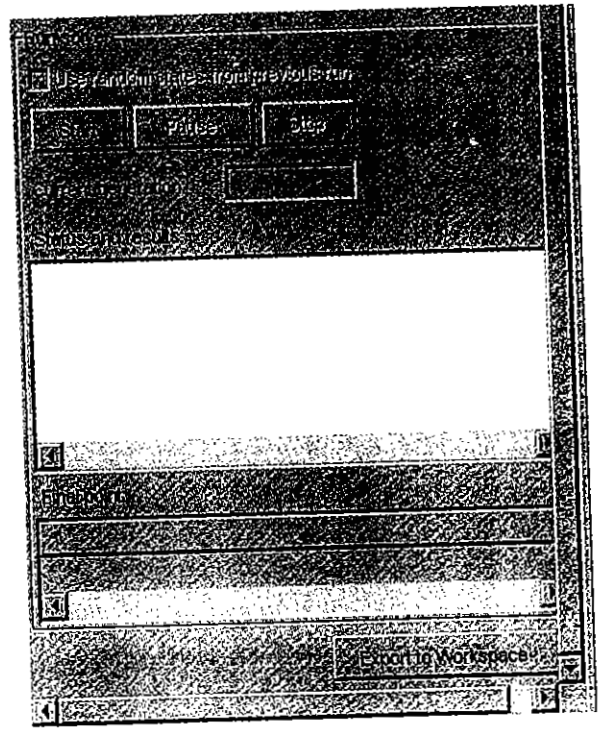


Figure 19-26 Run solver.

19.7 Neural Network MATLAB Source Codes

1. Write a program to implement AND function using ADALINE with bipolar inputs and outputs

Source Code

```
clear all;
clc;
disp('adaline network for and function bipolar inputs, bipolar
      targets');
  x1=[1 1 -1 -1]; %input pattern
  x2=[1 -1 1 -1]; %input pattern
  x3=[1 1 1 1]; %x3 for bias
  t=[1 -1 -1 -1]; %target
  w1=0.1;
  w2=0.1;
  b=0.1;
  alpha=0.1;
  e=2;
  delw1=0;
  delw2=0;
```

```
delb=0;
epoch=0;
while(e>1.018)
  epoch=epoch+1
  e=0;
  for i=1:4
    nety(i)=w1*x1(i)+w2*x2(i)+b;
    nt=[nety(i) t(i)]; %netinput,target
    delw1=alpha*(t(i)-nety(i))*x1(i);
    delw2=alpha*(t(i)-nety(i))*x2(i);
    delb=alpha*(t(i)-nety(i))*x3(i);
    wc=[delw1 delw2 delb]; %weight chanches
    w1=w1+delw1; %updating of weights
    w2=w2+delw2;
    b=b+delb;
    w=[w1 w2 b]; %weights
    x=[x1(i) x2(i) x3(i)]; %input pattern
    pr=[x nt wc w] %to print the result
  end
  for i=1:4
    nety(i)=w1*x1(i)+w2*x2(i)+b;
    e=e+(t(i)-nety(i)) 2;
  end
end
```

2. Write a program to implement AND function using MADALINE with bipolar inputs and outputs.

Source Code

```
clear all;
clc;
disp('madaline network for and function bipolar inputs, bipolar
     targets');
x1=[1 1 -1 -1]; %input pattern
x2=[1 -1 1 -1]; %input pattern
x3=[1 1 1 1]; %x3 for bias
t=[1 -1 -1 -1]; %target
w11=0.1;
w12=0.1;
w21=0.1;
w22=0.1;
b1=0.1;
b2=0.1;
b3=0.5;
v1=0.5;
v2=0.5;
alpha=0.5;
e=2;
delw11=0;
delw12=0;
delw21=0;
delw22=0;
```



```
delb1=0;
delb2=0;
delb3=0;
delv1=0;
delv2=0;
epoch=0;
while (e>1.00)
  epoch=epoch+1
  e=0;
  for i=1:4
    zin1=x1(i)*w11+x2(i)*w21+b1;
    zin2=x1(i)*w12+x2(i)*w22+b2;
    z=[zin1 zin2];
    if (zin1>=0)
      z1=1;
    else
      z1=-1;
    end
    if (zin2>=0)
      z2=1;
    else
      z2=-1;
    end
    hid=[z1 z2];
    nety=b3+z1*v1+z2*v2;

    if (nety>=0)
      y=1;
    else
      y=-1;
    end
    nt=[t(i) nety y];

    if (t(i)==1)
      if (zin1<zin2)
      delb1=alpha*(1-zin1);
      b1=b1+delb1;
      delw11=alpha*(1-zin1)*x1(i);
      w11=w11+delw11;
      delw21=alpha*(1-zin1)*x1(i);
      w21=w21+delw21;
    else
      delb2=alpha*(1-zin2);
      b2=b2+delb2;
      delw12=alpha*(1-zin2)*x2(i);
      w12=w12+delw12;
      delw22=alpha*(1-zin2)*x2(i);
      w22=w22+delw22;
    end
```

```
    elseif (t(i)==-1)
      if (zin1>0)
        delb1=alpha*(-1-zin1);
        b1=b1+delb1;
        delw11=alpha*(-1-zin1)*x1(i);
        w11=w11+delw11;
        delw21=alpha*(-1-zin1)*x1(i);
        w21=w21+delw21;
      else
        delb2=alpha*(-1-zin2);
        b2=b2+delb2;
        delw12=alpha*(-1-zin2)*x2(i);
        w12=w12+delw12;
        delw22=alpha*(-1-zin2)*x2(i);
        w22=w22+delw22;
      end
    end

    del=[delw11 delw21 delb1 delw12 delw22 delb2];
    in=[x1(i) x2(i) x3(i)];
    bi=[v1 v2 b3];

    pr=[in z hid nt del bi]
  end

  for i=1:4
    zin1=b1+x1(i)*w11+x2(i)*w21;
    zin2=b2+x1(i)*w12+x2(i)*w22;
    z=[zin1 zin2];
    if (zin1>=0)
      z1=1;
    else
      z1=-1;
    end
    if (zin2>=0)
      z2=1;
    else
      z2=-1;
    end
    nety=v1*z1+v2*z2+b3;
    e=e+(t(i)-nety) 2;
  end
end
```

3. Write a MATLAB program to construct and test auto associative network for input vector using HEBB rule.

Source Code

```
clear all;
clc;
disp(' AUTO ASSOCIATIVE NETWORK-----HEBB RULE');
```



```
w=[0 0 0 0 ;0 0 0 0 ;0 0 0 0 ;0 0 0 0 ];
s=[1 1 1 -1];
t=[1 1 1 -1];
ip=[1 -1 -1 -1];
disp('INPUT VECTOR');
s
for i=1:4
  for j=1:4
    w(i,j)=w(i,j)+(s(i)*t(j));
  end
end
disp('WEIGHTS TO STORE THE GIVEN VECTOR IS');
w
disp('TESTING THE NET WITH VECTOR');
ip
yin=ip*w;
for i=1:4
  if yin(i)>0
    y(i)=1;
  else
    y(i)=-1;
  end
end
if y==s
  disp('PATTERN IS RECOGNIZED')
else
  disp('PATTERN IS NOT RECOGNIZED')
end
```

Output

```
>> AUTO ASSOCIATIVE NETWORK-----HEBB RULE
INPUT VECTOR
s =
  1 1 1 -1
WEIGHTS TO STORE THE GIVEN VECTOR IS
w =
   1  1  1 -1
   1  1  1 -1
   1  1  1 -1
  -1 -1 -1  1
TESTING THE NET WITH VECTOR
ip =
  1 -1 -1 -1
PATTERN IS NOT RECOGNIZED
```

4. Write a MATLAB program to construct and test auto associative network for input vector using outer product rule.

Source Code

```
clear all;
clc;
```

```
disp('To test Auto associatie network using outer product rule for
      following input vector');
x1=[1 -1 1 -1];
x2=[1 1 -1 -1];
n=0;
w1=x1'*x1;
w2=x2'*x2;
wm=w1+w2;
disp('input');
x1
x2
disp('Target');
x1
x2
disp('Weights');
w1
w2
disp('Weight matrix using Outer Products Rule');
wm
yin=x1*wm;
yin
for i=1:4
  if(yin(i)>0)
    y=1;
  else
    y=-1;
  end
  ny(i)=y;
  if(y==x1(i))
    n=n+1;
  end
end
ny
if(n==4)
  disp('This pattern is recognized');
else
  disp('This pattern is not recognized');
end
n=0;
yin=x2*wm;
yin
for i=1:4
  if(yin(i)>0)
    y=1;
  else
    y=-1;
  end
  ny(i)=y;
  if(y==x2(i))
    n=n+1;
```



```
  end
end
ny
if(n==4)
  disp('This pattern is recognized');
else
  disp('This pattern is not recognized');
end
end
```

Output

```
>> To test Auto associative network using outer product rule for
       following input vector
input
x1 =
  1 -1 1 -1
x2 =
  1 1 -1 -1
Target
x1 =
  1 -1 1 -1
x2 =
  1 1 -1 -1
Weights
w1 =
   1 -1  1 -1
  -1  1 -1  1
   1 -1  1 -1
  -1  1 -1  1
w2 =
   1  1 -1 -1
   1  1 -1 -1
  -1 -1  1  1
  -1 -1  1  1
Weight matrix using Outer Products Rule
wm =
   2  0  0 -2
   0  2 -2  0
   0 -2  2  0
  -2  0  0  2
yin =
   4 -4  4 -4
ny =
  1 -1 1 -1
This pattern is recognized
yin =
  4 4 -4 -4
ny =
  1 1 -1 -1
This pattern is recognized
```

5. Write a MATLAB program to construct and test heteroassociative network for binary inputs and targets.

Source Code

```
%To construct and test Heteroassociative network for binary inputs
        and targets
clear all;
clc;
disp('Heteroassociative Network');
x1=[1 0 0 0];
x2=[1 1 0 0];
x3=[0 0 0 1];
x4=[0 0 1 1];
t1=[1 0];
t2=[1 0];
t3=[0 1];
t4=[0 1];
n=0;
for i=1:4
  for j=1:2
    w(i,j)=((2*x1(i))-1)*((2*t1(j))-1)+((2*x2(i))-1)*((2*t2(j))-1)+
        ((2*x3(i))-1)*((2*t3(j))-1)+((2*x4(i))-1)*((2*t4(j))-1);
  end
end
w
yin1=x1*w
yin2=x2*w
yin3=x3*w
yin4=x4*w
t1=[ 1 -1];
t2=[ 1 -1];
t3=[-1  1];
t4=[-1  1];
for i=1:2
  if(yin1(i)>0)
    y1(i)=1;
  elseif (yin1(i)==0)
    y1(i)=0;
  else
    y1(i)=-1;
  end
end
y1
for i=1:2
  if(y1(i)==t1(i))
    n=n+1;
  end
end
if (n==2)
  disp('The pattern is matched');
else
```



```
  disp('The pattern is not matched');
end
n=0;
for i=1:2
  if(yin2(i)>0)
    y2(i)=1;
  elseif (yin2(i)==0)
    y2(i)=0;
  else
    y2(i)=-1;
  end
end
y2
for i=1:2
  if(y2(i)==t2(i))
    n=n+1;
  end
end
if (n==2)
  disp('The pattern is matched');
else
  disp('The pattern is not matched');
end
n=0;
for i=1:2
  if(yin3(i)>0)
    y3(i)=1;
  elseif (yin3(i)==0)
    y3(i)=0;
  else
    y3(i)=-1;
  end
end
y3
for i=1:2
  if(y3(i)==t3(i))
    n=n+1;
  end
end
if (n==2)
  disp('The pattern is matched');
else
  disp('The pattern is not matched');
end
n=0;
for i=1:2
  if(yin4(i)>0)
    y4(i)=1;
  elseif (yin4(i)==0)
    y4(i)=0;
```

```
  else
    y4(i)=-1;
  end
end
y4
for i=1:2
  if(y4(i)==t4(i))
    n=n+1;
  end
end
if (n==2)
  disp('The pattern is matched');
else
  disp('The pattern is not matched');
end
n-0;
```

