

9

Future Horizons for Architecture

9.1 INTRODUCTION

The von Neumann computer performs poorly on certain tasks, such as emulating the natural information processing that humans handle routinely. In many real-world applications, the processing of information in a reasonable time requires exploiting the tolerance for imprecision and uncertainty for which von Neumann machines are ill adapted. To overcome this problem, many theories and technologies have been proposed. Zadeh [ZAD 94, ZAD 95] refers to these theories and technologies as *soft computing*. Basically, soft computing is a collection of methodologies that in one way or another aim to exploit the tolerance for imprecision and uncertainty to achieve tractability, robustness, and low solution cost. There are three main classes of methodologies that form soft computing; they are *neural networks*, *fuzzy logic*, and *probabilistic reasoning*. Although each of these classes of methodologies may be used to resolve certain types of applications, they are in fact complementary to each other, and in many cases it is better to employ them in combination rather than exclusively.

In this chapter, two of the constituents of soft computing that currently offer the most potential for future architectures are discussed. These are neural networks and fuzzy logic. In addition, one subject subsumed by fuzzy logic, multiple-valued logic, is also explained.

Neural networks address the issue of effective information organization and processing. Since biological brains are working examples of massively parallel, densely interconnected, self-organizing computational networks, they represent an ideal prototype after which special-purpose hardware can be modeled. To extract a design of a machine from a model of the brain's functioning requires an intimate understanding of the brain's most basic processing elements, the neurons, and the dynamics of their operation. This chapter examines the neuron together with the dynamics of neural processing, and surveys some well-known proposed neural networks.

Multiple-valued logic addresses the need to conserve chip area in order to have complex circuits. Multiple-valued logic circuits allow signals with more than two values and therefore provide a significant savings in chip area. This chapter describes the basic features of this logic.

Finally, fuzzy logic attempts to deal effectively with imprecision and approximate reasoning, as opposed to precision and formal reasoning, and it overcomes some of the inconveniences associated with the classical logic. It mimics the remarkable ability of the human mind to summarize data and focus on decision-relevant information. Fuzzy logic has been applied successfully in many cases. This chapter defines fuzzy logic, explains its use in control systems, and discusses the future of this theory.

9.2 NEURAL NETWORKS

As a blueprint for computer architecture, no single source has as much potential or is as challenging as the human brain. The brain's intrinsic relevance to the science of computers lies in its obvious information-processing capabilities and its incorporation of desirable computing characteristics. The most significant computing characteristics that are evident in the structure of the brain include concurrent and distributed data processing, functional modularity, massive parallelism, and a capacity for self-organization. As Vidal [VID 83] has suggested, the incorporation of precisely these characteristics into the design of a computing system is a prerequisite of the successful management of the functional and testing complexity of future VLSI designs.

Since the significance of these characteristics to the development of advanced computer architecture is beyond dispute, the importance of understanding the architecture of the brain that provides for these characteristics is essential. The principal and most evident architectural features of the brain relevant to its

processing characteristics are those of layering, modularity, dense interconnections, and distribution of input processing.

Layering. The cells of the brain are grouped into large networks according to a plan of hierarchical superposition, permitting information to be processed in a stratified manner, layer by layer, Fairhurst [FAI 78] describes the brain's mechanism of information-processing in terms of a hierarchically structured model of neural networks.

Modularity. Areas of the brain are divided into modules codetermined by the design-integrated considerations of sensory input mapping and functional output.

Dense interconnections. Particular cellular interconnections between and within layers provide for data sharing and also serve as feedback and feedforward mechanisms for transmitting data among interconnected regions containing stored data.

Distribution of input processing. Identical or similar input may be differentially represented as it passes through different processing routes (data channels) governed by specific relay mechanisms (category triggers) within the brain.

In the next section the basic terminology and concepts of neurophysiology will be reviewed. The basic concepts of neurophysiology provide a common ground for understanding the neural network models that are introduced later. The development of any model of neural processing requires that the fundamentals of neurophysiology be related to the elements of a computational model. The reader who is not interested in the details of neurophysiology can skip Section 9.2.1 and go directly to Section 9.2.2.

9.2.1 Fundamentals of Neurophysiology

The basic brain processing unit is the nerve cell, called the *neuron*. Figure 9.1 shows the main components of a neuron. The output of the neuron, called *axons*, branch out directionally from the cell body, the *soma*, and reach out to terminate on other nerve cell bodies, thus establishing contact between two neurons. The axons are used for transmitting information between neurons. The connection between a neuron and an axon is called a *synapse*. At the synapse the transmission of information from one cell to another occurs.

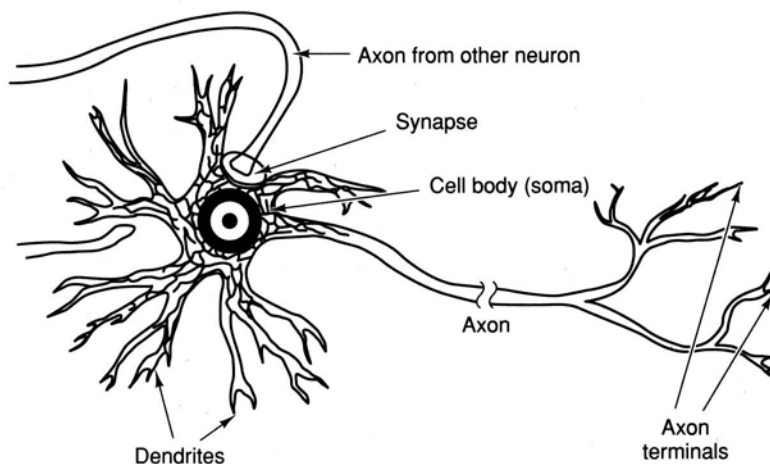


Figure 9.1 Main components of a neuron.

In each neuron there is a wall-like structure, called the *membrane*, that keeps substances beneficial to the cell in and undesirable elements out. Because of this barrier, the concentration of molecules inside and outside the cell are not equal. Both the intracellular and the extracellular spaces are filled with a dilute aqueous salt solution that causes most molecules to break down into charged atoms. There are two types of charged atoms, positive (cations) and negative (anions). There is an unequal distribution of charged

molecules, particularly small cations, between the inside and outside of the cell. This permits the membrane to be selectively permeable in favor of small cations, setting up an electrical relation called the *membrane potential*. The membrane potential resides in tension or equilibrium in which forces pushing small cations out of the cell are in balance with the forces pushing them back into the cell.

The membrane potential is an essential element for the transfer of information within the central nervous system (CNS). A stimulus disturbs the resting membrane potential of a cell by changing its permeability to certain ions. Whenever the membrane potential change is in the positive direction and reaches a threshold level (the assumption will be made that the threshold value is equivalent in all cells and remains constant), an action potential is generated. The action potential is a stereotyped sequence of depolarizations and hyperpolarizations occurring spontaneously at the membrane surface. The action potential spreads along the membrane surface from the point of stimulation into neighboring regions of the membrane, thus progressing from point to point from the soma down the length of the axon. When the action potential arrives at a synapse it becomes a stimulus, known as a *postsynaptic potential* (PSP), to the next cell, whose effect is to change the permeability of the next cell to small ions.

There are two types of postsynaptic potentials in the CNS: *excitatory postsynaptic* (EPSP) and an *inhibitory postsynaptic* (IPSP). An EPSP causes the membrane potential of the receiving cell to have a more positive value, thus increasing the likelihood of an action potential generation. An IPSP has the opposite effect on the membrane potential, thus making an action potential harder to generate. Each PSP, whether excitatory or inhibitory, has a prespecified life span. Its influence on a cell's membrane potential is greatest at the point and time of arrival, decreasing at a constant rate until it either serves to generate an action potential or disappears.

Typically, the EPSPs are below threshold: they cannot change a membrane potential enough to initiate an action potential. Therefore, two or more EPSPs must combine in an additive relation (sum).

A simple illustration of action potential is shown in Figure 9.2 [GRI 81]. Figure 9.2a shows that excitatory postsynapses (E_1 and E_2) and an inhibitory postsynapse (I) are stimulated at S , and the postsynaptic change is recorded at R . Figure 9.2b shows that E_1 alone cannot produce an action potential, but that the stimulation of both E_1 and E_2 causes an action potential because the summation of E_1 and E_2 exceeds the threshold level. When all three synapses (E_1 , E_2 and I) are stimulated, the inhibitory synapse (I) blocks the development of an action potential. The key concept here is that "Neurons are analogue devices" [AND 83].

The EPSPs and IPSPs can be represented as digital pulses, with approximately the same height and duration, and can be thought of as binary bits. A pulse at a given synapse may either add to (if it is an EPSP) or subtract from (if it is an IPSP) the membrane potential. This potential can be represented as an analog voltage that corresponds to the algebraic sum of the inputs. When the summed inputs to the cell exceed the threshold level, the cell puts a pulse on the axon. Whenever this occurs, the voltage in the cell body is reset to the initial value.

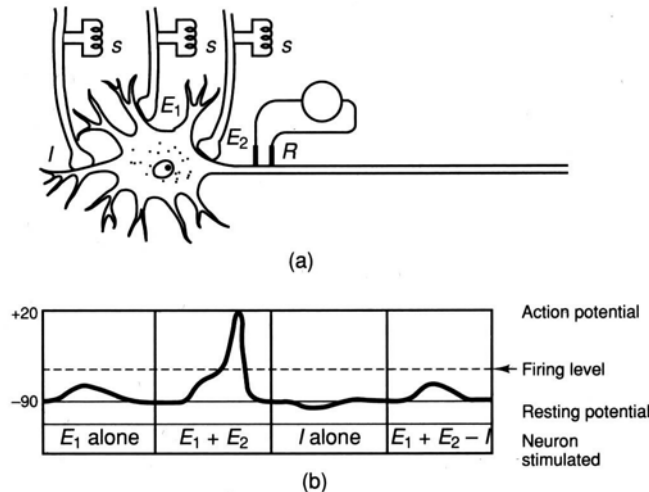


Figure 9.2 Example of neural communication (from [GRI 81]). Reprinted with permission of Simon & Schuster, Inc. from the Macmillan College text "Introduction to Human Physiology" 2/E by Mary Griffiths. Copyright 1981.

Information-processing in the nervous system. The simple illustration of the flow of information-processing of the brain from the sensory receptors (input) to motor neurons (output) is shown in Figure 9.3. The center box with the question mark, which is primarily composed of the brain, is far less known at present. Input of any sensory signal is performed by receptor cells, and eventually output is transmitted to motor neurons terminating on muscle cells. How a piece of information is processed after the input and before the output is far less known. In this section, each part of the information-processing is examined briefly.

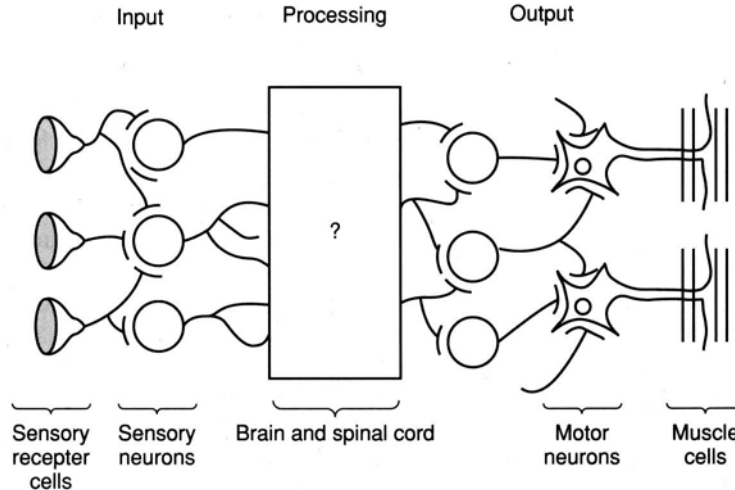


Figure 9.3 Flow of information-processing.

The function of sensory receptors is to transmit information about the internal and external environments to the central nervous system. A receptor responds to a stimulus by developing a generator potential in a sensory neuron. If this potential reaches threshold level, it generates action potentials in the sensory neuron. The greater the generator potential, the greater the frequency of impulses generated and the stronger the sensation or response.

When a sensory neuron sends action potentials to the central nervous system, it branches extensively, making many synaptic contacts. The intensity of a sensation is determined by the frequency of impulses in the sensory neurons and the number of receptors stimulated. Meaning is given to the sensory input, and

sensory information may be stored in the central nervous system.

A hierarchy exists in the central nervous system for the control of muscle activities. Simple reflexes are coordinated at the level of the spinal cord, but are modified by impulses from several levels of the brain. Higher functions of the brain depend on the central nervous system. The higher functions include consciousness, learning, memory, language, and thought. The results of the processing are sent to the motor neurons, which terminate at muscle cells.

Most of the activities of information processing in the central nervous system are unknown because of limitations of present technology [PAL 82]. For example, the activities of only a few neurons can be recorded at the same time. Therefore, it is practically impossible to reconstruct a global activity.

Even though the computational principle of models based on the brain rely on the neuron's simple firing mechanism, the methods, purposes, and objects vary from model to model. The following sections review these variations of the brain models.

9.2.2 Artificial Neural Networks

Work on neural net models has a long history. Development of detailed mathematical models began more than 50 years ago with the work of McCulloch and Pitts [MCC43], Rosenblatt [ROS 62], and Widrow [WID 59, WID 60]. More recent works by Hopfield [HOP 82, HOP 84, HOP 86], Rumelhart and McClelland [RUM 86a], and others have led to a resurgence in the field. This new interest is due to the development of new net methods and algorithms and new VLSI implementation techniques, as well as the growing fascination with the functioning of the human brain. Interest is also increasing because areas of speech and image recognition require enormous amounts of processing to achieve humanlike performance. Artificial neural networks (ANNs) provide one technique for obtaining the processing power required, using large numbers of processing elements operating in parallel.

Although ANNs can be simulated on conventional computers, they are intended to be implemented on special-purpose hardware. ANNs are capable of learning, adaptive to changing environments, and able to cope with serious disruptions.

The artificial neuron was designed to mimic some of the characteristics of the biological neuron. Each neuron has a set of inputs and one or more outputs. A weight is assigned to each input. This weight is analogous to the synaptic strength of a biological neuron. All the inputs are multiplied by their weights and then are summed to determine the activation level of the neuron. Once the activation level is determined, an activation function is applied to produce the output signal. Figure 9.4 presents an artificial neuron that has n inputs, labeled x_1, x_2, \dots, x_n , and one output denoted by y . The output y can be produced as

$$y = f(u),$$

where

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

and f is an activation function.