6. Write a MATLAB program to implement Discrete Hopfield Network and test the input pattern.

Source Code

```
clear all;
clc;
disp('Discrete Hopfield Network');
theta=0;
x=[1 -1 -1 -1;-1 1 1 -1;-1 -1 -1 1]
%Calculating Weight Matrix
w=x'*x
%calculating Energy
k=1;
while(k<=3)
  temp=0;
  for i=1:4
    for j=1:4
      temp=temp+(x(k,i)*w(i,j)*x(k,j));
    end
  end
  E(k)=(-0.5)*temp;
  k=k+1;
end
%Energy Function for 3 samples
E

%Test for given pattern s=[-1 1 -1 -1]
disp('Given input pattern for testing');
x1=[-1 1 -1 -1]
temp=0;
for i=1:4
  for j=1:4
    temp=temp+(x1(i)*w(i,j)*x1(j));
  end
end
```



```
SE=(-0.5)*temp
disp('By synchronous updation method');
disp('The net input calculated is');
yin=x1*w
for i=1:4
  if(yin(i)>theta)
    y(i)=1;
  elseif(yin(i)==theta)
    y(i)=yin(i);
  else
    y(i)=-1;
  end
end
disp('The output calculated from net input is');
y
temp=0;
for i=1:4
  for j=1:4
    temp=temp+(y(i)*w(i,j)*y(j));
  end
end
SE=(-0.5)*temp
n=0;
for i=1:3
  if (SE==E(i))
    n=0;
    k=i;
  else
    n=n+1;
  end
end

if(n==3)
  disp('Pattern is not associated with any input pattern');
else
  disp('The test pattern');
  x1
  disp('is associated with');
  x(k,:)
end
```

Output

```
>> Discrete Hopfield Network
x =
     1    -1    -1    -1
    -1     1     1    -1
    -1    -1    -1     1
```

```
w =
    3 -1 -1 -1
   -1  3  3 -1
   -1  3  3 -1
   -1 -1 -1  3
E = -10 -12 -10
Given input pattern for testing
x1 = -1 1 -1 -1
SE = -2
By synchronous updation method
The net input calculated is
yin = -2 2 2 -2
The output calculated from net input is
y = -1 1 1 -1
SE = -12
The test pattern
x1 = -1 1 -1 -1
is associated with
ans = -1 1 1 -1
```

7. Write a program to implement Kohonen self-organizing feature maps for given input pattern using learning rate as 0.6.

Source Code

```
clear all;
clc;
disp('Kohonen self organizing feature maps');
disp('The input patterns are');
x=[1 1 0 0; 0 0 0 1; 1 0 0 0 ; 0 0 1 1]
t=1;
alpha(t)=0.6;
e=1;
disp('Since we have 4 input pattern and cluster unit to be formed
    is 2, the weight matrix is');
w=[0.2 0.8; 0.6 0.4; 0.5 0.7; 0.9 0.3]
disp('The learning rate of this epoch is');
alpha
while(e<=3)
  i=1;
  j=1;
  k=1;
  m=1;
  disp('Epoch =');
  e
  while(i<=4)
    for j=1:2
      temp=0;
      for k=1:4
        temp= temp + ((w(k,j)-x(i,k)) 2);
      end
```



```
      D(j)=temp
    end
    if(D(1)<D(2))
      J=1;
    else
      J=2;
    end
    disp('The winning unit is ');
    J
    disp('Weight updation');
    for m=1:4
      w(m,J)=w(m,J) + (alpha(e) * (x(i,m)-w(m,J)));
    end
    w
    i=i+1;
  end
  temp=alpha(e);
  e=e+1;
  alpha(e)=(0.5*temp);
  %disp('First Epoch completed');
  %disp('Learning rate updated for second epoch');
  alpha(e)
end
```

8. Write a MATLAB program to implement full counter propagation network for a given input pattern.

Source Code

```
clear all;
clc;
disp('FULL COUNTERPROPAGATION NETWORK');
x=[1 0 0 0];
y=[1 0];
alpha=0.4;
beta=0.3;
a=0.2;
b=0.1;
e=1;
v=[0.8 0.2; 0.8 0.2; 0.2 0.8; 0.2 0.8];
w=[0.5 0.5; 0.5 0.5];
t=[0.6 0.4 0.4 0.6];
u=[0.7 0.7];

while(e<=3)
  m=1;
  n=1;
  for j=1:2
    temp=0;
    for k=1:4
      temp= temp + ((v(k,j)-x(k))^ 2);
    end
```

```
    for k=1:2
      temp= temp + ((w(k,j)-y(k))^ 2);
    end
    D(j)=temp
  end
  if(D(1)<D(2))
    J=1;
  else
    J=2;
  end
  disp('The winning unit is ');
  J
  disp('Weight updation');
  for m=1:4
    v(m,J)=v(m,J) + (alpha(e) * (x(m)-v(m,J)));
  end
  v
  for n=1:2
    w(n,J)=w(n,J) + (beta(e) * (y(n)-w(n,J)));
  end
  w

  temalpha=alpha(e);
  tembeta=beta(e);
  tema=a(e);
  temb=b(e);
  oe=e;
  te(e)=e;
  e=e+1;
  te(e)=e;
  te1(oe)=oe;
  alpha(e)=(0.5*temalpha);
  alpha
  beta(e)=(0.5*tembeta);
  beta

  disp('for Weight updation from cluster unit to output unit');
  for m=1:4
    v(m,J)=v(m,J) + (alpha(e) * (x(m)-v(m,J)));
  end
  v
  for n=1:2
    w(n,J)=w(n,J) + (beta(e) * (y(n)-w(n,J)));
  end
  w
  for m=1:4
    t(m)=t(m) + (b(oe) * (x(m)-t(m)));
  end
  t
  for n=1:2
```



```
    u(n)=u(n) + (a(oe) * (y(n)-u(n)));
  end
  u

  a(e)=(0.5*tema);
  b(e)=(0.5*temb);
  a
  b
end
te1(e)=e;
x1=te;
x2=te1;
y1=alpha;
y2=beta;
y3=a;
y4=b;
figure(1)
h=plot(x1,y1,x1,y2,x2,y3,x2,y4)
set(h,{'Color'},{'r';'g';'b';'m'})
grid on
xlabel('EPOCH')
ylabel('ERROR RATE')
title('COUNTERPROPAGATION NETWORK')
legend(h,'alpha','beta','a','b')
```

The error rate versus epoch for full counter propagation network is shown in Figure 19-27.

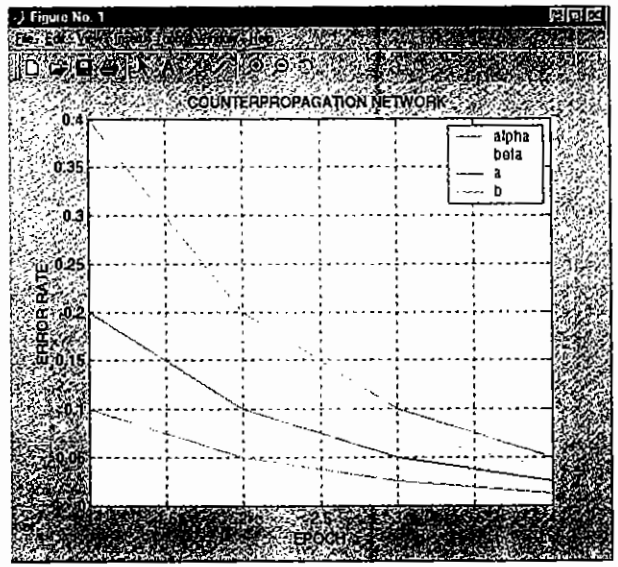


Figure 19-27 Epoch vs error rate for full counter propagation network.

9. Implement a back propagation network for a given input pattern by a suitable MATLAB program. Perform 3 epochs of operation.

Source Code

```
%back propagation network
clear all;
clc;
disp('Back propagation Network');
v=[0.7 -0.4;-0.2 0.3]
x=[0 1]
t=[1]
w=[0.5;0.1]
t1=0;
wb=-0.3
vb=[0.4 0.6]
alpha=0.25
e=1;
temp=0;
while (e<=3)
  e
  for i=1:2
    for j=1:2
      temp=temp+(v(j,i)*x(j));
    end
    zin(i)=temp+vb(i);
    temp1=e (-zin(i));
    fz(i)=(1/(1+temp1));
    z(i)=fz(i);
    fdz(i)=fz(i)*(1-fz(i));
    temp=0;
 end

 for k=1
   for j=1:2
     temp=temp+z(j)*w(j,k);
   end
   yin(k)=temp+wb(k);
   fy(k)=(1/(1+(e -yin(k))));
   y(k)=fy(k);
   temp=0;
end

for k=1
  fdy(k)=fy(k)*(1-fy(k));
  delk(k)=(t(k)-y(k))*fdy(k);
end
for k=1
  for j=1:2
    dw(j,k)=alpha*delk(k)*z(j);
  end
```



```
    dwb(k)=alpha*delk(k);
  end

  for j=1:2
    for k=1
      delin(j)=delk(k)*w(j,k);
    end
    delj(j)=delin(j)*fdz(j);
  end

  for i=1:2
    for j=1:2
      dv(i,j)=alpha*delj(j)*x(i);
    end
    dvb(i)=alpha*delj(i);
  end

  for k=1
    for j=1:2
      w(j,k)= w(j,k)+dw(j,k);
    end
    wb(k)=wb(k)+dwb(k);
  end
  w,wb

  for i=1:2
    for j=1:2
      v(i,j)=v(i,j)+dv(i,j);
    end
    vb(i)=vb(i)+dvb(i);
  end
  v,vb
  te(e)=e;
  e=e+1;
end
```