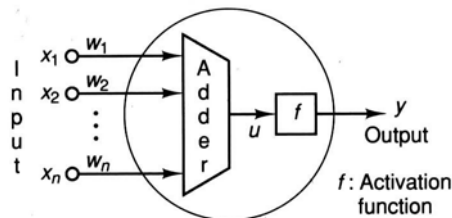


Figure 9.4 Architecture of an artificial neuron.

One simple activation function is the *hard-limiting* function. As shown in Figure 9.5, hard-limiting function is defined as

$$y = \begin{cases} 1, & \text{if } u > T \\ -1, & \text{otherwise} \end{cases}$$

where T is a constant threshold value.

Another activation function is the nonlinear *sigmoid* function. As shown in Figure 9.6, a sigmoid function is expressed as

$$y = 1/(1 + e^{-u}).$$

The sigmoid function is often used because it is differentiable and makes construction of neural network models easier.

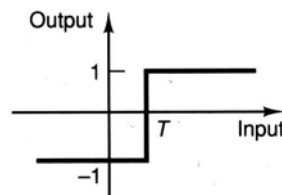


Figure 9.5 Hard-limiting function.

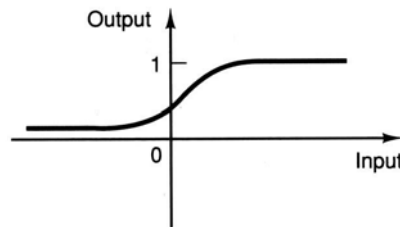


Figure 9.6 Sigmoid function.

Taxonomy of ANN models. ANN models constitute a large class of computational mechanisms that share together the basic features of parallel operation and dense interconnection between the processing elements. At the same time, major differences exist among the individual models regarding their architecture, learning rules, and mode of interaction with the environment. A general taxonomy of these models will prove useful in understanding their functionality and fitness to practical applications.

The most general distinction among different ANN models is considered to be the extent to which the environment specifies the input/output mapping that the ANN is supposed to learn. If the environment provides the training examples in the form of input/output pairs of vectors, the mode of operation of the ANN is said to be *supervised*. This mode is also called *learning with a teacher*, since the environment serves as a teacher to the network by providing detailed examples of what is to be learned. If, on the contrary, the environment specifies the input but not the output, the learning is *unsupervised*. In the latter case the network has to discover on its own the solution to the learning problem. Somewhere in between supervised and unsupervised learning lies *reinforcement* learning: some output information is supplied by the environment, but this information is in the form of evaluation of the ANN's performance, rather than in the form of training examples. Sometimes reinforcement learning is called learning with a *critic*, as opposed to learning with a teacher, because the environment does not specify what is to be learned, but only if what is being learned is correct. Figure 9.7 shows the three classes of learning algorithms: supervised, reinforcement, and unsupervised.

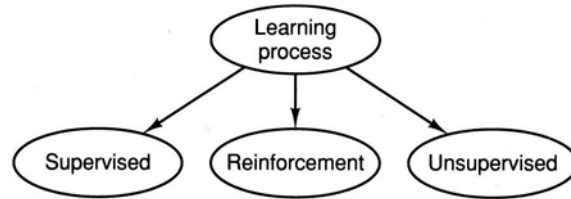


Figure 9.7 Different classes of ANN learning algorithms.

Another important distinction among different ANN models is based on their architectures. Here, architecture refers to the type of processing performed by the artificial neurons and the interconnection between them. As shown in Figure 9.8, ANN can be divided into two groups: *deterministic* and *stochastic*.

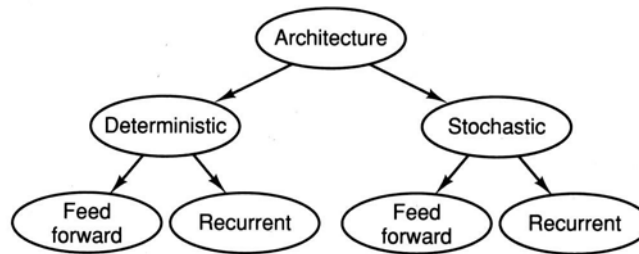


Figure 9.8 Different types of ANN architectures.

Deterministic networks always produce the same output result when presented with the same input, while the output for a given input in stochastic networks can vary according to some output probability distribution. Stochastic models are usually harder to analyze and simulate, but at the same time they are more realistic in many applications. For example, if the output of a certain physical system, whose input does not change, is to be measured several times with standard measurement devices, the readings will be close to each other, but nevertheless different. In this case it would make much more sense to match the input of the system to the probability distribution of the output rather than to a single hypothetical average of all measurements.

If the directed connectivity graph of an ANN has loops or cycles, the network is said to be *recurrent*; otherwise, it is called *feed-forward*. (The directed connectivity graph of an ANN is a directed graph in which the vertices correspond to the neurons of the network and edges correspond to the connections between the neurons.) While this distinction does not seem too critical at first glance, the feed-forward networks turn out to be much easier to analyze and train than their recurrent counterparts. At the same time, there are usually restrictions about the type of mappings that strictly feed-forward networks can learn. A recurrent model is always computationally more powerful than its corresponding feed-forward model, because the feed-forward model is a special case of the more general recurrent model.

Based on the learning process criteria, the ANN models in a given group, supervised learning, reinforcement learning, and unsupervised learning, have similar functionality and are usually used in similar applications. These three groups and the principal task that each is applied to are discussed next.

Supervised Learning. Supervised learning models can use either deterministic or stochastic processing elements. The class of deterministic supervised learning algorithms solves the fundamental problem of function approximation. This problem can be divided into two tasks:

1. **Loading task.** Store a set of p pairs of input/output patterns (x^k, y^k) , $k=1$ to p , in such a way that when presented with any input pattern x^k the network responds by producing the correct output pattern y^k . The set of input/output patterns (x^k, y^k) is called the *training set* of the task. The number of elements in the input and output patterns determines the dimensions of the input and output spaces, respectively; these dimensions need not be the same. In this way, the training set defines a

function that maps the input space onto the output space.

- 2. Generalization task.** When presented with a *new* input pattern, different from any pattern in the training set, the network responds with an *adequate* output, a pattern that depends on the new input in the same manner as the output patterns in the training set depend on their corresponding input patterns.

Algorithms for training feed-forward deterministic supervised networks include, among others, Adaline and Madaline [WID 60, WID 88], back propagation [RUM 86a, RUM 86b], quick propagation [FAH 89], conjugate gradient [KRA 89, MAK 89] and cascade correlation [FAH 90]. A great deal of research in ANN is concentrated on this class of algorithms because of their relatively simple learning rules and their usefulness in practical applications. Deterministic feed-forward supervised models are employed in many successful ANN applications: speech recognition and synthesis [LIP 89], automatic target recognition [GOR 88], car navigation [POM 89], image compression [COT 87], signal prediction and forecasting [LAP 87], handwritten character recognition [LEC 90], and others.

Examples of recurrent deterministic supervised models are Hopfield networks [HOP 82], bidirectional associative memory [KOS 92], mean-field theory [PET 87], recurrent back propagation [PIN 87, ALM 88], real-time recurrent learning [WIL 89] and time-dependent recurrent back propagation [PEA 89]. These algorithms can be used to train ANNs to perform input/output mappings that cannot be learned by feed-forward networks, for example, finite-state machines and sequence recognition. This increase in approximation power is paid for by higher computational costs, which sometimes make the learning and simulation of the problem being investigated infeasible.

If stochastic processing elements are used in the supervised learning paradigm, the function approximation problem can be extended to functions whose result is not a single value, but a probability distribution. The class of models that solves the task of associating input and output probability distributions includes Boltzmann machines [HIN 86], expectation maximization [RED 84], and Cauchy machines [SZU 86]. These models require substantial computational resources even for small problems and typically scale up poorly. In spite of their representational power, they are not likely to be the modeling tool of choice in the future unless their requirements for computational resources decrease dramatically.

Reinforcement Learning. The second largest class of learning algorithms is the well-defined group of reinforcement learning algorithms. The task they try to solve is considerably more difficult than the associative task that supervised learning algorithms tackle. The correct output that corresponds to a given input is not known to the reinforcement learning algorithm. The only possible way for the system to discover the correct output is by trial and error, which requires exploratory behavior on the part of the system. This can be achieved by using stochastic units that produce different outputs for the same input. For example, if the same input vector x is presented to the system two times in a row, the outputs will be two different vectors, $y(1)$ and $y(2)$. The learning algorithm compares the correctness of these two vectors and adjusts the system in order to change the probabilities that these two vectors will be output subsequently: the probability of the more successful output is increased, while the probability of the less successful is decreased.

Essential in this process is the ability of the system to evaluate which of the two vectors is better. The search of the system for the correct output is guided by a single scalar variable called *reinforcement signal* (usually in the range $[-1,1]$) that evaluates the correctness of the output that the system has chosen to produce. A value of $+1$ indicates a perfect guess, while a value of -1 is returned by the environment when the output is completely incorrect. In practice, all reinforcement learning algorithms are optimization schemes that try to maximize the value of the reinforcement signal.

The environment can in its turn be either deterministic or stochastic. In deterministic environments the reinforcement signal is always the same for a given input/output pair. In this way the system is guaranteed a reward each time it makes a successful guess. In stochastic environments, a particular input/output determines only the *probability* of certain reinforcement, while the actual value of the reinforcement comes from a probability distribution. With this type of environment, the system can in fact receive low

reinforcement even if the output has been good. This makes clear that learning in stochastic environments is much harder than learning in deterministic ones.

In addition to that, the environment might delay the reinforcement. For example, if a robot balances a broomstick and makes a wrong hand movement, the negative reinforcement will come only after the broomstick has fallen down, before which the robot will have made many more movements. It is very difficult to determine exactly which of these hand movements was wrong and led to the loss of balance so that it can be punished. In this case we have the problem of *temporal* credit assignment in addition to the usual *structural* credit assignment problem. Structural credit assignment deals with determining the change in which connection led to better performance, while temporal credit assignment deals with determining *when* (at which moment in time) correct outputs have been generated and when the outputs have been incorrect.

Examples of reinforcement learning algorithms are associative reward-penalty [BAR 85], TD(λ) [SUT 88], Q-learning [WAT 92], and adaptive heuristic critic and the REINFORCE group of gradient ascent methods [WIL 92].

Unsupervised Learning. The third and last largest class of ANN models is the unsupervised algorithms. Target values are not supplied, nor is reinforcement provided. The network has to discover for itself patterns, features, regularities, correlations, or categories in the input data and code for them in the output. The units and connections must display some degree of *self-organization*. Unsupervised learning algorithms can perform clustering, prototyping, principal component analysis, encoding and feature mapping among other tasks encountered in science and engineering.

Several sorts of regularities can be discovered by unsupervised learning models [HER 91]:

1. **Clustering.** Each input should be classified in one of several groups, or clusters, based on the similarity of the input to the vectors in the corresponding cluster. The output layer of the system has as many units as number of clusters; each unit is responsible for one and only one cluster. If the unit is on, this will mean that the input vector is classified as belonging to the cluster for which this unit is responsible. If the classification is to be unequivocal, only one unit in the output layer should be on at a time. In this way the output units compete with each other to represent the input vector; hence the algorithms that provide this property are called *competitive* learning algorithms.
2. **Prototyping.** One step beyond clustering is prototyping: instead of merely determining the correct cluster, the network is expected to provide a typical example of that cluster.
3. **Familiarity.** The task of the system is to produce a single continuous-valued output that estimates how similar the current input is to the input vectors that the system has observed in the past.
4. **Principal component analysis.** In many cases the measurable variables that constitute the input vector are not independent; rather, a hidden set of independent variables called *factors* or *principal components* exists that actually produces the measurable output of the system. The task of the learning algorithm is to discover this set of factors.
5. **Encoding.** When encoding a certain input vector, the system should reduce the dimensionality of the input without losing too much discriminating information. One way to do that is to use principal component analysis and describe the input vector in the space of the principal components.
6. **Feature mapping.** This problem includes the encoding problem with the additional requirement that the topology of the input space be preserved so that close vectors in the input space remain close in the space of the transformed image.

Examples of unsupervised learning algorithms are self-organizing feature maps [KOH 82], adaptive resonance theory [GRO 87], cognitron [FUK 75], and neocognitron [FUK 80].

The following sections describe some of the fundamental networks, such as Adaline, Madaline, and perceptron networks. The sections introduce each of these networks and describe how they are implemented. The decision space, which describes the network's ability to distinguish between patterns or decisions, will be analyzed for each network. In addition, the Hopfield network is also explained. The Hopfield networks are important because of their direct implementation in hardware.

Adaptive linear neurons. Widrow et al. [WID 60, WID 88] have proposed an artificial neural network with *adaptive linear neurons* called *Adaline*. The neurons are characterized as adaptive because their weights are adjustable and as linear because the function of the neuron is linear.

Many of the proposed ANNs are based on Widrow's Adaline method. This neural method performs very well for classifying linear pattern/decision vector space problems. Its roots are based on the Hebbian rule [HEB 49], which states that a physical change has to take place in a network in order to support learning and that this physical change requires a strengthening of the connections among elements of the network. More specifically, Hebb proposed the following: *Whenever two connected neurons are active at the same time, the strength of the connection between them should be increased.*

This learning rule reflects the principle of *contiguity* that is believed to be the basis of biological learning [OSH 90]. This principle states that, when two events (images, ideas) occur at the same time, they are associated with each other so that, when one of them becomes active in future time, the other will be activated too. For example, let *A* and *B* be two neurons, where *A* is one of the neurons providing input to neuron *B*. If neuron *A*'s activity tends to be high whenever neuron *B*'s activity is high, the future contribution that the firing of neuron *A* makes to the firing of neuron *B* should increase.

The Hebbian rule can be accomplished by changing weights on the inputs of neurons in response to some function of the correlated activity of the connected units. Other networks that are based on the Hebbian rule and related to the Adaline are the Madaline and perceptron networks. As is shown in later sections, these networks, unlike the Adaline, can describe nonlinear decision space, such as the exclusive-or (XOR) function.

Figure 9.9 shows an Adaline neuron with n inputs $x_1, x_2, \dots,$ and x_n . Each input can take only one of two binary values, +1 or -1. In addition, the neuron also has a constant input x_0 , which has the value +1 all the time. A weight is associated to each input. These weights can take any real number. The weight corresponding to the input x_0 is w_0 and is called the *bias weight*. Later we shall see how this bias weight controls the threshold level. All the inputs are multiplied by their weights and then summed to determine the activation level of the neuron, u . For the output of this neuron, y , to be connected to the input of other neurons, the real value u needs to be converted into a binary value of +1 or -1. A hard-limiting activation function does this conversion. The output of the neuron is assigned a value of +1 if u is greater than zero and a value of -1, otherwise.