10. Write a program to implement ART 1 network for clustering input vectors with vigilance parameter.

Source Code

```
clear all;
clc;
disp('Adative Resonance Theory Network 1');
L=2;
m=3;
n=4;
rho=0.4;
te=L/(L-1+n);
te=te/2;
b=[te te te;te te te;te te te;te te te]
```

```
t=ones(3,4)
s=[1 1 0 0;0 0 0 1;1 0 0 0;0 0 1 1]
e=1;

while(e<=4)
  temp=0;
  for i=1:4
    temp=temp+s(e,i);
  end
  ns=temp;
  x(e,:)=s(e,:);
  for i=1:3
    temp=0;
    for j=1:4
      temp=temp+(x(e,j)*b(j,i));
    end
    yin(i)=temp;
  end
  j=1;
  if (yin(j)>=yin(j+1)& yin(j)>=yin(j+2))
    J=1;
  elseif (yin(j+1)>=yin(j)&yin(j+1)>=yin(j+2))
    J=2;
  else
    J=3;
  end
  J
  for i=1:4
    x1(i)=x(e,i)*t(J,i);
  end
  x1;
  temp=0;
  for i=1:4
    temp=temp+x1(i);
  end
  nx=temp;
  m=nx/ns;
  if (m<rho)
    yin(J)=-yin(J);
    j=1;
    if (yin(j)>=yin(j+1)&yin(j)>=yin(j+2))
      J=1;
    elseif (yin(j+1)>=yin(j)&yin(j+1)>=yin(j+2))
      J=2;
    else
      J=3;
    end
    J
  end
  for i=1:4
```



```
    temp=0;
    temp=L-1+nx;
    b(i,J)=(L*x1(i))/temp;
  end
  b
  for i=1:4
    t(J,i)=x1(i);
  end
  t
  e=e+1;
end
```

Output

```
>> Adative Resonance Theory Network 1
b =
  0.2000  0.2000  0.2000
  0.2000  0.2000  0.2000
  0.2000  0.2000  0.2000
  0.2000  0.2000  0.2000

t =
  1  1  1  1
  1  1  1  1
  1  1  1  1
s =
  1  1  0  0
  0  0  0  1
  1  0  0  0
  0  0  1  1
J = 1
b =
  0.6667  0.2000  0.2000
  0.6667  0.2000  0.2000
  0  0.2000  0.2000
  0  0.2000  0.2000
t =
  1  1  0  0
  1  1  1  1
  1  1  1  1
J = 2
b =
  0.6667  0  0.2000
  0.6667  0  0.2000
  0  0  0.2000
  0  1.0000  0.2000
t =
  1  1  0  0
  0  0  0  1
  1  1  1  1
```

```
J = 1
b =
  1.0000   0   0.2000
  0   0   0.2000
  0   0   0.2000
  0   1.0000   0.2000
t =
  1   0   0   0
  0   0   0   1
  1   1   1   1
J = 2
b =
  1.0000   0   0.2000
  0   0   0.2000
  0   0   0.2000
  0   1.0000   0.2000
t =
  1   0   0   0
  0   0   0   1
  1   1   1   1
```

11. Implement adaptive resonance theory network 2 for given inputs by a MATLAB program. Perform 2 iterations only.

Source Code

```
clear all;
clc;
disp('Adative Resonance Theory 2');
s=[0.8 0.6]
a=10;
b=10;
c=0.1;
d=0.9;
e=0;
rho=0.9;
theta=0.7;
wb=[7.0 7.0];
wt=[0 0];
alpha=0.6;
it=1;
u=[0.0 0.0]
tem=0;
for i=1:2
  tem=s(i) 2+tem;
end
ns=sqrt(tem);
p=[0 0]
for i=1:2
  x(i)=s(i);
  w(i)=s(i);
```



```
  q(i)=p(i);
end
x
w
q
temp=0;
temp1=0;
for i=1:2
  temp=w(i) 2+temp;
  temp1=p(i) 2+temp1;
end
nw=sqrt(temp);
np=sqrt(temp1);

for i=1:2
  if (x(i)>=theta)
    fx=x(i);
  else
    fx=0;
  end
  if (q(i)>=theta)
    fq=q(i);
  else
    fq=0;
  end
  v(i)=fx+(b*fq);
end
  v

tem=0;
for i=1:2
  tem=tem+v(i) 2;
end
nv=sqrt(tem);

disp('Updating F1 activation again');
for i=1:2
  u(i)=v(i);
  w(i)=s(i)+(a*u(i));
  p(i)=u(i);
end
u
w
p
tem=0;
temp=0;
for i=1:2
  tem=tem+w(i) 2;
  temp=temp+p(i) 2;
end
```

```
nw=sqrt(tem);
np=sqrt(temp);
for i=1:2
  x(i)=w(i);
  q(i)=p(i);
end
x
q
for i=1:2
  if (x(i)>=theta)
    fx=x(i);
  else
    fx=0;
  end
  if (q(i)>=theta)
    fq=q(i);
  else
    fq=0;
  end
  v(i)=fx+(b*fq);
end
v
disp('Computing signal to F2');
for i=1:2
  temp=0;
  temp=temp+wb(i)*p(i);
  y(i)=temp;
end
y
temp=0;
temp1=0;
for i=1:2
  temp=temp+v(i) 2;
  temp1=temp1+u(i) 2;
end
nv=sqrt(temp);
nu=sqrt(temp1);
for i=1:2
  u(i)=v(i);
  p(i)=u(i)+(d*wt(i));
end
u
p
temp=0;
for i=1:2
  temp=temp+p(i) 2;
end
np=sqrt(temp);
for i=1:2
  r(i)=(u(i)+c*p(i))/(e+nu+(c*np));
```



```
end
temp=0;
for i=1:2
  temp=r(i) 2+temp;
end
nr=sqrt(temp);
i=1;
if(y(i)>=y(i+1))
  J=1;
else
  J=2;
end
%Check for RESET
if (nr>=rho)
  for i=1:2
    w(i)=s(i)+a*u(i);
    v(i)=fx+b*fq;
    temp=0;
    tem=0;
    for i=1:2
      temp=temp+w(i) 2;
      tem=tem+p(i) 2;
    end
    nw=sqrt(temp);
    np=sqrt(tem);
    x(i)=w(i)/(e+nw);
    q(i)=p(i)/(e+np);
  end
end
disp('Update weights for 2 iterations');
while (it<=2)
  wt(J)=(alpha*d*u(J))+1+(alpha*d)*wt(J);
  wb(J)=(alpha*d*u(J))+1+(alpha*d)*wb(J);
  wt
  wb
  for i=1:2
    u(i)=v(i);
    p(i)=u(i)+d*wt(i);
    w(i)=s(i)+a*u(i);
    x(i)=w(i);
    q(i)=p(i);
    v(i)=fx+b*fq;
  end
  it=it+1;
end
```

12. A perceptron neural net uses a hard-limit transfer function. Plot this transfer function.

Source Code

```
%Plot of hard limit transfer function
x = -4:0.1:4;
```

```
y = hardlim(x);
plot(x,y)
```

Output

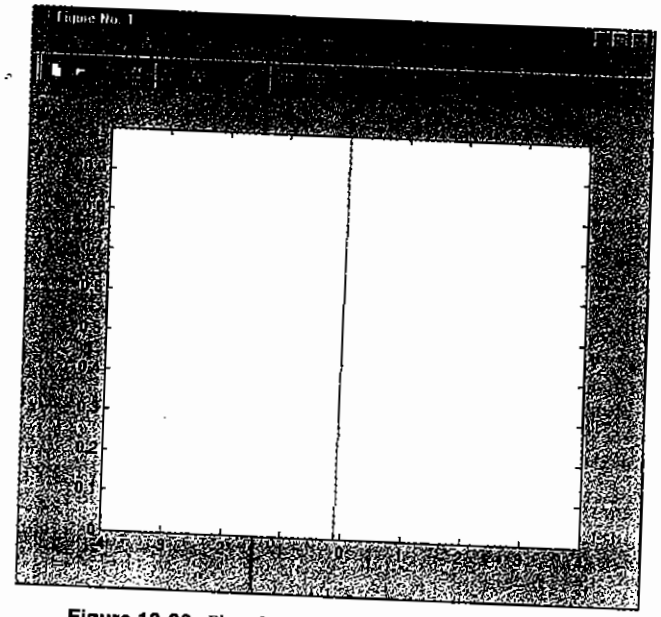


Figure 19-28 Plot of perceptron hard limit transfer function.

13. Create a perceptron network using the command "newp" and obtain its performance.

Source Code

```
%Program to create a perceptron network using command 'newp'
net = newp([ -2 2; -2 2], 1);
%No. of epochs is given as 4
net.trainParam.epochs = 4;
%Let define the input vectors and the target vector
p = [ [2 ;2 ] [1; -2] [-2 ; 2], [-1; 1] ];
t = [0 1 0 1];
%The net can be train with
net = train(net, p, t) ;
%Finally simulate the trained network for each of the inputs.
a = sim(net,p)
```

Output

```
TRAINC, Epoch 0/4
TRAINC, Epoch 3/4
TRAINC, Performance goal met.
```



```
a =
  0  1  0  1
```

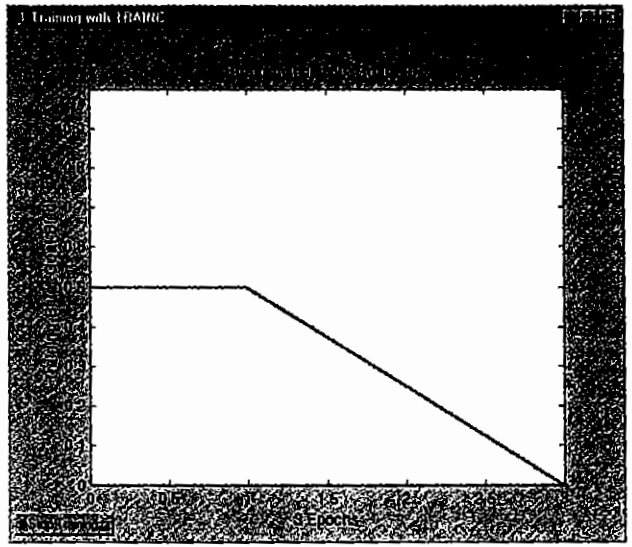


Figure 19-29 Training performance for a perceptron net.

14. Write a MATLAB program to create a feed forward network and perform Batch training

Source Code

```
%Program to create a feed forward network and perfrom batch training
%Create a training set of inputs p and targets t.
%For batch training, all of the input vectors are placed in one matrix.
p = [-1 -1 2 2;0 5 0 5];
t = [-1 -1 1 1];
%Create the feedforward network. The function minmax is used to
%determine the range of the inputs to be used in creating the network.
net=newff(minmax(p),[3,1],{'tansig','purelin'},'traingd');
%Set training parameters.
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-5;
% Now train the network.
[net,tr]=train(net,p,t);
% The training record tr contains information about the progress
    of training.
% Now the trained network can be simulated to obtain its response
    to the inputs in the
% training set.
a = sim(net,p)
```

Output

```
>> TRAINGD, Epoch 0/300, MSE 0.69466/1e-005, Gradient 2.29478/1e-010
TRAINGD, Epoch 50/300, MSE 4.17837e-005/1e-005,
    Gradient 0.00840093/1e-010
TRAINGD, Epoch 68/300, MSE 9.35073e-006/1e-005,
    Gradient 0.0038652/1e-010
TRAINGD, Performance goal met.
a =
  -1.0008 -0.9996 1.0053 0.9971
```

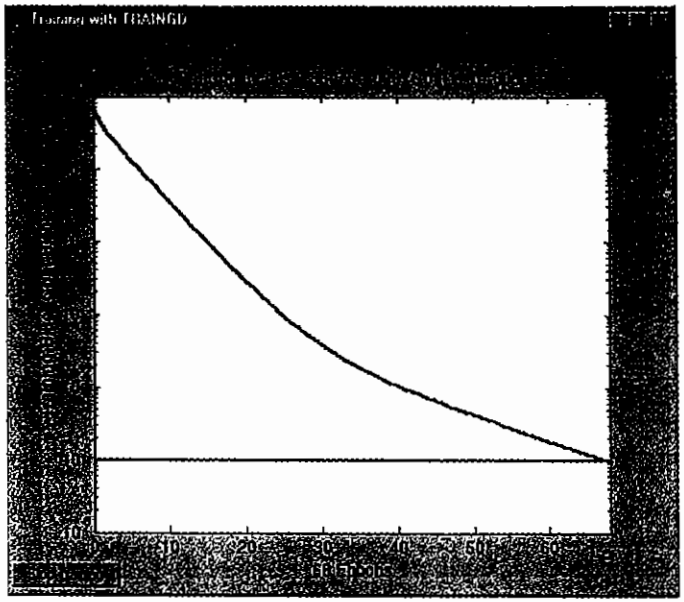


Figure 19-30 Training performance of a feed-forward network.