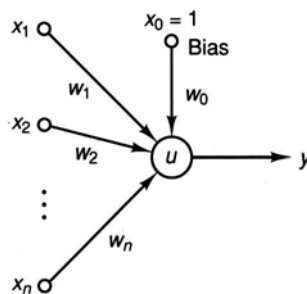


Figure 9.9 An Adaline neuron.

To clarify the operation of an Adaline neuron, let us consider its use in representing some elementary logic functions such as the OR and AND functions. Figure 9.10 shows the desired output y for different

combinations of the inputs x_1 and x_2 for these functions.

x_1	x_2	y
-1	-1	-1
-1	+1	+1
+1	-1	+1
+1	+1	+1

x_1	x_2	y
-1	-1	-1
-1	+1	-1
+1	-1	-1
+1	+1	+1

(a)
(b)

Figure 9.10 Truth tables for the (a) OR and (b) AND functions.

Figure 9.11 shows a two-input Adaline neuron. Suppose that we want this neuron to represent an OR function. This requires finding the weights w_0 , w_1 , and w_2 , so that the neuron can represent the desired mapping. For example, if $w_0=1.5$, $w_1=+1$, and $w_2=+1$, then the neuron represents an OR function. (Later we will learn how these weights can be found in general.) A two-input Adaline neuron is also able to represent an AND function; this can be done by choosing $w_0=-1.5$, $w_1=+1$, and $w_2=+1$; see Figure 9.11.

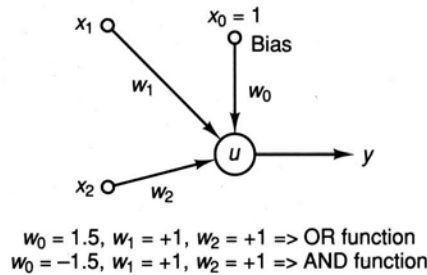


Figure 9.11 A two-input Adaline.

A single neuron divides the input patterns into two classes, one for which the output is +1 and the other for which the output is -1. The distinction between outputs +1 and -1 occurs when the weighted sum u equals 0. For example, in Figure 9.11, we have

$$x_1w_1 + x_2w_2 + x_0w_0 = 0,$$

or

$$x_1w_1 + x_2w_2 + w_0 = 0,$$

or for the AND function,

$$x_1 + x_2 - 1.5 = 0,$$

which is the equation of a straight line. As shown in Figure 9.12, this line divides the input pattern vector space into two parts. Notice that the input pattern (+1, +1), (which generates an output +1) is on one side of the line and the other input patterns (which generate an output -1) are on the other side of the line. Also notice that if we have not added the bias weight, w_0 , the line equation becomes $x_1+x_2=0$, which always passes through the origin. It is obvious that no lines passing through the origin can separate the input pattern (+1, +1) from other inputs. Therefore, it is necessary to have the bias weight w_0 in order to represent certain functions. However, there still exist some functions that cannot be represented even by addition of the bias weight. For example, in Figure 9.12 it can easily be seen that there is no single line that can put (+1, +1) and (-1, -1) in one class and (+1, -1) and (-1, +1) in another class. This is the XOR function. Thus a single Adaline neuron cannot represent an XOR function.

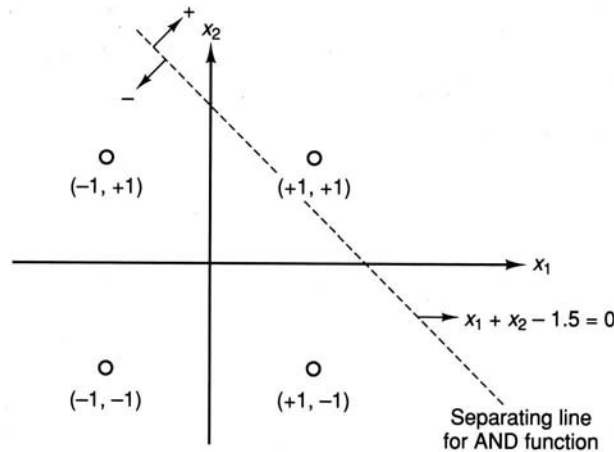


Figure 9.12 Input pattern space for a two-input Adaline.

In general, an N -input Adaline neuron can have 2^N possible input patterns. Each pattern can be classified as +1 or -1, so there can be a total of 2^{2^N} possible logic functions for this neuron. However, as we have seen, a single neuron can realize only linearly separable logic functions (decision regions). To realize functions that are not linearly separable, a combination of neurons is required.

Training an Adaline Network. The training process adjusts the weights so that the network produces target (desired) outputs for the given input patterns. Each iteration of the training process adjusts the weights so that the new weights produce an output closer to the target output.

Let us assume that t_k denotes the target output and y_k denotes the actual output for the k^{th} input vector. Also let δ_k denote the neuron error, which can be expressed as

$$\delta_k = (t_k - y_k).$$

Notice that each neuron error can only have the values of +2, 0, and -2. (Obviously, when $\delta_k = 0$, the neuron responds correctly to the input pattern x_k .) If $\delta_k = +2$, then the actual output is -1, which is supposed to be +1. Therefore, the weighted sum u should be increased until it becomes greater than 0. To increase u , the weight of positive inputs must be increased while the weight of negative inputs must be decreased. Similarly, if $\delta_k = -2$, then u should be decreased until it becomes less than 0. To decrease u , the weight of negative inputs should be increased while decreasing the weight of positive inputs. These changes can be formalized as follows:

$$W_{\text{new}} = W_{\text{old}} + \Delta W,$$

where $\Delta W = \rho \delta_k x_k$ determines the amount of change, W denotes the vector of weights, and ρ is a positive constant, called the learning rate.

This equation has various names, such as the *Widrow-Hoff learning law*, the *least mean square (LMS) learning law*, and the *delta rule*.

Madaline networks. The single Adaline neuron is not able to represent certain functions, such as XOR. To overcome this incapability the Adaline neurons can be connected in layers, with each layer having a number of such elements. The resulting network is called Madaline (many Adalines). Madalines are used to represent nonlinear (linearly inseparable) functions. Figure 9.13 shows a two-layer Madaline with a total of three Adalines.

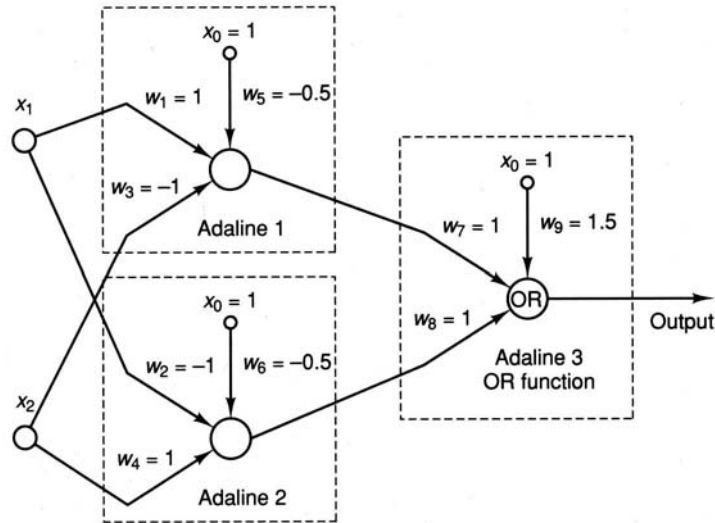


Figure 9.13 A two-layer Madaline model for the XOR function.

The Madaline has an adaptive first layer and fixed threshold functions for the second (output) layer. The neurons in the second layer can consist of AND functions, OR functions, or Majority vote takers.

As you will recall, the Adaline neuron separated the pattern vector space into two categories. The Madaline of Figure 9.13 with two Adalines can further divide the pattern vector space. Figure 9.14 shows the separating boundaries for the exclusive-or problem. Note that the two-layer Madaline has no trouble in classifying this simple nonlinear problem.

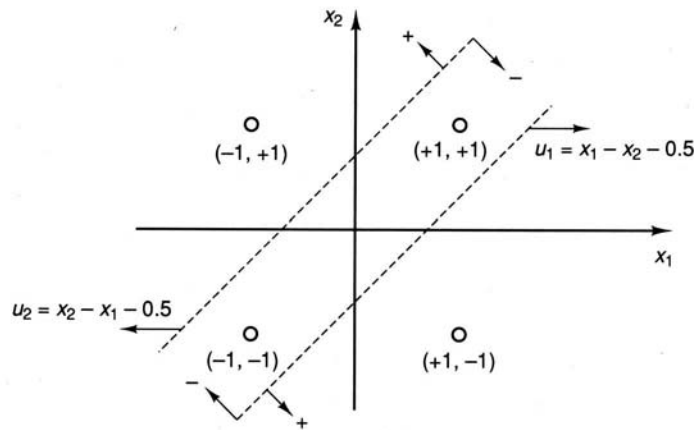


Figure 9.14 Input pattern space for a two-input, two-layer Madaline used to solve the exclusive-or problem.

Classical perceptron. A classical perceptron neuron is similar to the Adaline neuron. However, the activation function of a perceptron neuron can be linear or sigmoid. Also, in contrast to the Madalines of the 1960s in which the weight of the first layer was adaptive, but not that of the second layer, a perceptron network can have many layers that are all adaptive.

When the activation function is nonlinear, the input patterns are divided into different classes depending on the number of output neurons. By dividing the input pattern space into decision regions, all the input patterns resulting in a particular class can be determined. The decision regions are formed by the number of layers and neurons in the network [LIP 87]. For example, as shown in Figure 9.15a, a single-layer perceptron forms two decision regions separated by a hyperplane. The hyperplane divides the input pattern space into parts, one for which the output is zero and the other for which the output is 1. For the case of a

single neuron with two-inputs, the hyperplane separating the two regions is a line. In general, similar to Adaline, the single-layer perceptron can represent the functions that are linearly separable. But if the function is linearly inseparable, it cannot be realized by a one-layer perceptron and thus requires more than one layer.

A two-layer perceptron is able to form simple decision regions of the type shown in Figure 9.15b. Each node in the first layer forms two decision regions separated by a hyperplane. The nodes in the second layer take the intersection of these regions to form (open or closed) convex regions.

A three-layer perceptron can form arbitrary complex decision regions (See Figure 9.15c). Similar to the two-layer perceptron, each node in the second layer indicates whether the input pattern lies in a particular region. Each node in the third layer merges several of these regions in order to construct a bigger region. In general, the layers in a perceptron network can be viewed as the levels of a clustering tree. At each level of a clustering tree, each node represents a class of input patterns in which every input belongs to at least one of its children nodes.

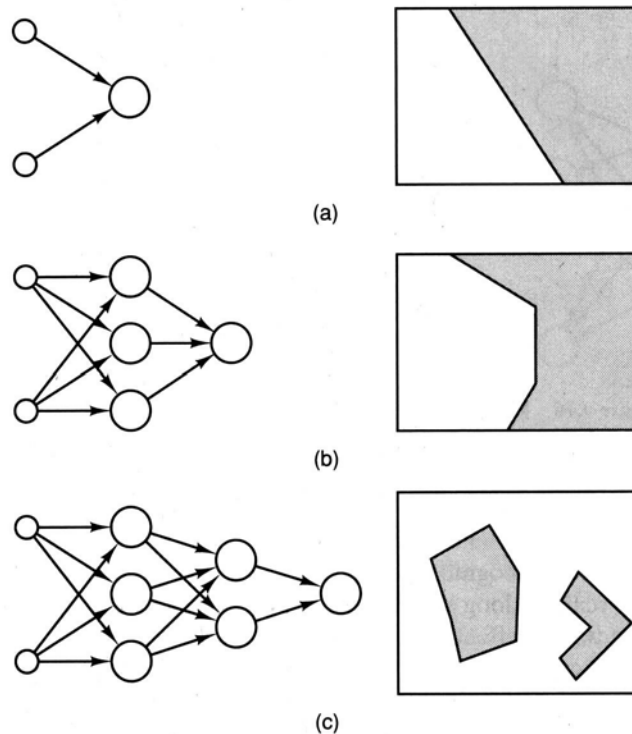


Figure 9.15 Different types of decision regions.

Training Multilayer Perceptrons. Multilayer perceptrons are enhanced versions of single-layer perceptrons. As shown in Figure 9.16, these models have several layers of perceptrons, including one input layer, one output layer, and several *hidden* layers. The outputs of one layer are connected to the inputs of the next layer. To increase the representation capability of these networks, a sigmoid activation function and continuous-valued inputs are often used.

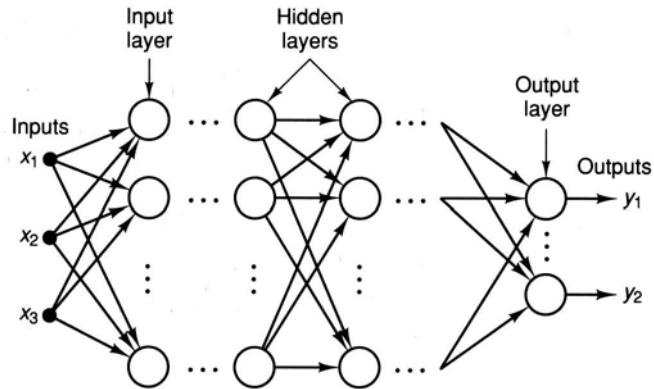


Figure 9.16 Multilayer perceptron.

Kolmogorov's mapping neural network existence theorem [NIE 90] says that any continuous function $f: [0,1]^n \rightarrow R^m$ (R is set of real numbers) can be represented by a two-layer perceptron having n inputs, $2n+1$ neurons in the input layer, and m neurons in the output layer. However, it has still not been shown that the multilayer networks can learn all the functions that they can represent, but in most problems of current interest (like pattern recognition) it has been shown to be possible.

Although it was realized long ago that multilayer perceptrons could realize most of the functions, there was no effective training algorithm for such networks. The reason for this was the difficulty associated with obtaining desired weights for the inputs of the hidden layers. However, in the 1980s this difficulty seemed to be solved as researchers explored new training algorithms. One of these algorithms, which became very popular, is the *back-propagation training algorithm*. This algorithm has been used in many applications and most of the time has produced good results.

Back-propagation Training Rule. Until the mid 1980s there was no proper algorithm or training rule to train multilayer perceptrons. It is very easy to adapt the neurons in the output layer, since the actual response of the network can be compared with the target response for each training input vector. The difficulty lies in changing the weights associated with the neurons in other layers since they do not have a target response. The back-propagation algorithm overcomes this difficulty by propagating the error back through the network.

The back-propagation algorithm was first reported by Werbos [WER 74], then by Parker [PAR 85], and finally by Rumelhart, Hinton, and Williams [RUM 86b]. It is an iterative algorithm in which the actual output gets closer to the target output after each iteration. Each iteration consists of two main passes, *forward pass* and *reverse pass*. The steps in each pass are as follows:

FORWARD PASS

1. Apply an input pattern to the network.
2. Based on the current weights of the network, compute the actual output of the network.

REVERSE PASS

3. Compute the error between the actual output value and the target output value. (Often the mean square error is considered to be the error.)
4. Adjust the weights of the network in such a way that the new weights cause a reduction of the error.

At every iteration of the algorithm, these four steps are repeated for each input pattern. Iterations are repeated until the error for each input pattern becomes less than an acceptable value. In the fourth step (which is the heart of algorithm), the weights are adjusted by propagating the error *backward* in the network. That is, the weights are adjusted layer by layer, starting from the output layer and going toward the first layer. The weights of the output layer are modified similarly to the method used for training an

Adaline; that is,

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

where Δw_{ij} amount of change, computed as

$$\Delta w_{ij} = \rho \delta_j y_i$$

w_{ij} = weight from neuron i (in hidden layer) to neuron j (in the output layer).
 y_i = output value of neuron i .
 δ_j = error for neuron j in the output layer, computed as $\delta_j = y_j(1 - y_j)(t_j - y_j)$.
 ρ = learning rate.
 y_j = output value of the output neuron j .
 t_j = target value for the output neuron j .

Adjusting the weights in the hidden layers involves a little more computation than for the output layer. The only difference is in the computation of the neuron error. In this case the error for a particular hidden layer l is evaluated as

$$\delta_j = y_j(1 - y_j) \left(\sum_{\substack{m \\ \text{neurons in} \\ \text{layer } l+1}} \delta_m w_{jm} \right).$$

For each hidden layer, the error δ is calculated in the same way. The errors of one layer are propagated to the preceding layer. This process continues until the error propagates through the first layer.

The amount of change in the weights of each hidden layer is computed similarly to the output layer case, that is,

$$\Delta w_{ij} = \rho \delta_j y_i.$$

However, for the first layer, the output y_i should be substituted by x_i :

$$\Delta w_{ij} = \rho \delta_j x_i.$$

In the preceding algorithm, the method used to update the connections is called *on-line* updating. In on-line updating the weight changes are applied after each input pattern is presented. Another option, called *batch* updating, is also used for changing the weights. In batch updating the weight changes are accumulated and applied after *all* patterns have been run through the network. The effect of batch updating is to average the weight changes over the whole set of patterns, thus achieving a smoother movement in weight space. It should be noted that the learning rule for the back-propagation algorithm is derived under the assumption that batch updating is used. However, in many cases it is not practical to provide storage for the accumulation of the weight changes until the whole set is processed; especially if specialized VLSI hardware is used, it is much more convenient to use on-line updating instead.

Neither of these two update modes is superior to the other in terms of speed of convergence; generally, approximately the same number of iterations is needed to train the network in the two modes. This does not mean that the weight change dynamics are the same; it can happen that in one mode the learning process converges, while in the other mode it does not.

In the following, an example is given to clarify the steps of the back-propagation algorithm when using on-line updating.

XOR example. Consider the problem of implementing the XOR function. The activation function for each

neuron is the sigmoid function. The output of a sigmoid function can reach the values of 0 and 1 only when the weights are infinitely large. Therefore, it is better to expect outputs other than 0 and 1 from the network. As a result, the truth table has to be changed slightly for the training process. The modified truth table is given in Figure 9.17.

Inputs		Output
x_1	x_2	t
0.1	0.1	0.1
0.1	0.9	0.9
0.9	0.1	0.9
0.9	0.9	0.1

Figure 9.17 Modified truth table for XOR function.

A two-layer perceptron model to implement the XOR with three neurons is shown in Figure 9.18. The network has nine variable weights, as shown.

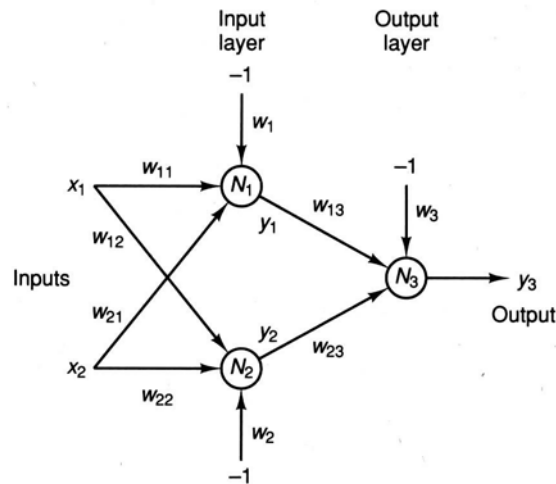


Figure 9.18 Perceptron model for XOR problem.

To start the training, the weights are assigned random numbers in the range $[-0.2, 0.2]$. The first input pattern is picked (i.e., $x_1=0.1, x_2=0.1$) and applied to the network. The learning coefficient is taken to be 0.5. Let the initial weights be:

$$\begin{aligned}
 w_{11} &= -0.13, & w_{21} &= -0.16, & w_1 &= 0.18, \\
 w_{12} &= 0.15, & w_{22} &= 0.02, & w_2 &= -0.18, \\
 w_{13} &= -0.09, & w_{23} &= 0.10, & w_3 &= 0.08.
 \end{aligned}$$

The output of neuron N_1 is

$$\begin{aligned}
 y_1 &= \text{sigmoid}[w_{11}x_1 + w_{21}x_2 + (-1.0)(w_1)] \\
 &= \text{sigmoid}[(-0.13)(0.1) + (-0.16)(0.1) + (-1.0)(0.18)] \\
 &= \text{sigmoid}[-2.090E-01] = 1/(1 + e^{2.090E-01}) = 4.479E-01.
 \end{aligned}$$

Similarly, the output of neuron N_2 is

$$\begin{aligned}
 y_2 &= \text{sigmoid}[w_{12}x_1 + w_{22}x_2 + (-1.0)(w_2)] \\
 &= \text{sigmoid}[(0.15)(0.1) + (0.02)(0.1) + (-1.0)(-0.18)] \\
 &= \text{sigmoid}[1.970E-01] = 1/(1 + e^{-1.970E-01}) = 5.491E-01.
 \end{aligned}$$

Finally, the output of the network is

$$y_3 = \text{sigmoid}[w_{13}y_1 + w_{23}y_2 + (-1.0)(w_3)]$$

$$\begin{aligned}
&= \text{sigmoid}[(-0.09)(4.479\text{E-}01) + (0.10)(5.491\text{E-}01) + (-1.0)(0.08)] \\
&= \text{sigmoid}[-6.541\text{E-}02] = 1/(1 + e^{6.541\text{E-}02}) = 4.837\text{E-}01.
\end{aligned}$$

This represents a simple calculation for the output of the network. Now the error signals can be calculated starting from the outermost layer. Neuron N_3 is an output neuron. The error signal for this neuron is

$$\delta_3 = y_3(1 - y_3)(t - y_3),$$

where y_3 is the actual output and t is the target output. Therefore,

$$\delta_3 = 4.837\text{E-}01(1.0 - 4.837\text{E-}01)(0.1 - 4.837\text{E-}01) = -9.581\text{E-}02.$$

Now, the weights of w_{13} , w_{23} , and w_3 can be updated. The new weights are

$$\begin{aligned}
w_{13} &= w_{13} + \Delta w_{13} = w_{13} + \rho y_1 \delta_3 \\
&= -0.09 + [0.5 * (4.479\text{E-}01)(-9.581\text{E-}02)] = -1.115\text{E-}01, \\
w_{23} &= w_{23} + \Delta w_{23} = w_{23} + \rho y_2 \delta_3 \\
&= 0.10 + [0.5 * (5.491\text{E-}01)(-9.581\text{E-}02)] = 7.370\text{E-}02, \\
w_3 &= w_3 + \Delta w_3 = w_3 + \rho(-1.0) \delta_3 \\
&= 0.08 + [0.5 * (-1.0)(-9.581\text{E-}02)] = 1.279\text{E-}01.
\end{aligned}$$

To calculate the weight changes for the hidden layer, the error must be propagated back toward the input. As such, the error signal for neuron N_1 becomes:

$$\begin{aligned}
\delta_1 &= y_1(1 - y_1)(\delta_3 w_{13}) \\
&= [(4.479\text{E-}01)(1.0 - 4.479\text{E-}01)][(-9.581\text{E-}02)(-1.115\text{E-}01)] \\
&= 2.641\text{E-}03.
\end{aligned}$$

The error δ_1 is used to update the weights coming from the inputs to the neuron N_1 . The new weights are:

$$\begin{aligned}
w_{11} &= w_{11} + \Delta w_{11} = w_{11} + \rho \delta_1 x_1 \\
&= -0.13 + [0.5 * (2.641\text{E-}03) * 0.1] = -1.299\text{E-}01, \\
w_{21} &= w_{21} + \Delta w_{21} = w_{21} + \rho \delta_1 x_2 \\
&= -0.16 + [0.5 * (2.641\text{E-}03) * 0.1] = -1.599\text{E-}01, \\
w_1 &= w_1 + \Delta w_1 = w_1 + \rho \delta_1(-1.0) \\
&= 0.18 + [0.5 * (2.641\text{E-}03)(-1.0)] = 1.787\text{E-}01.
\end{aligned}$$

The error signal for neuron N_2 becomes

$$\begin{aligned}
\delta_2 &= y_2(1 - y_2)(\delta_3 w_{23}) \\
&= [(5.491\text{E-}01)(1.0 - 5.491\text{E-}01)][(-9.581\text{E-}02)(7.370\text{E-}02)] \\
&= -1.748\text{E-}03.
\end{aligned}$$

Thus the weights w_{12} , w_{22} , and w_2 become

$$\begin{aligned}
w_{12} &= w_{12} + \Delta w_{12} = w_{12} + \rho \delta_2 x_1 \\
&= 0.15 + [0.5 * (-1.748\text{E-}03) * 0.1] = 1.499\text{E-}01, \\
w_{22} &= w_{22} + \Delta w_{22} = w_{22} + \rho \delta_2 x_2 \\
&= 0.02 + [0.5 * (-1.748\text{E-}03) * 0.1] = 1.991\text{E-}02, \\
w_2 &= w_2 + \Delta w_2 = w_2 + \rho \delta_2(-1.0) \\
&= -0.18 + [0.5 * (-1.748\text{E-}03)(-1.0)] = -1.791\text{E-}01.
\end{aligned}$$

Using this set of weights, a new output value for the network can be found. Performing the first pass of the back-propagation algorithm gives $y_3=4.657E-01$, which means that the new output is closer to the target output. In the next iteration the second pattern (i.e., $x_1=0.1$, $x_2=0.9$) is inputted to the network and the weights are updated with new error signals.

This procedure is repeated until the mean square error for each input vector becomes less than a small number (say 0.1). [The mean square error is defined as $(t-y_3)^2$.] The network is then said to be trained for the XOR function; that is, the final set of weights of the network will be such that it will give the correct response (output) for each input vector. One such set of weights is shown in Figure 9.19. However, this set of weights is not unique, and there could be many other solutions.

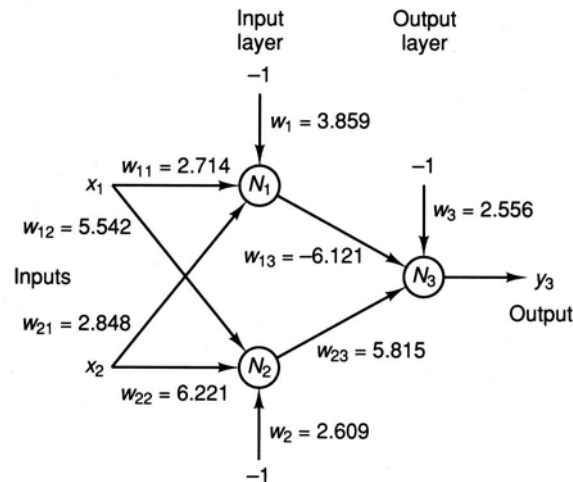


Figure 9.19 Final values of the weights in the XOR mode.

Comments on the back-propagation algorithm. Rumelhart and McClelland [RUM 86a] have expressed the following suggestions for the back-propagation rule:

1. When the initial weights are random and small, the network has a better chance to converge to an optimal solution without getting trapped in local minima.
2. Increasing the learning rate ρ will speed up the back-propagation algorithm. However, ρ should be kept small to avoid diverging oscillations.

The back-propagation algorithm follows the slope of the error surface downward, adjusting the weights until a minimum is reached. However, networks with hidden layers and nonlinear activation functions may have local minima in the error function, causing the algorithm to fail.

In the mid 1980s the back-propagation algorithm was very popular, but, it has recently lost this popularity. Kosko [KOS 92] has mentioned several reasons for this. One is that the back-propagation algorithm has failed to converge to a local minimum even when it was trained with nonlocal information. Also, White [WHI 89a, KOS 92] has shown that the back-propagation algorithm reduces to a special case of stochastic approximation and that there is nothing new about this algorithm. In fact, the back-propagation algorithm simply offers an efficient (parallel) way to implement the estimated gradient descent algorithm.

Another disadvantage of the back-propagation algorithm is that it requires supervision and a lengthy training period. It also requires synchronization between the neurons, which is hard to maintain.

Hopfield network. Hopfield has worked extensively in the field of recurrent neural networks [HOP 82, HOP 84, HOP 85, HOP 86]. Hence, the configurations that he worked with are now called *Hopfield networks*. Hopfield networks can be used to solve certain optimization problems or as an associative memory. Figure 9.20 shows the general structure of a Hopfield network. Notice that the output values of

the neurons are fed back to the inputs. In Hopfield's earlier work [HOP 82, HOP 84], each neuron i has a simple threshold activation function with a fixed threshold value T_i . The network changes state according to the following algorithm. Each neuron changes the value of its output according to the following rule:

$$y_i = \begin{cases} 1 & \text{if } \sum_{j=1}^n w_{ji} y_j + x_i > T_i \\ y_i & \text{if } \sum_{j=1}^n w_{ji} y_j + x_i = T_i \\ 0 & \text{if } \sum_{j=1}^n w_{ji} y_j + x_i < T_i \end{cases}$$

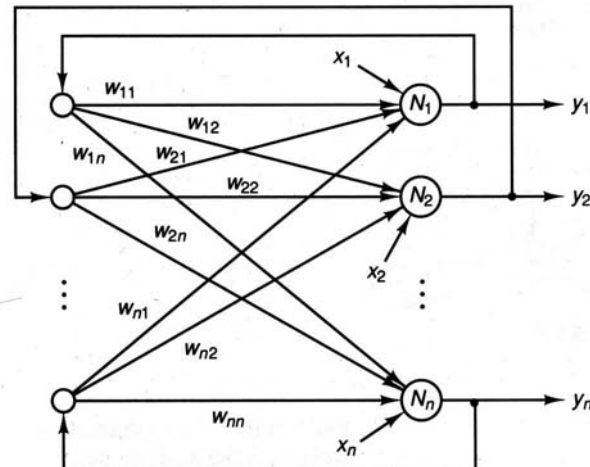


Figure 9.20 A Hopfield network.