15. A radial basis network is a network with two layers. It consists of a hidden layer of radial basis neurons and an output layer of linear neurons. Plot a radial basis function.

Source Code

```
%Plot of radial basis function
clear all;
clc;
x = -5:.1:5;
y = radbas(x);
plot(x,y)
```


Output

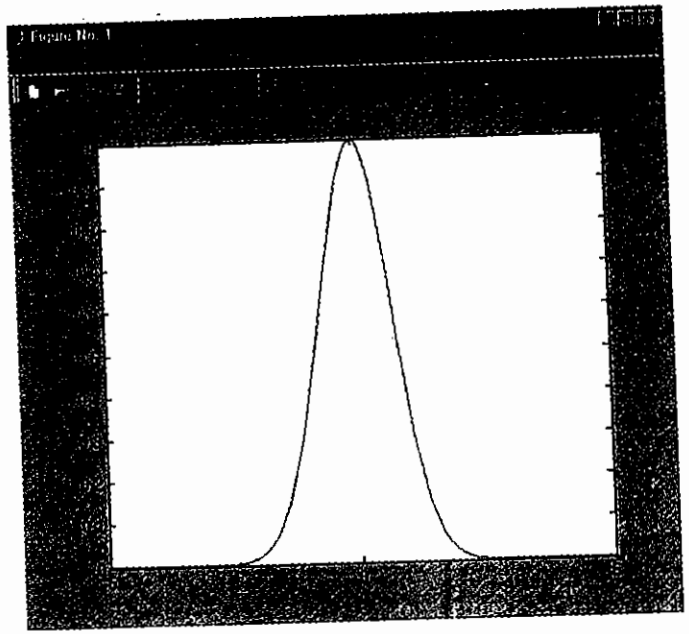


Figure 19-31 Radial basis function.

16. Consider a surface described by $z = \sin(x)\cos(y)$ defined on a square $-3 \le x \le 3, -3 \le y \le 3$.

- Plot the surface z as a function of x and y.
- Design a neural network which will fit the data. You should study different alternatives and test the final result by studying the fitting error.

Source Code

```
%Generate data
x = -3:0.25:3
y = -3:0.25:3
z = sin(x)'*cos(y)
surf(x,y,z)
xlabel('x axis');
ylabel('y axis');
zlabel('z axis');
title('surface z = sin(x)cos(y)');
%Store data in input matrix P and output vector T
P = [x;y];
T = z;
%Set small number of neurons in the first layer, say 25, 25 in
%the output.
%Initialize the network
```

```
net=newff([-3 3; -3 3], [25 25], {'tansig' 'purelin'},'trainlm');
%Apply Levenberg-Marquardt algorithm
%Define parameters
net.trainParam.show = 50;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 300;
net.trainParam.goal = 1e-3;
%Train network
net1 = train(net,P,T);
a= sim(net1,P);
surf(x,y,a)
```

Output

```
TRAINLM, Epoch 0/300, MSE 6.57445/0.001, Gradient 1010.2/1e-010
TRAINLM, Epoch 4/300, MSE 0.000424834/0.001, Gradient 10.0448/1e-010
TRAINLM, Performance goal met.
```

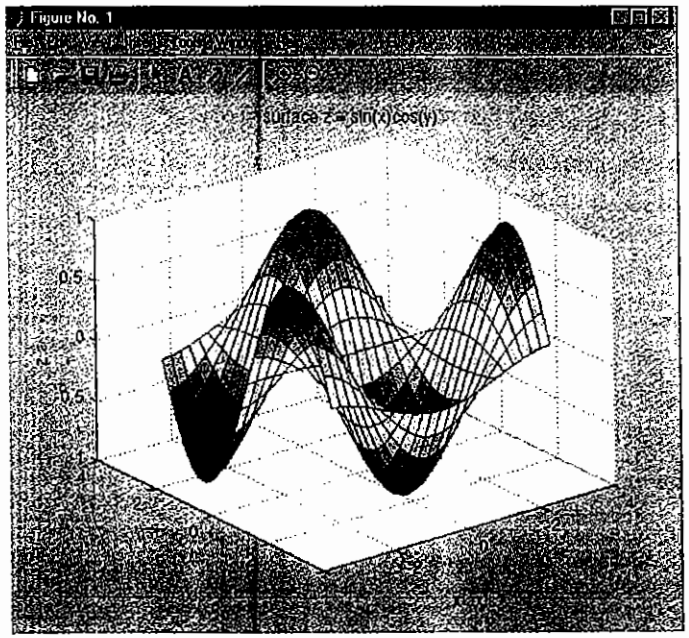


Figure 19-32 Surface for $z = \sin(x)\cos(y)$.

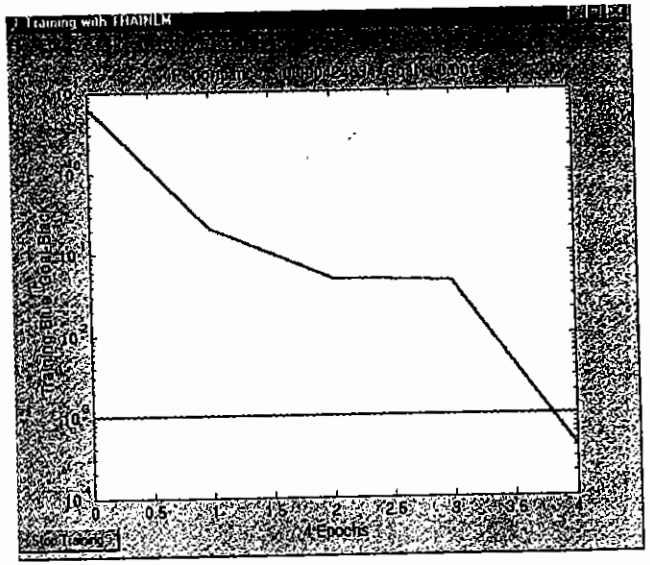


Figure 19-33 Training performance

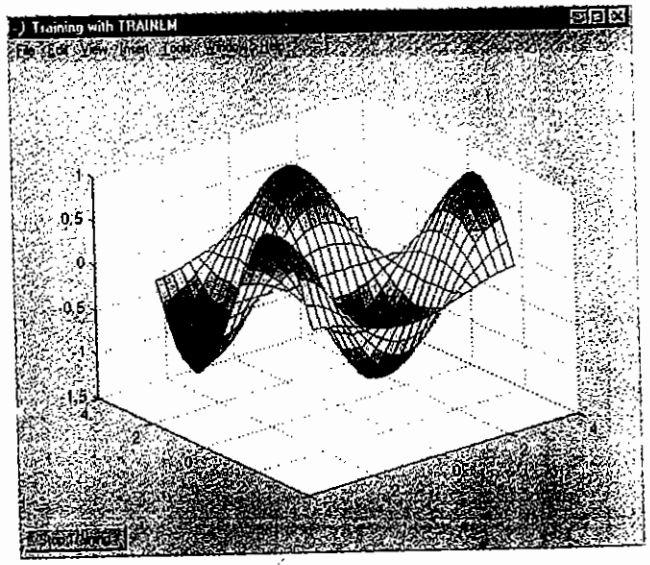


Figure 19-34 Surface of (x, y, a).

17. Find a neural network model, which produces the same behavior as Van der Pol equation:

$$\dot{x} + (x^2 - 1)\dot{x} + x = 0$$

Solution: The given Van der Pol equation can be represented in state space form as

$$\dot{x}_1 = x_2(1 - x_1^2) - x_1$$
$$\dot{x}_2 = x_1$$

Here various initial functions can be used. On applying vector notations, the above state space form is given as

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x})$$

where

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \text{and} \quad \mathbf{f}(\mathbf{x}) = \begin{bmatrix} f_1(x_1, x_2) \\ f_2(x_1, x_2) \end{bmatrix} = \begin{bmatrix} x_2(1 - x_1^2) - x_1 \\ x_1 \end{bmatrix}$$

For building this Van der Pol model, Simulink is used.

Source Code

The simulink model for Van der Pol equation is shown in Figure 19-35.

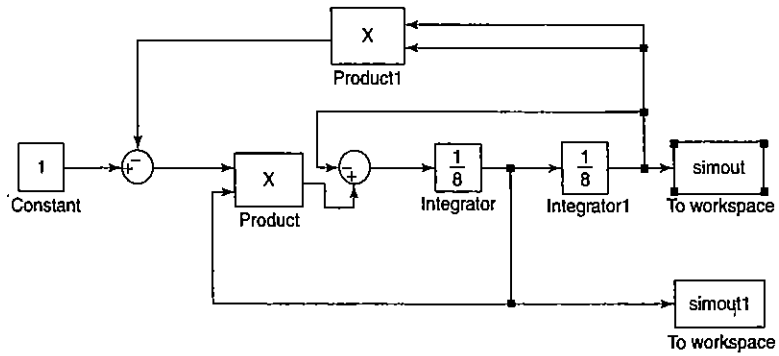


Figure 19-35 Simulink model for Van der Pol equation.

```
% Define the simulation parameters for Van der Pol equation
% The period of simulation: tfinal = 15 seconds;
tfinal = 15;
% Solve Van der Pol differential equation
[t,x]=sim('vandpol',tfinal);
% Plot the states as function of time
plot(t,x)
xlabel('time (secs)');
ylabel('x1 and x2 - states');
title('Van Der Pol Equation');
```



```
grid
%Plot of training vectors
P = t';
T = x';
plot(P,T,'+');
title('Training Vectors');
xlabel('Input Vector P');
ylabel('Target Vector T');
% Define the learning algorithm parameters- a feed forward
    network chosen
net=newff([0 20], [10,2], {'tansig','purelin'},'trainlm');
%Define parameters
net.trainParam.show = 100;
net.trainParam.lr = 0.05;
net.trainParam.epochs = 500;
net.trainParam.goal = 1e-3;
%Train network
net1 = train(net, P, T);
```

Output

```
TRAINLM, Epoch 0/500, MSE 6.81368/0.001, Gradient 408.177/1e-010
TRAINLM, Epoch 100/500, MSE 0.0208639/0.001, Gradient 0.112283/1e-010
TRAINLM, Epoch 200/500, MSE 0.0208277/0.001, Gradient 0.00613187/1e-010
TRAINLM, Epoch 300/500, MSE 0.0208226/0.001, Gradient 0.0603704/1e-010
TRAINLM, Epoch 400/500, MSE 0.0208181/0.001, Gradient 1.62252/1e-010
TRAINLM, Epoch 500/500, MSE 0.0208168/0.001, Gradient 0.0403124/1e-010
TRAINLM, Maximum epoch reached, performance goal was not met.
```

The states of the Van der Pol equation are plotted as function of time as shown in Figure 19-36.

The training vectors are shown in Figure 19-37.

The convergence has not occurred (performance goal not met), since network structure is simple. As a result, by modifying its structure, perform further iterations to achieve the performance goal.

Figure 19-38 shows the training performance.

19.8 Fuzzy Logic MATLAB Source Codes

1. Write a MATLAB program to implement fuzzy set operation and properties

Source Code

```
%Program for fuzzy set with properties and operations
clear all;
clc;
disp('Fuzzy set with properties and operation');
a=[0 1 0.5 0.4 0.6];
b=[0 0.5 0.7 0.8 0.4];
c=[0.3 0.9 0.2 0 1];
phi=[0 0 0 0 0];
disp('Union of a and b');
```

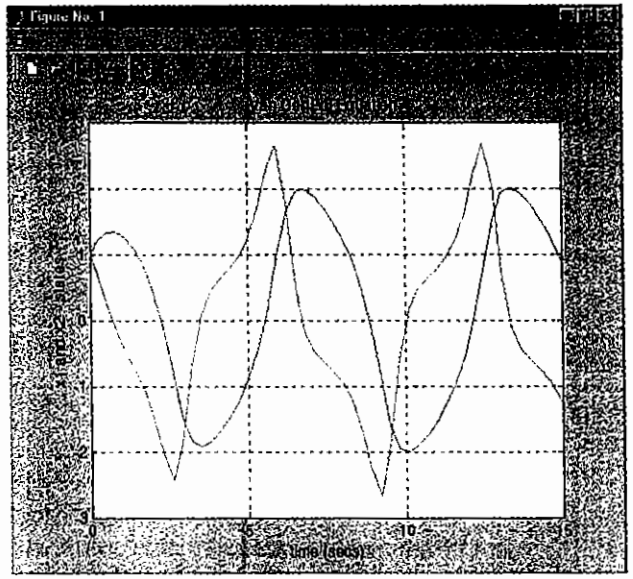


Figure 19-36 Plot of states vs time (Van der Pol equation).

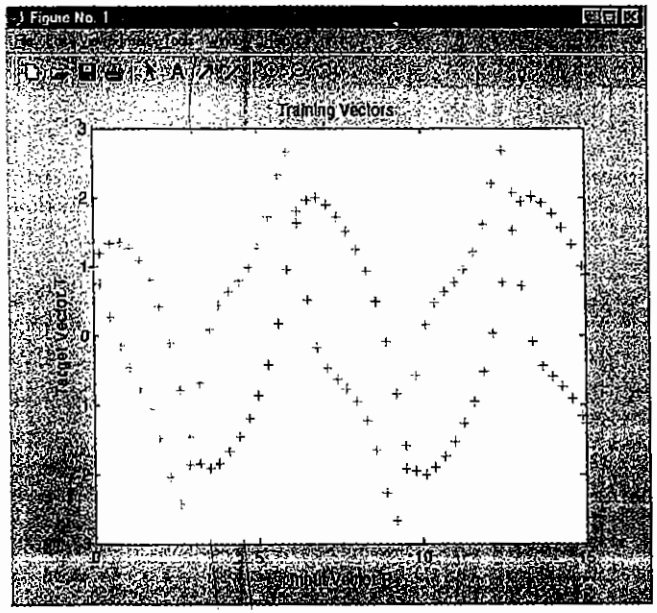


Figure 19-37 Plot of training vectors.

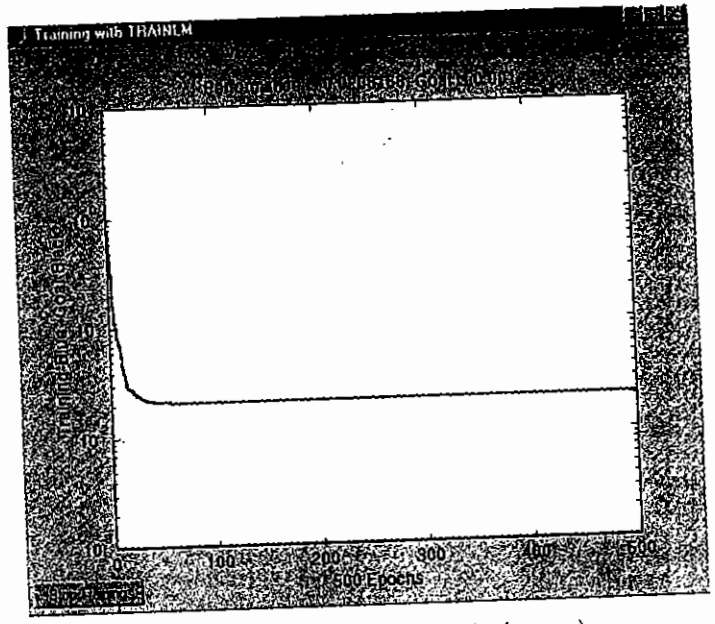


Figure 19-38 Training performance (goal not met).

```
au=max(a,b)
disp('Intersection of a and b');
iab=min(a,b)
disp('Union of b and a');
bu=max(b,a)
if (au==bu)
  disp('Commutative law is satisfied');
else
  disp('Commutative law is not satisfied');
end
disp('Union of b and c');
cu=max(b,c)
disp('a U (b U c)');
acu=max(a,cu)
disp('(a U b) U c)');
auc=max(au,c)
if (acu==auc)
  disp('Associative law is satisifed');
else
  disp('Associative law is not satisfied');
end
disp('intersection of b and c');
ibc=min(b,c)
disp('a U (b I c)');
```

```
dls=max(a,ibc)
disp('Union of a and c');
uac=max(a,c)
disp('(a U b) I (a U c)');
drs=min(au,uac)
if (dls==drs)
  disp('Distributive law is satisfied');
else
  disp('distributive law is not satisfied');
end
disp('a U a');
idl=max(a,a)
a
if (idl==a)
  disp('Idempotency law is satisfied');
else
  disp('Idempotency law is not satsified');
end
disp('a U phi');
idtl=max(a,phi)
a
if (idtl==a)
  disp('Identity law is satisfied');
else
  disp('Identity law is not satisfied');
end
disp('a I phi');
idtl=min(a,phi)
phi
if (idtl==phi)
  disp('Identity law is satisfied');
  else
  disp('Identity law is not satisfied');
end
disp('Complement of (a I b)');
for i=1:5
  ciab(i)=1-iab(i);
end
ciab
disp('Complement of a');
for i=1:5
  ca(i)=1-a(i);
end
ca
disp('Complement of b');
for i=1:5
  cb(i)=1-b(i);
end
cb
disp('a Complement U b Complient');
```



```
dml=max(ca,cb)
if (dml==ciab)
  disp('Demorgans law is satisfied');
else
  disp('Demorgans law is not satisfied');
end
disp('Complement of complement of a');
for i=1:5
  cca(i)=1-ca(i);
end
cca
a
if (a==cca)
  disp('Involution law is satisified');
else
  disp('Involution law is not satisfied');
end
```

2. Write a program to implement composition of Fuzzy and Crisp relations

Source Code

```
%program for composition on Fuzzy and Crisp relations
clear all;
clc;
disp('Composition on Crisp relation');
a=[0.2 0.6]
b=[0.3 0.5]
c=[0.6 0.7]
for i=1:2
  r(i)=a(i)*b(i);
  s(i)=b(i)*c(i);
end
r
s
irs=min(r,s)
disp('Crisp - Composition of r and s using max-min composition');
crs=max(irs)
for i=1:2
  prs(i)=r(i)*s(i);
end
prs
disp('Crisp - Composition of r and s using max-product composition');
mprs=max(prs)
disp('Fuzzy Composition');
firs=min(r,s)
disp('Fuzzy - Composition of r and s using max-min composition');
frs=max(firs)
for i=1:2
  fprs(i)=r(i)*s(i);
end
fprs
```

```
disp('Fuzzy - Composition of r and s using max-product composition');
fmprs=max(fprs)
```

3. Consider the following fuzzy sets

$$A=\left\{\frac{1}{2}+\frac{0.4}{3}+\frac{0.6}{4}+\frac{0.3}{5}\right\}$$

$$B=\left\{\frac{0.3}{2}+\frac{0.2}{3}+\frac{0.6}{4}+\frac{0.5}{5}\right\}$$

Calculate, $A \cup B$, $A \cap B$, $\bar{A}$, $\bar{B}$ by a MATLAB program.

Source Code

```
%Program to find union, intersection and complement of fuzzys sets
% Enter the two Fuzzy sets
u=input('enter the first fuzzy set A');
v=input('enter the second fuzzy set B');
disp('Union of A and B');
w=max(u,v)
disp('Intersection of A and B');
p=min(u,v)
[m]=size(u);
disp('Complement of A');
q1=ones(m)-u
[n]=size(v);
disp('Complement of B');
q2=ones(n)-v
```

Output

```
enter the first fuzzy set A[1 0.4 0.6 0.3]
enter the second fuzzy set B[0.3 0.2 0.6 0.5]
Union of A and B
w =
  1.0000   0.4000   0.6000   0.5000
Intersection of A and B
p =
  0.3000   0.2000   0.6000   0.3000
Complement of A
q1 =
  0 0.6000   0.4000   0.7000
Complement of B
q2 =
  0.7000   0.8000   0.4000   0.5000
```

4. Find whether the following relation is a tolerance relation or not by writing a MATLAB file.

$$R=\begin{bmatrix}1&1&0&0&0\\1&1&0&0&0\\0&0&1&0&0\\0&0&0&1&1\\0&0&0&1&1\end{bmatrix}$$

Source Code

```
% Program to check whether the given relation is tolerance relation
  or not
p=input('enter the relation')
sum=0;
sum1=0;
[m,n]=size(p);
if(m==n)
  for i=1:m
    if(p(1,1)==p(i,i))
    else
      fprintf(' the given relation is irrelexive and ');
      sum1=1;
      break;
    end
  end
  if(sum1 ~= 1)
    fprintf('the given relation is reflexive and ');
  end
  for i=1:m
    for j=1:n
      if(p(i,j)==p(j,i))
      else
        fprintf('not symmetry hence ');
        sum=1;
        break;
      end
    end
    if(sum==1)
      break;
    end
  end
  if(sum~=1)
    fprintf('symmetry hence ');
  end
end
if(sum1~=1)
  if(sum~=1)
    fprintf('the given relation tolerance relation');
  else
    fprintf(' the given relation is not tolerance relation');
  end
else
  fprintf(' the given relation is not tolerance relation');
end
```

Output

```
enter the relation[1 1 0 0 0;1 1 0 0 0;0 0 1 0 0;0 0 0 1 1;0 0 0 1 1]
```

```
p =
   1   1   0   0   0
   1   1   0   0   0
   0   0   1   0   0
   0   0   0   1   1
   0   0   0   1   1
```

The given relation is reflexive and symmetry hence the given relation is a tolerance relation.