Although each neuron randomly and asynchronously reevaluates its output, the algorithm requires that all neurons change states at the same average rate. Also notice that the algorithm does not have a learning law for adjusting the input weights. The weights are assumed to be determined in advance. For example, to use the Hopfield network as a content addressable memory, the following steps must be performed.

Step 1. Determine connection weights. To store a set of known patterns Y^1, Y^2, \dots, Y^m , the weights can be computed as

$$w_{ij} = \begin{cases} \sum_{s=1}^m (2 y_i^s - 1)(2 y_j^s - 1) & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

Step 2. Initialize the network's outputs with the unknown input pattern Y^s , that is, $y_i = y_i^s$ for $i=1$ to n .

Step 3. Calculate the new output of each neuron and feed it back to the network. Repeat this process until a stable state is obtained.

A stable state is said to be obtained when no more outputs change on successive iterations. The pattern specified by the output values in a stable state represents one of the stored known patterns that matches (or is close to) the unknown input pattern.

Each neuron has two states, 0 or 1. Therefore, a network with n neurons has 2^n distinct states. The network moves from one vertex of an n -dimensional hypercube to another till it reaches a stable state. Hopfield [HOP 82] and Cohen Grossberg [COH 83] proved that this type of network converges to a stable state when the weight matrix is symmetric and has zeros on its main diagonal; that is, $w_{ij}=w_{ji}$ and $w_{ii}=0$ for all i and j .

This convergent property can be proved by considering an energy function that never increases with consecutive iterations in the network. Eventually, this function reaches a local minimum, ensuring that the network is stable. *Lyapunov functions* have this property; one such function is defined as

$$E = -1/2 \sum_{i=1}^n \sum_{j=1}^n w_{ij} y_i y_j - \sum_{i=1}^n x_i y_i + \sum_{i=1}^n T_i y_i$$

where T_i denotes the threshold of neuron i .

To show that this function decreases with every iteration, let us consider the change in energy ΔE due to a change in some neuron k .

$$\Delta E = E - E'$$

where E is the present energy value and E' is the previous value.

$$\begin{aligned} \Delta E &= -1/2 \sum_{i=1}^n \sum_{j=1}^n w_{ij} y_i y_j - \sum_{i=1}^n x_i y_i + \sum_{i=1}^n T_i y_i - \\ &\quad (-1/2 \sum_{i=1}^n \sum_{j=1}^n w_{ij} y'_i y'_j - \sum_{i=1}^n x_i y'_i + \sum_{i=1}^n T_i y'_i). \end{aligned}$$

Since $y_i = y'_i$ for all $i \neq k$, and also $w_{ij} = w_{ji}$ and $w_{ii} = 0$, ΔE can be reduced to:

$$\begin{aligned} \Delta E &= -(y_k \sum_{j=1}^n w_{kj} y_j) - x_k y_k + T_k y_k + (y'_k \sum_{j=1}^n w_{kj} y'_j) + x_k y'_k - T_k y'_k \\ &= -(y_k - y'_k) \left(\sum_{j=1}^n w_{kj} y_j \right) + x_k - T_k \\ &= -\Delta y_k (u_k), \end{aligned}$$

where $\Delta y_k = y_k - y'_k$ and $u_k = \sum_{j=1}^n w_{kj} y_j + x_k - T_k$.

In the preceding expression, Δy_k and u_k always have the same sign; that is ΔE is always negative. When the output value of the neuron k changes from 0 to 1 ($y'_k = 0$ and $y_k = 1$), Δy_k and u_k both become positive. This is because, according to the rules for changing output values, y_k is 1 only when u_k is positive. On the other hand, when the neuron k changes from 1 to 0, Δy_k and u_k both become negative. Thus, any change in E is negative. Since E is bounded, the algorithm leads the network to a stable state that does not change further with time.

Based on the earlier work, Hopfield later represents a model in which inputs and outputs are continuous variables [HOP 84, HOP 86]. The activation function is a continuous and monotone-increasing function like a sigmoid function. The superiority of this model is that it has an electrical circuit implementation, as shown in Figure 9.21. The amplifiers in the circuit serve as the neurons. The resistors represent the weights and connect each neuron's output to the inputs of all the others. Ordinary positive-valued resistors can be used as weights in spite of the fact that the weights can be negative because the amplifiers have both inverting and noninverting outputs. Such a circuit with symmetric connections ($w_{ij} = w_{ji}$) converges to a stable state that is one of the local minima of the energy function:

$$E = -1/2 \sum_{i=1}^n \sum_{j=1}^n w_{ij} y_i y_j - \sum_{i=1}^n x_i y_i \quad (9.1)$$

where x_i is the i^{th} component of external input.

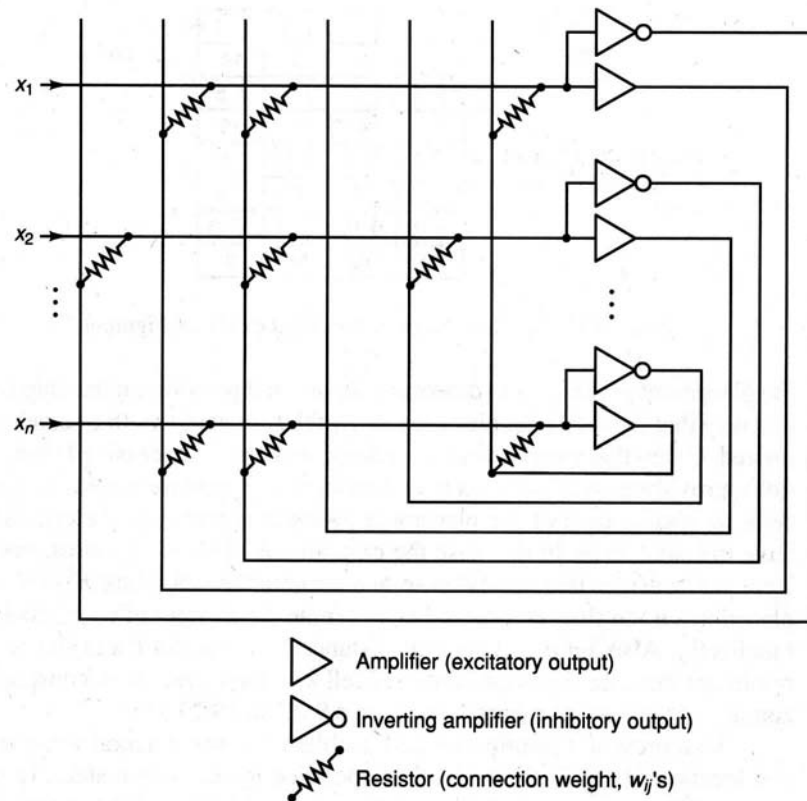


Figure 9.21 General structure of Hopfield's analog circuit.

Hopfield networks can be used to compute solutions to specific optimization problems. One problem to which Hopfield and Tank [HOP 86] applied their model is a classic optimization problem, the traveling salesman problem, (which is defined as finding the minimum distance of a valid tour of n cities starting from a given city, where a valid tour is defined as visiting each city exactly once). To map this problem onto the neural network, they have chosen a representation scheme in which the final location of any individual city is specified by the output states of a set of n neurons. Therefore, to represent a complete tour, a total of n^2 neurons, displayed as an $n \times n$ square array, has been used. To enable the n^2 neurons to represent a complete tour, the network is described by an energy function in which the lowest energy state (the most stable state of the network) corresponds to the best path. (See [HOP 86] for more detail.) Although, Hopfield networks can be used as special-purpose hardware for solving certain optimization problems, it turns out they may not even be able to provide local minimum solutions under certain conditions.

Example of a Hopfield Network Application. A Hopfield network can be used for finding a solution to the placement problem in VLSI design. The objective of the placement problem is to determine an optimal position on the chip for a set of cells in a way that the total occupied area and total estimated length of connections are minimized. Given that the main cause of delay in a chip is excessive length of the connections, providing shorter connections becomes an important objective in placing a set of cells. A special case of the placement problem is when all the cells are squares and have the same area. In this case the chip area is divided into slots, one for each cell. Here we consider this special case and assume that there are n^2 cells that should be placed in a $n \times n$ slots chip area. Let c_{ij} denote the number of connections between cell i and cell j . Also, let d_{km} denote the distance between slot k and slot m . The d_{km} is the minimum distance from the center of cell k to the center of m considering only horizontal and vertical segments (see Figures 9.22 and 9.23).

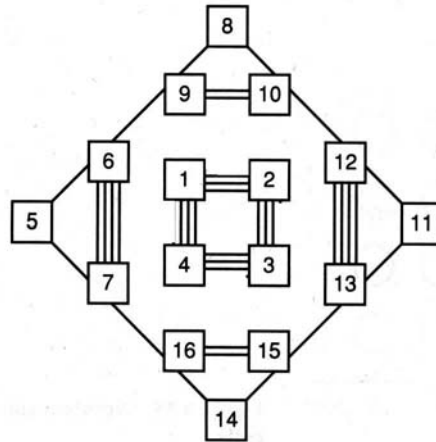


Figure 9.22 A set of cells with their interconnections.

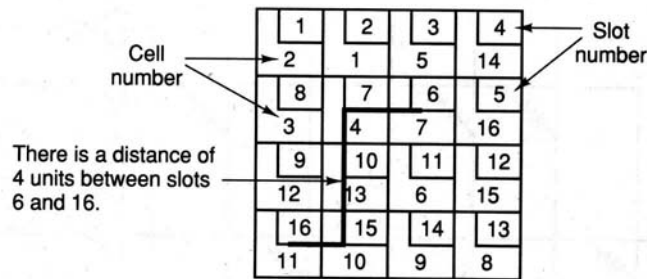
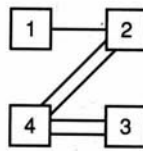
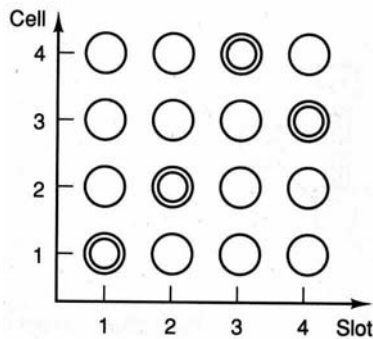


Figure 9.23 A placement solution for the cells of Figure 9.22.

To represent a solution for this problem, we use a neural structure in which the slot location of any individual cell is specified by the output states of a set of $n^2 \times n^2$ neurons. This structure is similar to the one used in [DAT 90]. As an example, let us consider the cells given in Figure 9.24a. Since the problem has four cells, it requires a total of 16 neurons. Figure 9.24b represents a neuron structure for this placement problem.



(a)



(b)

Figure 9.24 Neuron structure for four cells.

For example, the activated neuron at position (2, 2) (represented by a double circle) represents the assignment of cell 2 to slot 2. In general, an optimum solution must meet (satisfy) the following conditions:

1. Each cell is allocated to exactly one slot.
2. Each slot is assigned to exactly one cell.
3. Total wire length is minimized.

Conditions 1 and 2 mean that, in the array of neurons, exactly one neuron is high in each row (with the rest of them being low), and also exactly one neuron is high in each column (with all others being low). This can be written in terms of two energy functions, one specifying that the sum of neurons in a row (column) is 1 and the second specifying that the cross-product of neurons in a row (column) is 0. The last condition can be written in terms of an energy function that represents the total wire length. Hence

$$\begin{aligned}
 E = & \frac{A}{2} \sum_i \left[\left(\sum_j y_{ij} \right) - 1 \right]^2, \text{ sum of neurons in each row} = 1, \\
 & + \frac{A}{2} \sum_j \left[\left(\sum_i y_{ij} \right) - 1 \right]^2, \text{ sum of neurons in each column} = 1, \\
 & + \frac{B}{2} \sum_k \sum_i \sum_{j \neq i} y_{ki} y_{kj}, \text{ cross-product of neurons in a row} = 0, \\
 & + \frac{B}{2} \sum_i \sum_k \sum_{j \neq k} y_{ki} y_{ji}, \text{ cross-product of neurons in a column} = 0, \\
 & + \frac{D}{2} \sum_i \sum_k y_{ik} \left(\sum_{j \neq i} \sum_{m \neq k} y_{jm} c_{ij} d_{km} \right), \text{ minimize wire length,} \quad (9.2)
 \end{aligned}$$

where y_{ij} output of the neuron in row i and column j ,
 c_{ij} weight of the connection between cell i and cell j ,
 d_{km} distance between the slot k and slot m ,
 A, B, D weights for each energy function.

(Note: All the summations are to n^2 , where n^2 is the number of cells.)

As mentioned before the general energy function for the stable state with a local minimum is

$$E = -1/2 \sum_i \sum_j w_{ij} y_i y_j - \sum_i x_i y_i$$

In this equation the neurons are labeled in a linear fashion. To address the neurons in a two-dimensional space, the equation can be represented as

$$E = -1/2 \sum_{i=1}^{n^2} \sum_{j=1}^{n^2} \sum_{k=1}^{n^2} \sum_{l=1}^{n^2} w_{ij,kl} y_{ij} y_{kl} - \sum_{i=1}^{n^2} \sum_{j=1}^{n^2} x_{ij} y_{ij} \quad (9.3)$$

Equations (9.2) and (9.3) can now be equated to find the weights $w_{ij,kl}$ and the bias x_{ij} . By comparing the coefficients of each term in these equations, the values of all w 's and x 's can be defined as follows:

$$\begin{aligned}
 w_{ij,km} = & -2A, & \text{if } i=k \text{ and } j=m \\
 & = -(A+B), & \text{if } i=k \text{ and } j \neq m \\
 & = -(A+B), & \text{if } i \neq k \text{ and } j=m \\
 & = -Dc_{ik}d_{jm} & \text{otherwise,} \\
 x_{ij} = & 4A, & \text{for every } i \text{ and every } j.
 \end{aligned}$$

In practice, to simulate the preceding network on a uniprocessor, the following steps were implemented:

1. Initialize the entries of the weighting matrix W , the vector X , and vector U . Set the interval Δt and the initial maximum number of iteration.
2. For $j=1$ to (maximum number of iterations)
 - {
 - For $i=1$ to N -- N denotes number of neurons.
 - $$u_i = u_i + \left(\sum_{i=1}^N \sum_{j=1}^N w_{ij} y_j + x_i \right) \Delta t$$
 - For $i=1$ to N
 - $$y_i = 1 / 2 [1 + \tanh(u_i / \alpha)]$$
 - }

To run this code for the placement problem, which was presented in Figure 9.22, the initial values $A=1000$, $B=200$, $D=40$, $\alpha=0.05$, and $\Delta t=0.0001$ were used. Also, initially, random numbers between 0.48 and 0.52 were assigned to every u_i . After 149 iterations, the code was able to produce an optimum solution with the total wire length of 42 (see Figure 9.23).