5. To find whether the following relation is equivalence or not using a MATLAB program.

$$R = \begin{bmatrix} 1 & 0.87 & 0 & 0.13 & 0.35 \\ 0.87 & 1 & 0.46 & 0 & 0.98 \\ 0 & 0.46 & 1 & 0 & 0 \\ 0.13 & 0 & 0 & 1 & 0.54 \\ 0.24 & 0.98 & 0 & 0.54 & 1 \end{bmatrix}$$

Source Code

```
%Program to check whether the given relation
%is an Equivalence relation or not
p=input('enter the matrix')
sum=0;
sum1=0;
sum2=0;
sum3=0;
[m,n]=size(p);
l=m;
if(m==n)
  for i=1:m
    if(p(1,1)==p(i,i))
    else
      fprintf(' the given relation is irreflexive ');
      sum1=1;
      break;
    end
  end
  if(sum1 ~= 1)
    fprintf(' the given relation is reflexive');
  end
  m;
  n;
  [m,n]=size(p)
  for i=1:m
    for j=1:n
      if(p(i,j)==p(j,i))
      else
        fprintf(' , not symmetry');
        sum=1;
        break;
      end
    end
```



```
    if(sum==1)
      break;
    end
  end
  if(sum~=1)
    fprintf(' ,symmetry');
  end
for i=1:m
  for j=1:n
    for k=1:-1:1
      lambda1=p(i,j);
      lambda2=p(j,k);
      lambda3=p(i,k);
      q=min(lambda1,lambda2);
      if(lambda3 >= q)
      else
        sum2=1;
        break;
      end
    end
  end
end
if(sum2 ~= 1)
  fprintf(' and transitivity hence ');
else
  fprintf(' and not transitivity hence ');
end
if(sum1~=1)
  if(sum~=1)
    if(sum2~=1)
    fprintf(' the given relation is equivalence relation');
  else
    fprintf('the given relation is not equivalence relation');
  end
else
  fprintf('not equivalence relation');
end
else
  fprintf('not equivalence relation');
end
end
```

Output

```
enter the matrix[1 0.87 0 0.13 0.35;0.87 1 0.46 0 0.98; 0 0.46 1 0 0;
                 0.13 0 0 1 0.54;0.24 0.98 0 0.54 1]
p =
   1.0000   0.8700        0   0.1300   0.3500
   0.8700   1.0000   0.4600        0   0.9800
        0   0.4600   1.0000        0        0
   0.1300        0        0   1.0000   0.5400
   0.2400   0.9800        0   0.5400   1.0000
```

The given relation is reflexive, not symmetry and not transitivity and hence not an equivalence relation.

6. Find the fuzzy relation using fuzzy max-min method for the following using MATLAB program:

$$R = \begin{bmatrix} 0.2 & 0.3 & 0.4 \\ 0.3 & 0.5 & 0.7 \\ 1 & 0.8 & 0.6 \end{bmatrix} \quad \text{and} \quad S = \begin{bmatrix} 0.1 & 1 \\ 0.4 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}$$

```
%Program to find a relation using Max-Min Composition
%enter the two vectors whose relation is to be find
R=input('enter the first vector')
S=input('enter the second vector')
% find the size of two vectors
[m,n]=size(R)
[x,y]=size(S)
if(n==x)
for i=1:m
for j=1:y
c=R(i,:)
d=S(:,j)
f=d'
%find the minimum of two vectors
q=min(c,f)
%find the maximum of two vectors
h(i,j)=max(q);
end
end
%print the result
display('the fuzzy relation between two vectors is');
display(h)
else
display('The fuzzy relation cannot be find')
end
```

Output

```
enter the first vector[0.2 0.3 0.4;0.3 0.5 0.7;1 0.8 0.6]
R =
   0.2000   0.3000   0.4000
   0.3000   0.5000   0.7000
   1.0000   0.8000   0.6000
enter the second vector[0.1 1;0.4 0.2;0.3 0.7]
S =
   0.1000   1.0000
   0.4000   0.2000
   0.3000   0.7000
ans =
the fuzzy relation between two vectors is
h =
   0.3000   0.4000
   0.4000   0.7000
   0.4000   1.0000
```


7. Use MATLAB commands to display the triangular and Gaussian membership function. Given $x = 0$ to 10 with increment of 0.1. Triangular membership function is defined between [5 6 7] and Gaussian function is defined between 2 and 4.

Source Code

```
%Program to depict membership functions
x=(0:0.1:10)';
y1=gaussmf(x,[2 4]);
%Plot of Gaussian membership function
plot(x,y1)
hold
%Plot of Triangular membership function
y2=trimf(x,[5 6 7]);
plot(x,y2)
```

Output

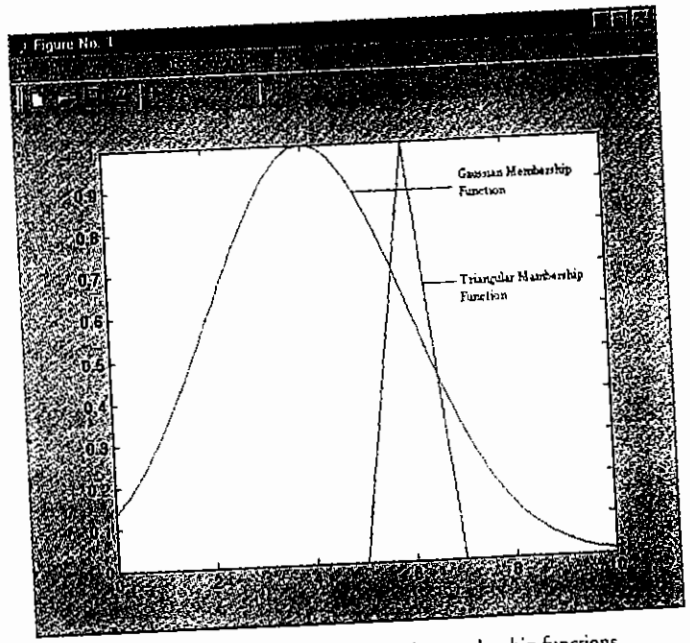


Figure 19-39 Gaussian and triangular membership functions.

8. Find the fuzzy relation between two vectors R and S using max-product method by a MATLAB program.

$$R = \begin{bmatrix} 0.2 & 0.3 & 0.4 \\ 0.3 & 0.5 & 0.7 \\ 1 & 0.8 & 0.6 \end{bmatrix} \quad \text{and} \quad S = \begin{bmatrix} 0.1 & 1 \\ 0.4 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}$$

Source Code

```
%Program to find a relation using Max-Product Composition
%enter the two input vectors
R=input('enter the first vector')
S=input('enter the second vector')
%find the size of the two vector
[m,n]=size(R);
[x,y]=size(S);
if(n==x)
for i=1:m
for j=1:y
c=R(i,:);
d=S(:,j);
[f,g]=size(c);
[h,q]=size(d);
%finding product
for l=1:g
e(1,l)=c(1,l)*d(l,1);
end
%finding maximum
t(i,j)=max(e);
end
end
disp('Max-product composition relation is');
disp(t)
else
display('Cannot find relation using max product composition');
end
```

9. Using MATLAB program find the crisp lambda cut set relations for lambda = 0.6. The fuzzy matrix is given by

$$R = \begin{bmatrix} 0.1 & 0.6 & 0.8 & 1 \\ 1 & 0.7 & 0.4 & 0.2 \\ 0 & 0.6 & 1 & 0.5 \\ 0.1 & 0.5 & 1 & 0.9 \end{bmatrix}$$

Source code

```
%Lambda Cut method of defuzzification
% Enter the given relational matrix
R=input('Enter the relational matrix')
% Enter the lambda value
lambda=input('enter the lambda value')
[m,n]=size(R);
for i=1:m
for j=1:n
if(R(i,j)<lambda)
b(i,j)=0;
```



```
else
b(i,j)=1;
end
end
end
% output value
display('the crisp value is')
display(b)
```

Output

```
Enter the relational matrix[0.1 0.6 0.8 1;1 0.7 0.4 0.2;0 0.6 1 0.5;
                           0.1 0.5 1 0.9]

R =
  0.1000  0.6000  0.8000  1.0000
  1.0000  0.7000  0.4000  0.2000
  0       0.6000  1.0000  0.5000
  0.1000  0.5000  1.0000  0.9000

enter the lambda value 0.6
lambda =
  0.6000
ans =
the crisp value is
b =
  0  1  1  1
  1  1  0  0
  0  1  1  0
  0  0  1  1
```

19.9 Genetic Algorithm MATLAB Source Codes

1. Write a MATLAB program for maximizing $f(x) = x^2$ using GA, where x is ranges from 0 to 31. Perform 5 iterations only.

Steps involved

Step 1: Generate initial four populations of binary string with 5 bits length.

Step 2: Calculate corresponding x and fitness value $f(x) = x^2$.

Step 3: Use the tournament selection method to generate new four populations.

Step 4: Apply crossover operator to the new four populations and generate new populations.

Step 5: Apply mutation operator for each population.

Step 6: Repeat the steps 2–5 for 5 iterations.

Step 7: Finally print the result.

Source Code

```
%program for Genetic algorithm to maximize the function f (x) =xsquare
clear all;
```

```
clc;
%x ranges from 0 to 31 2power5 = 32
%five bits are enough to represent x in binary representation
n=input('Enter no. of population in each iteration');
nit=input('Enter no. of iterations');
%Generate the initial population
[oldchrom]=initbp(n,5)
%The popultion in binary is converted to integer
FieldD=[5;0;31;0;0;1;1]
for i=1:nit
  phen=bindecod(oldchrom,FieldD,3); % phen gives the integer value
    of the binary population
  %obtain fitness value
  sqx=phen.^ 2;
  sumsqx=sum(sqx);
  avsqx=sumsqx/n;
  hsqx=max(sqx);
  pselect=sqx./sumsqx;
  sumpselect=sum(pselect);
  avpselect=sumpselect/n;
  hpselect=max(pselect);
  %apply roulette wheel selection
  FitnV=sqx;
  Nsel=4;
  newchrix=selrws(FitnV, Nsel);
  newchrom=oldchrom(newchrix,:);
  %Perform Crossover
  crossrate=1;
  newchromc=recsp(newchrom,crossrate); %new population after crossover
  %Perform mutation
  vlub=0:31;
  mutrate=0.001;
  newchromm=mutrandbin(newchromc,vlub,0.001); %new population
    after mutation
  disp('For iteration');
  i
  disp('Population');
  oldchrom
  disp('X');
  phen
  disp('f(X)');
  sqx
  oldchrom=newchromm;
end
```

Output

```
Enter no. of population in each iteration4
Enter no. of iterations5
At the end of fifth iteration, the output is
```



```
For iteration
i =
  5
Population
oldchrom =
  0  0  0  0  1
  0  0  0  1  0
  0  1  1  0  0
  0  1  0  0  0
X
phen =
  1
  2
  12
  8
f(X)
sqx =
  1
  4
  144
  64
```

2. Use Gatool and minimize the quadratic equation $f(x) = x^2 + 3x + 2$ within the range $-6 \le x \le 0$.

Function Definition

Define the given function $f(x) = x^2 + 3x + 2$ in a separate m-file as shown in Figure 19-40.

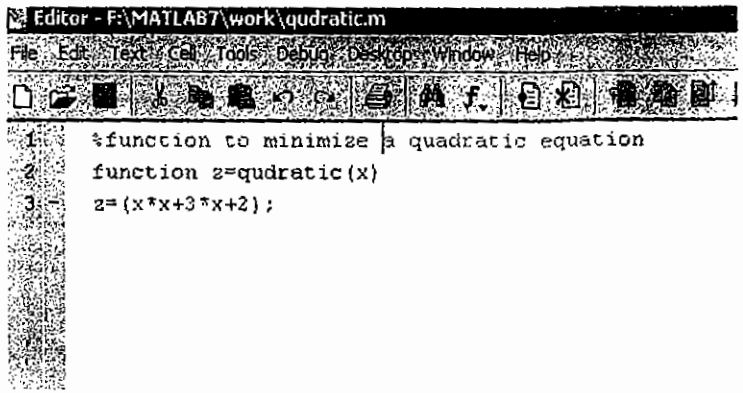


Figure 19-40 M-file showing defined quadratic function.

Creation of Gatool

On typing "gatool" in the command prompt, the GA toolbox opens. In tool, for fitness value type @qudratic and mention the number of variables defined in the function. Select best fitness in plot and specify the other parameters as shown in Figure 19-41.

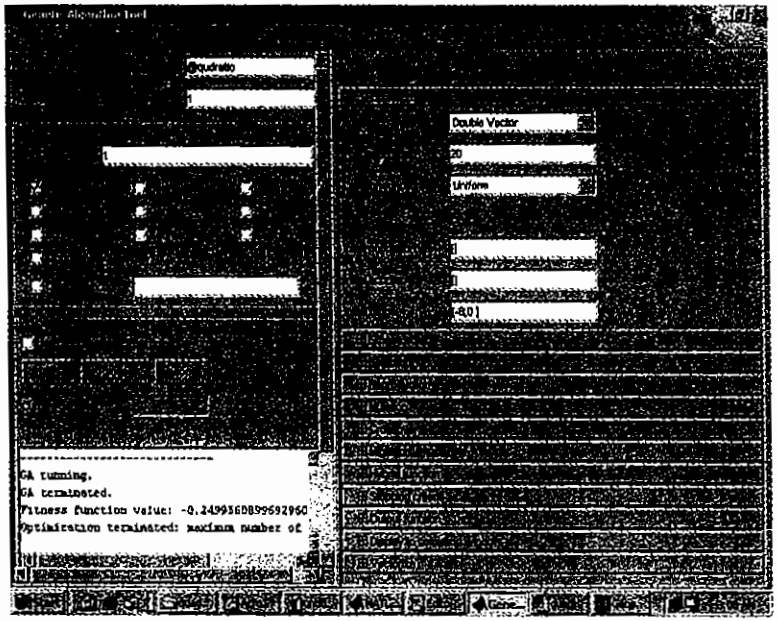


Figure 19-41 Genetic algorithm tool for quadratic equation.

Output

The output showing the best fitness for 50 generations is shown in Figure 19-42.
The status and results for this functions for 50 generations are shown in Figure 19-43.

3. Create a Gatool to maximize the function $f(x1, x2) = 4x_1 + 5x_2$ within the range 1~1.1.

Function Definition

Define the given function $f(x1, x2) = 4x_1 + 5x_2$ in a separate m-file as shown in Figure 19-44.

Creation of Gatool

On typing "gatool" in the command prompt, the GA toolbox opens. In tool, for fitness value type @twofunc and mention the number of variables defined in the function. Select best fitness and best individual in plot and specify the other parameters as shown in Figure 19-45.

Output

The output for 50 generations is as shown in Figure 19-46. The output also shows the best inidvidual.
The status and result for this function are shown in Figure 19-47.

4. Use Gatool and minimize the function $f(x1, x2, x3) = -5\sin(x1)\sin(x2)\sin(x3)+[-\sin(5x1)\sin(5x2)\sin(x3)]$, where $0 \leq xi \leq pi$, for $1 \leq i \leq 3$.

Function Definition

Define the given function
$f(x1, x2, x3) = -5\sin(x1)\sin(x2)\sin(x3) + [-\sin(5x1)\sin(5x2)\sin(x3)]$
in a separate m-file as shown in Figure 19-48.

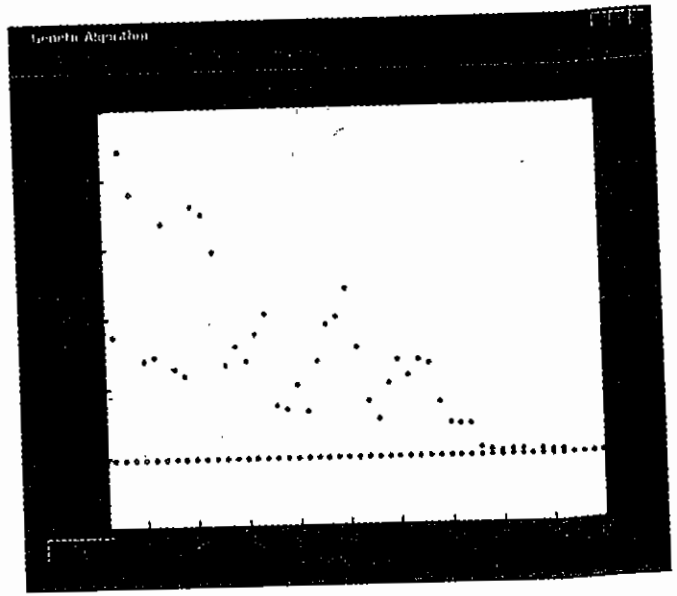


Figure 19-42 Output response (best fitness).

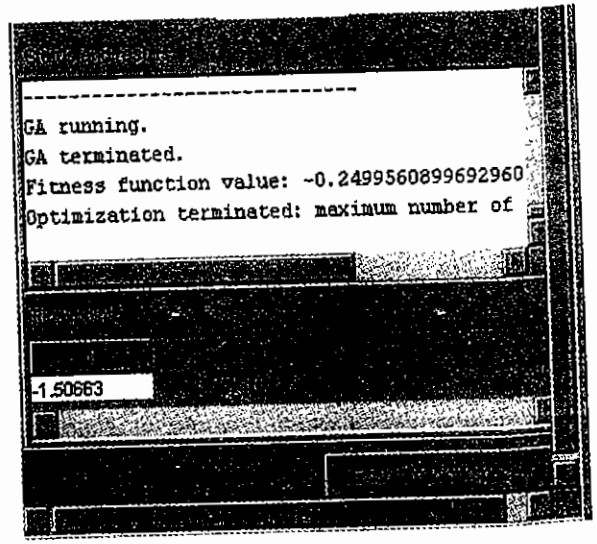


Figure 19-43 Status and results.

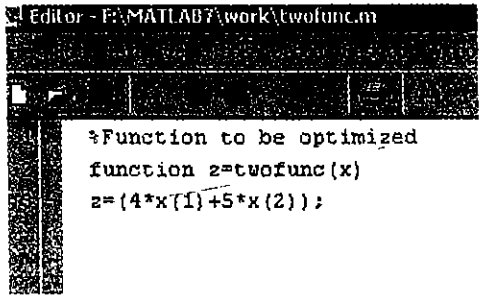


Figure 19-44 M-file showing defined function.

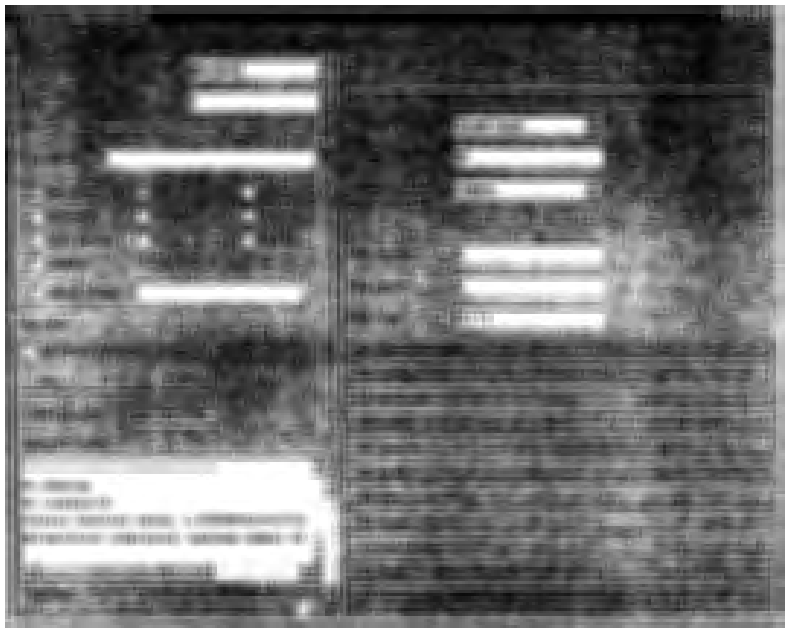


Figure 19-45 Genetic algorithm tool for given function.

Creation of Gatool

On typing "gatool" in the command prompt, the GA toolbox opens. In tool, for fitness value type @sinefn and mention the number of variables defined in the function. Select best fitness in plot and specify the other parameters as shown in Figure 19-49.

Output

The output for 100 generations is as shown in Figure 19-50.
The status and result for this function are shown in Figure 19-51.

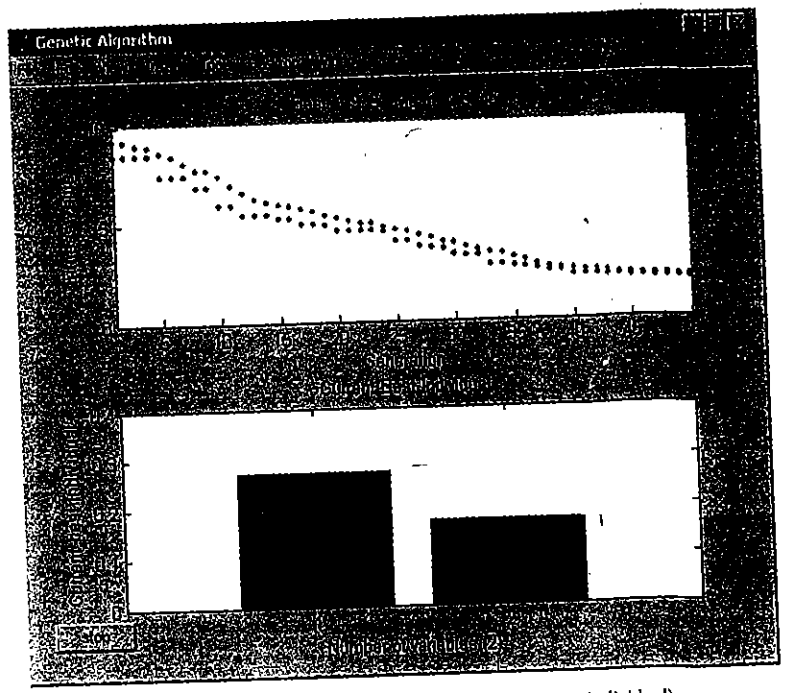


Figure 19-46 Output response (best fitness and best individual).