After running several examples, it was determined that for most small-sized problems a Hopfield network is able to obtain good solutions. However, for larger problems, apart from the known problems of long simulation times, the code is not able to find a solution in many cases.

9.2.3 Implementation of ANNs

In recent years, several neural chips have been developed. However, often neural network models are simulated by software. Whenever an ANN is simulated by software, it is flexible, but it is slow. Therefore, the most promising approach for implementing an ANN is through hardware implementation. In fact, one main reason that ANNs are becoming popular is that they can be realized in a VLSI chip using current technology.

In general, three different technologies are available for hardware implementation of an ANN: *electronic*, *optical*, and *electro-optical*. Electronic technology itself can be divided into three different implementations: *analog*, *digital*, and *hybrid*.

In an analog implementation, the quantities can take a value within a fixed range (for example, between 0 and 1). Although this type of implementation reduces the design complexity, it is less accurate and often is unable to obtain an accuracy level of 6 bits. (This is mainly due to the low level of the accuracy of resistors). Many applications require an accuracy level of more than 6 bits.

In contrast to analog designs, the quantities in digital implementation take digital values. The advantage of having digital values is that they provide greater accuracy than analog designs do. However, they often require more area on the chip. A hybrid implementation contains digital and analog elements in order to gain the advantages of both designs.

The connectivity between neurons poses serious problems for electronic circuits because of the delay and space on the chip. Optical technology promises a way to solve this problem. By interconnecting neurons with light beams, no insulation is required between signal paths since light rays can pass through each other without interacting. Also, the signal paths can be made in three dimensions. In addition, all signal paths can be operating simultaneously, which provides a tremendous data rate. Finally, the weights can be stored as holograms. Although optical technology offers an ideal solution in theory, many practical problems are associated with it, the most pressing of which is that the physical characteristics of the optical devices are not compatible with the requirements of neural networks.

In an electro-optical implementation, the interconnections are made optically. Since ANNs are highly interconnected, this method becomes an attractive implementation alternative.

Among the preceding methods, the electronic is currently the most practical. In particular, digital electronics has advanced significantly in recent years. Given that the current state of digital technology is the result of research and development investments of hundreds of billions of dollars over several decades, it would be reasonable to assume that the same level of development for the other two technologies is unlikely to happen in the near future.

In summary, ANNs may not be able to accomplish the wide variety of tasks that researchers have projected. Perhaps, in the near future, they will become more available as special-purpose devices for pattern recognition and home appliances.

9.3 MULTIPLE-VALUED LOGIC

For many years, researchers have questioned the use of the binary system in today's computers. They argue that it does not fully utilize interconnection wires between components, despite the fact that wires realize a large part of any computer system. Interconnections comprise a major part (about 70%) of any VLSI chip. However, by letting each wire carry more than two levels of logic, a significant savings in chip area can be achieved.

The multiple-valued logic circuits allow signals with more than two values. In general, for an r -valued system (radix r), the values can be labeled as $0, 1, 2, \dots$, and $r-1$. For example, in a four-valued system (also called quaternary) the possible values are 0, 1, 2, and 3. Usually, the radix r is chosen to be a power of 2, such as 4, 8, and 16. This choice makes the conversion between binary-valued logic and multiple-valued logic very easy and efficient. Given the fact that binary-valued logic currently dominates the design of digital systems and that at present multiple-valued logic can only be used as a subpart of a system, code conversions between binary- and multiple-valued signals are required. Figure 9.25 represents how a multiple-valued circuit can be embedded in a system with binary components. The encoder circuit converts binary inputs to multiple-valued outputs. The decoder circuit converts multiple-valued inputs to binary outputs.

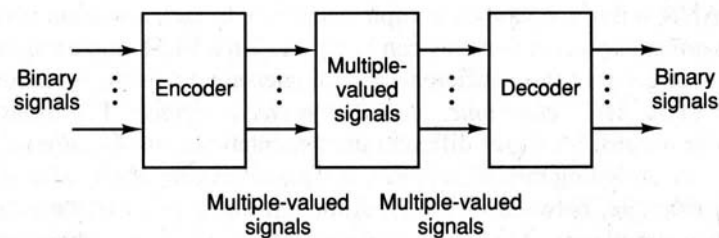


Figure 9.25 Use of a multiple-valued logic circuit as a component in a binary-valued system.

Often binary design techniques (such as a truth table) are used in designing the multiple-valued circuits. The only major difference is that more values must be considered in a multiple-valued design. For example, the truth table for the two-variable, four-valued half-adder shown in Figure 9.26 has 16 rows [SMI 88]. The half-adder adds two-input signals (denoted as A and B), producing a sum signal (denoted as S) and a carry signal (denoted as C).

Input		Output	
<i>A</i>	<i>B</i>	<i>C</i>	<i>S</i>
0	0	0	0
0	1	0	1
0	2	0	2
0	3	0	3
1	0	0	1
1	1	0	2
1	2	0	3
1	3	1	0
2	0	0	2
2	1	0	3
2	2	1	0
2	3	1	1
3	0	0	3
3	1	1	0
3	2	1	1
3	3	1	2

Figure 9.26 Truth table of a half adder for four-valued inputs *A* and *B*.

Based on the available technology, many designs have been presented for the implementation of multiple-valued circuits. These designs are based on MOS technology, CMOS technology, emitter-coupled logic (ECL), integrated injection logic, (I^2L), and charge-coupled device (CCD) technology. One proposed method for implementing a multiple-valued function is based on the use of a universal building block called the T-gate [KAM 87]. As shown in Figure 9.27a, the T-gate is actually a quaternary four-input multiplexer. The function of a T-gate can be defined as

$$Z = x_i, \text{ if } s = i,$$

where *s* is a four-valued selecting input signal that takes on the values 0, 1, 2, and 3. For *s*=0, input x_0 is selected; for *s*=1, input x_1 is selected, and so on. Each input x_i is also a four-valued signal with values 0, 1, 2, and 3. Figure 9.27b presents a more detailed schematic of a T-gate. In this figure, the pass transistors are used as switches for connecting an input signal to the output signal. An input signal appears at the output if the gate voltage of the corresponding pass transistor becomes V_{DD} (i.e., high). The gate voltage of each pass transistor is controlled by a gate, called a *literal gate*. The output voltages of literal gates *a*, *b*, *c*, and *d* become V_{DD} if the logical value *s* is 0, 1, 2, and 3, respectively. Otherwise, these voltages are zero. The logical values 0, 1, 2, and 3 of the select line correspond to voltages 0, 2, 4, and 6 volts, respectively. Each literal gate consists of a few transistors (two to three NMOS transistors), and its output increases based on a certain input voltage. For example, when signal *s* is at 4 volts, the output voltage of literal gate *c* becomes high, while the output of other literal gates remains low. (This is done by employing transistors with different threshold voltages in different literal gates; for more detail, see [KAM 87].)

The T-gate can be used to design any combinational or sequential circuit. For example, Figure 9.28 presents a block diagram for the sum output(s) of a half-adder with four-valued inputs *A* and *B*. The implementation follows the half-adder truth table. Note that the values for *s* in the truth table are given as the inputs to the T-gates T_1 , T_2 , T_3 , and T_4 . One of these inputs appears on the output *s*, depending on the values of *A* and *B*. Although the T-gate provides a structural and generic tool for designing multiple-valued circuits, it often does not lead to an efficient and minimizing design.

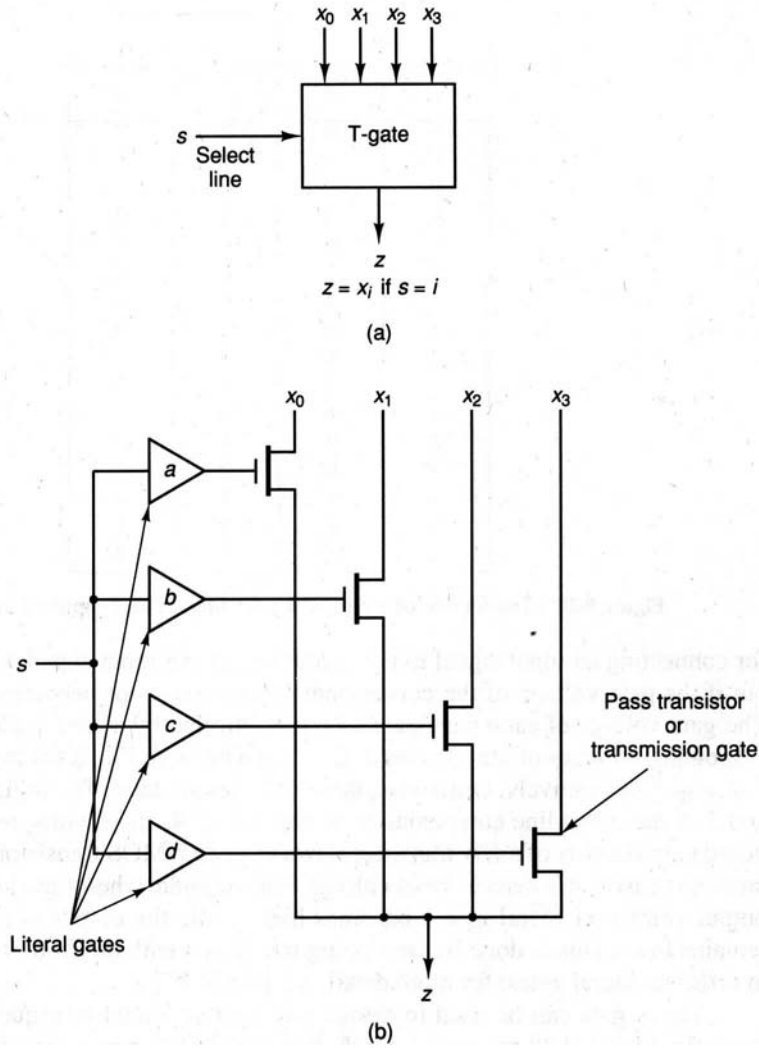


Figure 9.27 Block and circuit diagram of a T-gate.

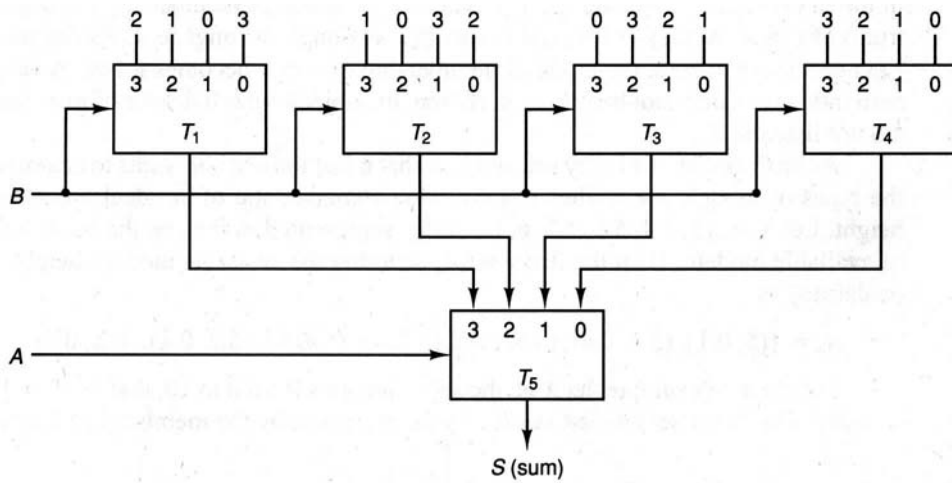


Figure 9.28 Block diagram for the sum output of a half-adder using T-gates.

In summary, multiple-valued logic may provide a good solution for certain applications, such as memory

design, for which it is desirable to reduce the number of lines for parallel transmission of large amounts of data [SMI 88]. In general, multiple-valued logic leads to a reduction in the number of pins. It also reduces the interconnection complexity and increases the data-processing capability per unit area of a chip [KAM 88]. However, multiple-valued logic should not be considered as a competitor to binary-valued logic. In fact, multiple-valued circuits should be used as sub-circuits in the binary-valued world.

9.4 FUZZY LOGIC

The basic idea underlying fuzzy logic was suggested by Zadeh [ZAD 65, ZAD 68, ZAD 72]. Zadeh proposed this logic to enhance the use of artificial intelligence (AI) techniques in certain areas such as speech recognition. An assumption is made that the reader is familiar with artificial intelligence theory. However, a brief discussion is given next to show the relationships between traditional AI and fuzzy logic.

There are generally two groups within the study of AI. Those who believe AI should be based on heuristic techniques, and those who believe AI should be based on classical two-valued logic (true or false). Heuristics can best be described as a rule of thumb to guide one's action. That is, we wish to attain a goal, and a number of possible actions are available at one point in time. A Heuristic technique is used to decide which is the best action to achieve this goal. A game of chess is an excellent example. The goal is to checkmate the opponent. At any point in the game, one or more possible moves could be taken. A heuristic is used to decide which move is best to achieve the goal.

The classical two-valued logic represents the meaning of a proposition as true or false. It is able to combine simple propositions through the use of connectives (such as “and,” “or,” and “not”), into more complex ones. For example, using the “and” connective, the following two propositions

All doctors have a college education.

Brian is a doctor.

can be combined as

*All doctors have a college education **and** Brian is a doctor.*

Whether this new proposition is true or false depends not only on the truth of each simple proposition, but also on the connective “and.” For if we change the connective to “or,” we may have completely different results. Several propositions may be used to perform reasoning. A simple example of reasoning can be

*All doctors have a college education **and** Brian is a doctor.*

Does Brian have a college education?

The answer is, of course, *true*.

Zadeh believes that we need logic in AI, but the kind of logic we need is not classical logic; it is fuzzy logic. This is because classical logic cannot represent a proposition with imprecise meaning. However, in fuzzy logic, which may be viewed as an extension of multiple-valued logic, a proposition may be true or false or have an intermediate value (such as very true). For example, classical two-valued logic cannot address the following questions, but fuzzy logic can:

1. *Most of those who are doctors have high incomes.*
Brian is a doctor. Is it true to say that he has a high income?
(Answer: may be.)
2. *Tomoko is much nicer than most of her friends.*
How nice is Tomoko?
(Answer: very nice.)

In general, fuzzy logic is concerned with formal principles of approximate reasoning, while classical two-valued logic is concerned with formal principles of reasoning [ZAD 88]. Classical two-valued logic

considers classes that have sharp boundaries, such as male or female, single or married, and boy or girl. In this way, an object is either a member of a class or not a member of a class. In contrast, fuzzy logic considers classes that do not have sharp boundaries, such as tall, short, nice, and intelligent. Here, a degree indicates the grade of membership of an object to a class. Usually, we have degrees between 0 and 1. For example, we can say Steve is tall to the degree 0.8.

9.4.1 Fuzzy Sets

Let X be a collection of objects denoted generically by x ; that is, $X = \{x\}$. A fuzzy set A in X is a set of ordered pairs:

$$A = \{(x, f_A(x)) / x \in X\},$$

where $f_A(x)$ is the *membership function* that associates with each $x \in X$ a real number in the interval $[0, 1]$. The value $f_A(x)$ indicates the grade of membership (or degree of truth) of x in A . When $f_A(x) = 1$, it means that x strongly belongs to A . As the value of $f_A(x)$ gets close to zero, the grade of membership of x in A becomes lower. A value of zero indicates x does not belong to A . (Often, the objects with 0 degree of membership are not listed in A .)

As an example of a fuzzy set, suppose that a fashion dresser wants to characterize the types of models she wishes to have. One characteristic of an ideal model is her height. Let $X = \{5, 5.4, 5.6, 5.7, 6, 6.2, 6.5\}$, represented in feet, be the set of heights of available models. Then the fuzzy set A , denoting the desirable model's height, may be defined as

$$A = \{(5, 0.1), (5.4, 0.4), (5.6, 0.8), (5.7, 1), (6, 0.6), (6.2, 0.4), (6.5, 0.1)\}.$$

For the next example, let X be the set of integers from 0 to 10, that is, $X = \{0, 1, \dots, 10\}$. The fuzzy set labeled *small* may be expressed by the membership function

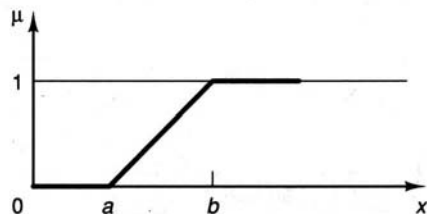
$$f_{small}(x) = \left[1 + \left(\frac{x}{2} \right)^4 \right]^{-1}$$

That is, $small = \{(0, 1), (1, 0.94), (2, 0.5), (3, 0.16), \dots, (10, 0.001)\}$

The membership functions should be defined so that they model precisely observed values in the real-world. However, in practice, it is difficult to derive membership functions with such characteristics. In practice, often membership functions are defined based on the data collected from past experiences and a set of well-shaped functions. Some of the commonly used membership functions are

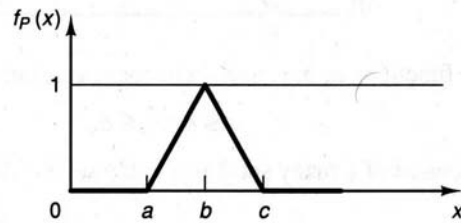
Linear function

$$f_L(x) = \begin{cases} 0 & x \leq a \\ (x-a)/(b-a) & a \leq x \leq b \\ 1 & x \geq b \end{cases}$$



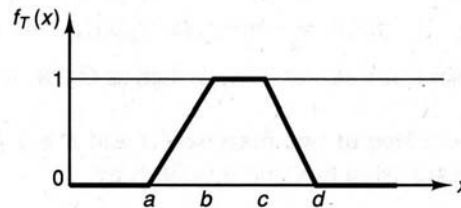
Piecewise linear (or triangular) function

$$f_P(x) = \begin{cases} 0 & x \leq a \\ (x-a)/(b-a) & a \leq x \leq b \\ (c-x)/(c-b) & b \leq x \leq c \\ 0 & x \geq c \end{cases}$$



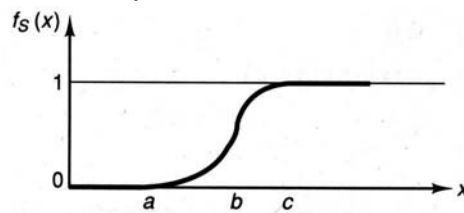
Trapezoidal function

$$f_T(x) = \begin{cases} 0 & x \leq a \\ (x-a)/(b-a) & a \leq x \leq b \\ 1 & b \leq x \leq c \\ (d-x)/(d-c) & c \leq x \leq d \\ 0 & x \geq d \end{cases}$$



S-function

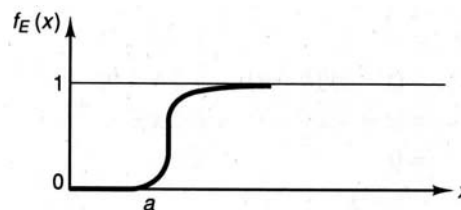
$$f_S(x) = \begin{cases} 0 & x \leq a \\ 2[(x-a)/(c-a)]^2 & a \leq x \leq b \\ 1 - 2[(x-c)/(c-a)]^2 & b \leq x \leq c \\ 1 & x \geq c \end{cases}$$



Exponential function

$$f_E(x) = \begin{cases} 0 & x \leq a \\ 1 - e^{-k(x-a)^2} & a \leq x \end{cases}$$

for some $k > 0$



In the preceding functions a , b , c , and d are some constants, where $a \leq b \leq c \leq d$.

The *complement* of a fuzzy set A is a fuzzy set \bar{A} whose membership function is defined by

$$f_{\bar{A}}(x) = 1 - f_A(x), \quad x \in X .$$

The *union* of two fuzzy sets A and B is a fuzzy set C , written as $(C = A \cup B)$, whose membership function is defined by

$$f_C(x) = \max \{f_A(x), f_B(x)\}, \quad x \in X .$$

The *intersection* of two fuzzy sets A and B is a fuzzy set C , written as $C = A \cap B$, whose membership function is defined by

$$f_C(x) = \min \{f_A(x), f_B(x)\}, \quad x \in X .$$

The fuzzy set C is a *subset* of A , written as $C \subseteq A$, if and only if $f_C(x) \leq f_A(x)$ for all x in X .

The *algebraic sum* of two fuzzy sets A and B is a fuzzy set C , written as $C = A \oplus B$, whose membership function is defined by

$$f_C(x) = f_A(x) + f_B(x) - f_A(x) * f_B(x), \quad x \in X .$$

The *algebraic product* of two fuzzy sets A and B is a fuzzy set C , written as $C = A * B$, whose membership function is defined by

$$f_C(x) = f_A(x) * f_B(x), \quad x \in X .$$

The *support* of a fuzzy set A is a set $S(A)$ such that for every $x \in S(A) : f_A(x) > 0$.

For example, let $X = \{5, 5.4, 5.6, 5.7, 6, 6.2, 6.5\}$, $A = \{(5.4, 0.4), (5.7, 1), (6, 0.6), (6.2, 0.4)\}$, and $B = \{(5.4, 0.3), (6.5, 1), (6, 0.5)\}$.

$$\begin{aligned} \bar{A} &= \{(5, 1), (5.4, 0.6), (5.6, 1), (6, 0.4), (6.2, 0.6), (6.5, 1)\} \\ A \cup B &= \{(5.4, 0.4), (6.5, 1), (5.7, 1), (6, 0.6), (6.2, 0.4)\} \\ A \cap B &= \{(5.4, 0.3), (6, 0.5)\} \\ C \subseteq A &= \{(5.4, 0.3), (6, 0.4)\} \\ A * B &= \{(5.4, 0.12), (6, 0.3)\} \\ A^2 &= \{(5.4, 0.16), (5.7, 1), (6, 0.36), (6.2, 0.16)\} \end{aligned}$$

Fuzzy relation. A fuzzy relation is a fuzzy set defined on the Cartesian product of crisp sets X_1, X_2, \dots, X_n , where tuples (x_1, x_2, \dots, x_n) may have varying degree of membership within the relation [KLI 88]. When $n=2$, the relation is called a binary relation. For example, we can define the fuzzy binary relation “very far” on given sets $X = \{\text{New Delhi, Tokyo}\}$ and $Y = \{\text{New York, Taipei}\}$ as follows, using real numbers between 0 and 1 as a degree of membership. A degree 1 means that the cities are very far from each other.

$$R(X, Y) = \begin{matrix} & \begin{matrix} \text{New Delhi} & \text{Tokyo} \end{matrix} \\ \begin{matrix} \text{New York} \\ \text{Taipei} \end{matrix} & \begin{bmatrix} 0.9 & 1 \\ 0.3 & 0.1 \end{bmatrix} \end{matrix}$$

Two binary relations can be combined to produce a new binary relation. This process is called composition. Given two binary relations $P(X, Y)$ and $Q(Y, Z)$, their composition $R(X, Z)$ can be represented as

$$R(X, Z) = P(X, Y) \circ Q(Y, Z).$$

The relation $R(X, Z)$ is a subset of the Cartesian product of X and Z , where $(x, z) \in R$ if and only if there exists at least one $y \in Y$ such that $(x, y) \in P$ and $(y, z) \in Q$.

There are different ways for computing the composition of two relations. Among them, the most well known method is the max-min composition. Given $R(X, Z) = P(X, Y) \circ Q(Y, Z)$, the max-min composition can be thought of as the strength of the relational tie between elements of X and Z . In this type of composition, the membership degree for each tuple $(x, z) \in R$ is defined as

$$f_R(x, z) = \max_{y \in Y} \{ \min [f_P(x, y), f_Q(y, z)] \} \text{ for all } x \in X \text{ and } z \in Z.$$

As an example, let the binary relations $P(x, y)$ and $Q(y, z)$ be defined as follows:

$$P = \begin{matrix} & y_1 & y_2 \\ x_1 & \begin{bmatrix} 0.8 & 0.4 \end{bmatrix} \\ x_2 & \begin{bmatrix} 1 & 0.3 \end{bmatrix} \end{matrix}$$

$$Q = \begin{matrix} & z_1 & z_2 \\ y_1 & \begin{bmatrix} 0.5 & 0.7 \end{bmatrix} \\ y_2 & \begin{bmatrix} 0.1 & 0.9 \end{bmatrix} \end{matrix}$$

$$f_R(x_1, z_1) = \max \{ \min [f_P(x_1, y_1), f_Q(y_1, z_1)], \min [f_P(x_1, y_2), f_Q(y_2, z_1)] \} = 0.5$$

$$f_R(x_1, z_2) = \max \{ \min [f_P(x_1, y_1), f_Q(y_1, z_2)], \min [f_P(x_1, y_2), f_Q(y_2, z_2)] \} = 0.7$$

$$f_R(x_2, z_1) = \max \{ \min [f_P(x_2, y_1), f_Q(y_1, z_1)], \min [f_P(x_2, y_2), f_Q(y_2, z_1)] \} = 0.5$$

$$f_R(x_2, z_2) = \max \{ \min [f_P(x_2, y_1), f_Q(y_1, z_2)], \min [f_P(x_2, y_2), f_Q(y_2, z_2)] \} = 0.7$$

Thus, R can be represented as

$$R = \begin{matrix} & z_1 & z_2 \\ x_1 & \begin{bmatrix} 0.5 & 0.7 \end{bmatrix} \\ x_2 & \begin{bmatrix} 0.5 & 0.7 \end{bmatrix} \end{matrix}$$

9.4.2 Linguistic Variables and Fuzzy Rules

Two of the main concepts that play an important role in many applications of fuzzy logic are the concepts of *linguistic variable* and *fuzzy if-then rules* [ZAD 73, ZAD 88, ZAD 94]. Linguistic variables are the main concept in exploiting the tolerance for imprecision. A linguistic variable is a variable whose values are words or sentences in a language. For example, height is a linguistic variable when its values are defined to be *tall*, *medium*, and *short*. As shown in Figure 9.29, each of these linguistic values represents a possibility distribution for the height. Each linguistic value is represented as a fuzzy set that is characterized by a membership function. The set of the linguistic values of a linguistic variable is called a *term set*. For example, the term set for linguistic variable age $T(\text{age})$, may be defined as

$$T(\text{age}) = \{ \text{young, very young, not young, old, very old, not old, extremely old, middle-aged} \}.$$

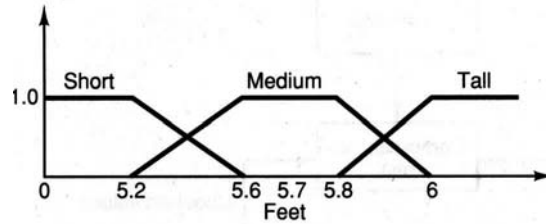


Figure 9.29 Membership functions for the linguistic values short, medium, and tall.

In general, a fuzzy rule can be represented as

if x_1 is A_1 and x_2 is A_2 and ... x_n is A_n then y_1 is B_1 and y_2 is B_2 and ... y_m is B_m ,

where $x_1, x_2, \dots, x_n, y_1, y_2, \dots, y_m$ are linguistic variables and $A_1, A_2, \dots, A_n, B_1, B_2, \dots, B_m$ are their respective linguistic values. For example,

if age is old and height is short then modeling-rate is not good.

9.4.3 Control System

Fuzzy logic has been applied to many applications, such as process control, image understanding, robotics, and expert systems. Fuzzy control is the first successful industrial application of fuzzy logic. A fuzzy controller is able to control systems that previously could only be controlled by skilled operators. Japan has achieved significant progress in this area and has applied it to variety of products, such as cruise control for cars, video cameras, rice cookers, and washing machines [SAN 91]. Researchers in Japan are now tackling the problem of integrating sophisticated human knowledge into a fuzzy framework [LIF 91]. After solving this problem based on fuzzy logic, they intend to make more intelligent and higher-speed computers. These computers will be able to process fuzzy logic at high speeds.

Fuzzy logic is very effective in nonlinear control processes because it models the experience of a human operator, rather than the process itself. In general, a fuzzy logic controller consists of four units: condition interface, rule base, computational unit, and action interface (see Figure 9.30). The condition interface observes the current state of the process and expresses that in terms of linguistic values. The rule base unit determines which rules are to be applied under which conditions. The computational unit performs the fuzzy computations. The action interface transforms the output control linguistic values into control action.