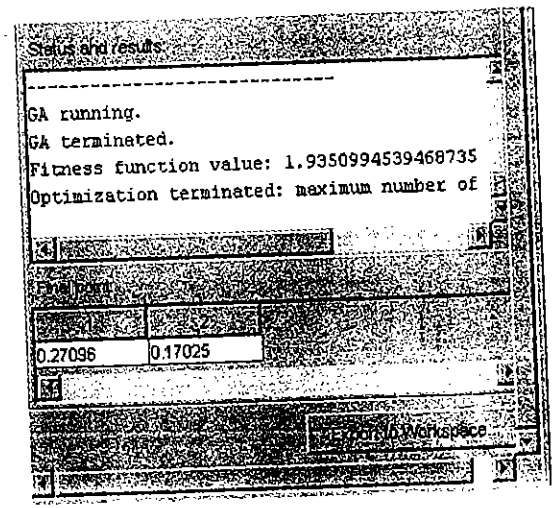


Figure 19-47 Status and results.

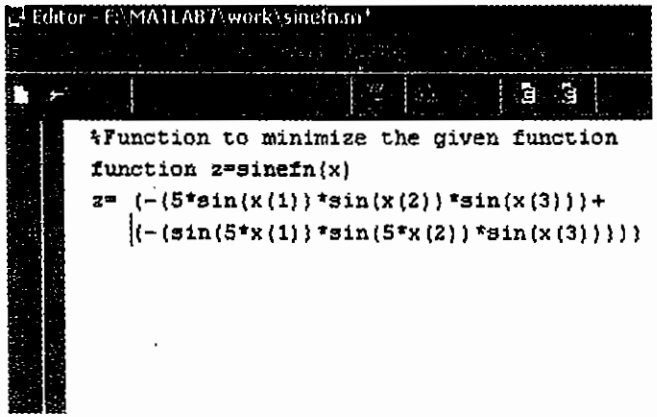


Figure 19-48 M-file showing defined sine function.

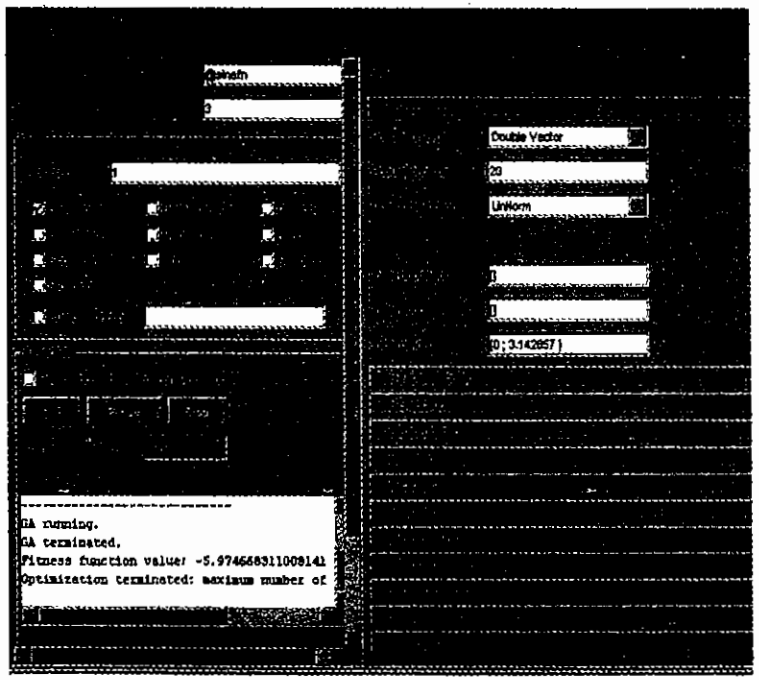


Figure 19-49 Genetic algorithm tool for sine equation.

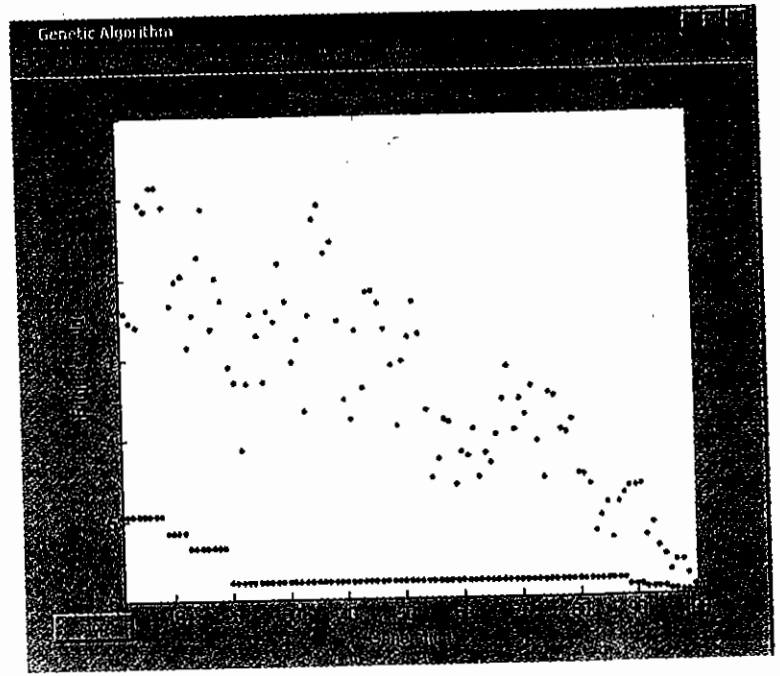


Figure 19-50 Output response (best fitness).

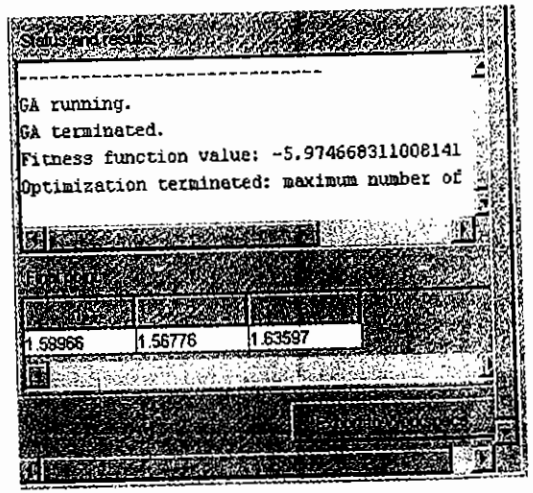


Figure 19-51 Status and results.

19.10 Summary

In this chapter soft computing techniques are implemented using MATLAB software. MATLAB software is very user-friendly and enables the user to simulate the ideas of soft computing for their applications. This chapter provides an overview of the various commands and GUI module involved in MATLAB for neural networks, fuzzy logic and GA approaches. The source codes developed using these commands and GUI toolbox for the soft computing techniques have also been included for the ready reference of the reader.

19.11 Exercise Problems

1. Implement the AND function using perceptron network using a MATLAB program.
2. Write a MATLAB program to apply back propagation network for a pattern recognition problem.
3. Implement OR function with bipolar inputs and targets with a MADALINE neural net.
4. Write a program to create an ART 1 network to cluster 7 input units and 3 cluster units.
5. Develop a Kohonen self-organizing feature map for a image recognition problem.
6. Write a program to implement various operations of fuzzy sets.
7. Implement the properties of fuzzy sets using an m-file.
8. Develop an m-file to perform compositional operations in fuzzy relations.
9. Maximize Rosenbrock's function using a MATLAB program.
10. Minimize Rastrigin's function using MATLAB GUI GA toolbox.
11. Given a polynomial equation of the form $f(x) = 4x^4 + 3x^3 + 2x^2 + x + 7$, find its roots using GA approach.
12. Consider a hyperbolic tangent function. Maximize it within the range $0 < x < 22/7$. Apply two-point crossover and tournament selection process. Construct a GA GUI toolbox.
13. Find the roots of the quadratic equation using genetic algorithm. The quadratic equation is $f(x) = 6x^2 + 5x + 3$.
14. Find the solution of the function $f(x) = \sin(7\pi x) + 10$ with the constraint $-3 < x < 3$ by using genetic algorithm and MATLAB programming.
15. Write a program to minimize "cosine" function.

Bibliography

1. Pal, S. K. (1998, January–April) Soft computing tools and pattern recognition. *IETE Journal of Research*, 44(1–2), 61–87.
2. Rao, D. H. (1998, July–October) Fuzzy neural networks. *IETE Journal of Research*, 44(4–5), 227–236.
3. Russo, M. (1998, August) FuGeNeSys – A fuzzy genetic neural system for fuzzy modeling. *IEEE Transactions on Fuzzy Systems*, 6(3), 373–388.
4. Pal, S. K. and Srimani, P. K. (1996) Neurocomputing motivation, models, and hybridization. *Proceedings of IEEE Computers*, 24–28.
5. Jain, A. K. and Mao, J. (1996, March) Artificial neural networks: A tutorial. *Proceedings of IEEE Computers*, 31–54.
6. Lippmann, R. P. (1987, April) An introduction to computing with neural nets. *IEEE ASSP Magazine*, 4–22.
7. Hush, D. R. and Horne, B. G. (1993, January) Progress in supervised NN. *IEEE Signal Processing Magazine*, 8–32.
8. Kohonen, T. (1990, September) The self organizing map. *Proceedings of IEEE*, 78(9), 1464–1478.
9. Kangas, J. A., Kohonen, T. K. and Laaksonen, J. T. (1990, March) Variants of self-organizing maps. *IEEE Transactions on Neural Networks*, 1(1), 93–99.
10. Pal, N. R., Bezdek, J. C. and Tsao, E. C-K. (1993, July) Generalized clustering networks and Kohonen's self-organizing scheme. *IEEE Transactions on Neural Networks*, 4(4), 549–557.
11. Carpenter, G. A. and Grossberg, S. (1992, September) A self-organizing neural network for supervised learning, recognition, and prediction – can neural networks learn to recognize new objects without forgetting familiar ones? *IEEE Communication Magazine*, 38–49.
12. Carpenter, G. A., Grossberg, S. and Rosen, D. B. (1991) Fuzzy ART: fast stable learning and categorization of analog patterns by an adaptive resonance system. *Neural Networks*, 4, 759–771.
13. Frank, T., Kraiss, K.-F. and Kuhlen, T. (1998, May) Comparative analysis of fuzzy ART and ART-2A, network clustering performance. *IEEE Transactions on Neural Networks*, 9(3), 544–559.
14. Fernandez-Delgado, M. and Ameneiro, S. B. (1998, January) MART: A multichannel ART-based neural network. *IEEE Transactions on Neural Networks*, 9(1), 139–150.
15. Setiono, R. and Liu, H. (1996, March) Symbolic representation of neural networks. *IEEE Computer*, 71–76.
16. Shang, Y. and Wah, B. W. (1996, March) Global optimization for neural network training. *IEEE Computer*.
17. Jain, A. K. and Mao, J. (1996, March) ANN: A tutorial. *IEEE Computer*.
18. Ng, K. and Lippmann, R. P. (1990, May) *A Comparative Study of the Practical Characteristics of Neural Network and Conventional Pattern Classifiers*, Masters Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.
19. Bortoian, G., Degani, R. and Williams, J. L. (1991) Neural networks for ECG classification. *IEEE Neural Networks*, 269–272.
20. Geczy, P. and Usui, S. (1997) Learning performance measures for MLP networks. *IEEE, IJCNN*, 1845–1849.