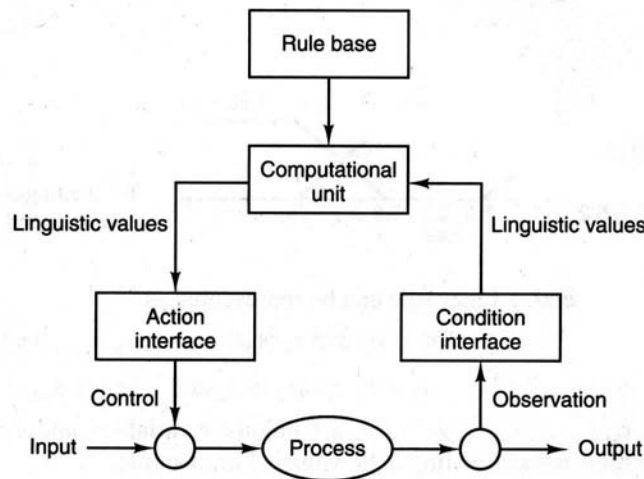


Figure 9.30 Structure of a fuzzy logic controller.

To represent how fuzzy logic can be used in a control system, an example for moving a robot toward a track is discussed. This example is based on a more complex example, which involves backing up a truck, discussed in [KOS 92]. In the truck example, the goal is to move a truck backward to a particular loading zone at a right angle. Here, we will consider a robot that moves forward toward a track, and once it is on

the track it moves toward the north direction. Figure 9.31 shows the robot and the track that the robot should go on. Figure 9.32 shows that the position of the robot is determined by two linguistic variables, the direction angle, denoted as α , and the distance from the center line of the track, denoted as x . The direction of the robot movement, denoted as β , is determined by the angle of the front wheel. For a given initial robot position within a specified area, the goal is to move the robot toward the center of the track. The desired final position is to let the robot move on the track toward the north.

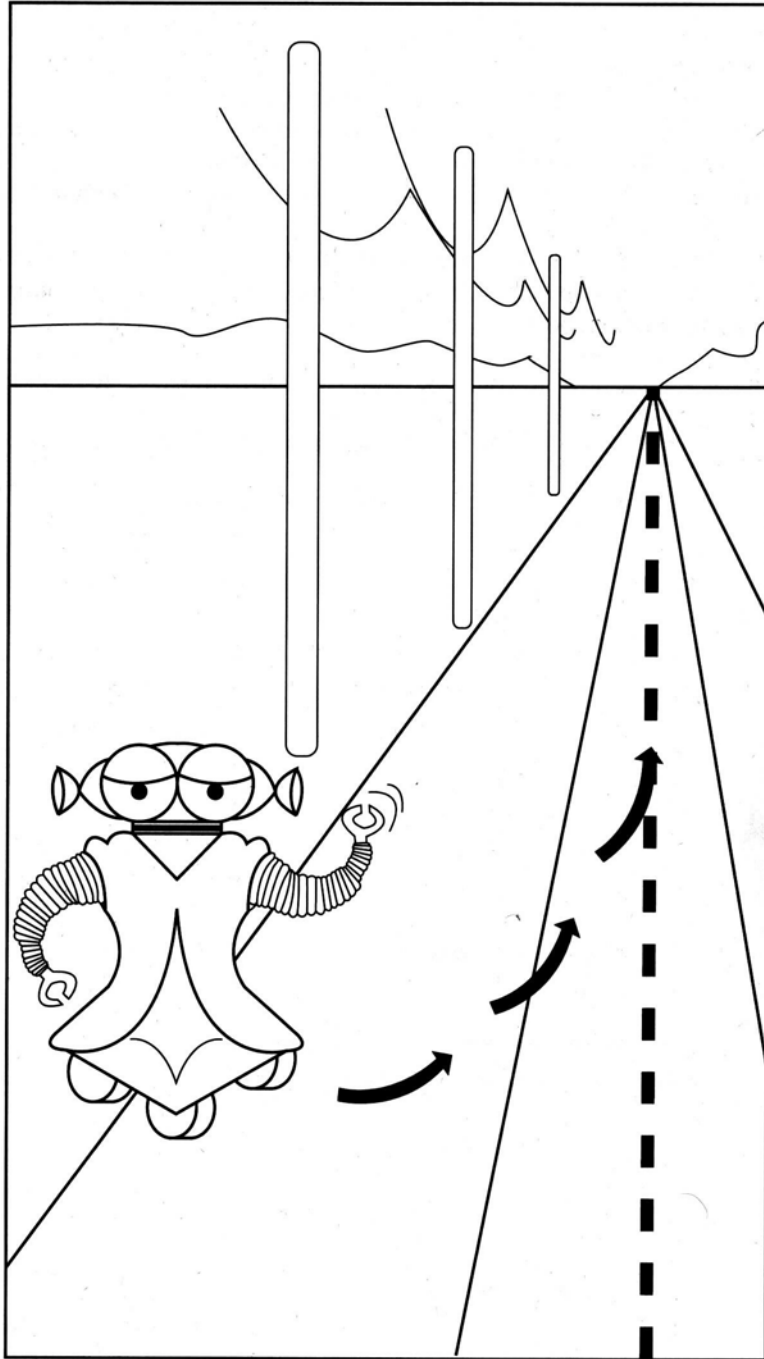


Figure 9.31 A robot and a track that the robot should go on.

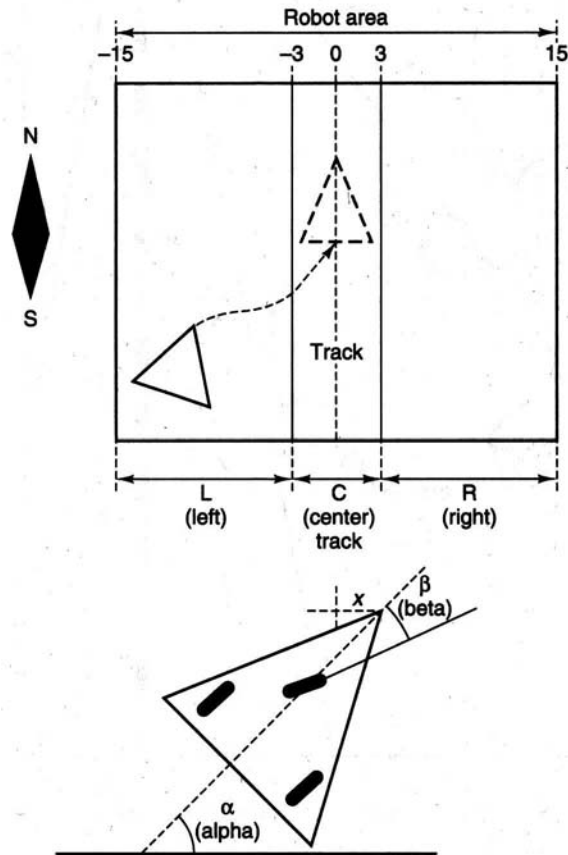


Figure 9.32 The measurements x and α are used to determine the position of the robot with respect to the center of track, and β is used to determine the angle of the front wheel.

Let's assume the ranges of the linguistic variables x , α , and β , are as follows:

$$\begin{array}{rclcl}
 -15 & \leq & x & \leq & 15, \\
 0 & \leq & \alpha & \leq & 360, \\
 -15 & \leq & \beta & \leq & 15.
 \end{array}$$

Notice that the rotation of the front wheel is limited to 15 degrees. Positive values of β represent a right turn of the front wheel, and negative values represent a left turn.

To each of these linguistic variables, a set of linguistic values are assigned as follows:

DISTANCE x		<i>(Input Variable)</i>
L	:	left side of the track
C	:	center of the track
R	:	right side of the track
DIRECTION ANGLE α		<i>(Input Variable)</i>
N	:	North
W	:	West
S	:	South
E	:	East

FRONT WHEEL ANGLE β (Output Variable)

TR : turn right
 ST : straight
 TL : turn left

As shown in Figure 9.33, a range of numerical values can be assigned to each linguistic value of a linguistic variable. In this figure, each graph, called a membership function, indicates the degree to which an input value belongs to a particular linguistic value. Such a degree of membership ranges from 0 to 1. The value 0 indicates no membership, and the value 1 represents full membership. A value between 0 and 1 represents partial membership. For example, $x = -10$ belongs to fuzzy value L with degree 1 and to C with degree 0. Similarly, $\alpha = 89^\circ$ belongs to N with degree 0.988 and to E with degree 0.01.

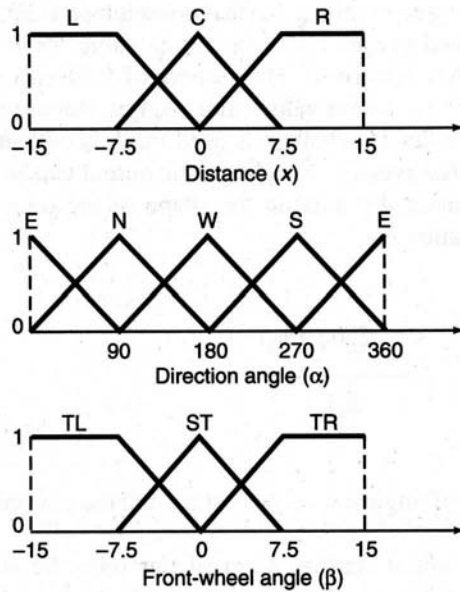


Figure 9.33 Membership functions for the distance, direction angle, and front-wheel angle.

Next, similar to an expert system, a set of rules must be defined. In general, each rule produces some output linguistic values based on some input linguistic values. For example, in the robot case, some of the rules can be defined as

if ($\alpha = S$ *and* $x = L$) *then* $\beta = TL$
if ($\alpha = S$ *and* $x = C$) *then* $\beta = TL$
if ($\alpha = S$ *and* $x = R$) *then* $\beta = TR$

These rules can be extended to consider all the possible values for α ; thus there will be 12 rules in all. Figure 9.34 represents all these rules, often called *fuzzy associative memory (FAM)* rules, in a table. The preceding three rules are shown in the first row of the table.

		Distance x		
		L	C	R
Direction α	S	TL	TL	TR
	E	ST	TL	TL
	N	TR	ST	TL
	W	TR	TR	ST

Figure 9.34 Set of rules to determine robot movement.

The linguistic values of the *if* part of a rule are referred to as *antecedents*, and the values of the *then* part are referred to as *consequents*. For example, in the rule

$$\text{if } (\alpha = \text{S and } x = \text{L}) \text{ then } \beta = \text{TL},$$

S and L are antecedents and TL is the consequent.

For given input values for x and α , the controller should determine an output value for β , the angle of the front wheel. First, for each input value the controller determines the membership degree of its corresponding linguistic values. Next, for each rule, the minimum of the membership degrees of its antecedents is chosen as a membership degree for the rule's consequent. This membership degree is considered as a weight for the rule's consequent. When there is more than one membership degree for a consequent, the maximum degree is chosen for that consequent. Hence, at this point, a membership degree is assigned to each linguistic output value. To compute the output value for β , defuzzification is performed. The purpose of defuzzification is to combine the effects of all the linguistic output values into a single output value.

Often the *centroid defuzzification* method is used for defuzzification [KOS 92]. This method provides a weighted average of all linguistic output values. The complexity of the centroid defuzzification depends on the shape of the output membership functions. A simplified calculation is

$$\frac{\sum_{i=1}^n (c_i * L_i)}{\sum_{i=1}^n L_i}$$

where the L_i 's are the weights of linguistic output values and the c_i 's are the weighting factors.

As an example, let the initial starting point of our robot be at $x=-10$ and $\alpha=89$. For these initial values, the membership degree of the linguistic input values are

$$\begin{array}{ll}
 x=-10 \rightarrow & \begin{array}{l} f_L(-10) = 1 \quad [f_L(-10) \text{ denotes the degree of L}] \\ f_C(-10) = 0 \\ f_R(-10) = 0, \end{array} \\
 \alpha=89 \rightarrow & \begin{array}{l} f_E(89) = 0.01 \\ f_N(89) = 0.988 \\ f_W(89) = 0 \\ f_S(89) = 0. \end{array}
 \end{array}$$

Now, for each rule we calculate a membership degree for its consequent. As shown next, the consequent's membership degree is the minimum of the membership degrees for α and x . This is because there is an *and* operation between α and x .

Rule No.	Input 1 α	Degree		Input 2 x	Degree	Output β	Degree (Minimum of degrees)
1	S	0	and	L	1	TL	0
2	E	0.01	and	L	1	ST	0.01
3	N	0.988	and	L	1	TR	0.988
4	W	0	and	L	1	TR	0
5	S	0	and	C	0	TL	0
.
.
.

Figure 9.35 represents a membership degree for each rule's consequent. Note that there are four degrees for the consequent TR. Among these degrees, the maximum degree, 0.988, is chosen for TR. In the same way, degrees 0.01 and 0 are chosen for ST and TL, respectively. Based on these degrees, the system output value can be evaluated as

$$\beta = \frac{(-15.0 * \text{MAX}(f_{TL}(\cdot))) + (0.0 * \text{MAX}(f_{ST}(\cdot))) + (15.0 * \text{MAX}(f_{TR}(\cdot)))}{\text{MAX}(f_{TL}(\cdot)) + \text{MAX}(f_{ST}(\cdot)) + \text{MAX}(f_{TR}(\cdot))}$$

$$= \frac{(-15.0 * 0.0) + (0.0 * 0.01) + (15.0 * 0.988)}{0.998}$$

$$= 14.8.$$

That is, the front wheel of the robot will turn to the right 14.8°. The robot moves for a short distance and then the process repeats for the new position. Figure 9.36 represents the track of the movement of the robot after 100 iterations.

	L	C	R
S	$f_{TL}(\cdot) = 0$	$f_{TL}(\cdot) = 0$	$f_{TR}(\cdot) = 0$
E	$f_{ST}(\cdot) = 0.01$	$f_{TL}(\cdot) = 0$	$f_{TL}(\cdot) = 0$
N	$f_{TR}(\cdot) = 0.988$	$f_{ST}(\cdot) = 0$	$f_{TL}(\cdot) = 0$
W	$f_{TR}(\cdot) = 0$	$f_{TR}(\cdot) = 0$	$f_{ST}(\cdot) = 0$

Figure 9.35 Membership degree for each rule when the initial starting point of the robot is at $x=-10$ and $\alpha=89$.

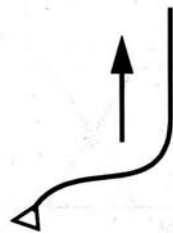


Figure 9.36 Movement of the robot for when $x=-10$ and $\alpha=89$.

In practice, fuzzy logic is applied to variety of products. For example, a group at Sanyo Corporation [SHI 91] has applied fuzzy logic to the cooking process in a rice cooker and obtained promising results. Their rice cooker is able to adjust itself appropriately to several factors (such as the water temperature and the rice quantity) to ensure delicious rice. In general, there are four states in making good rice.

1. *Water absorption.* The rice should absorb about 25% of the water.
2. *Boiling water.* Bringing the water to boil should take about 10 minutes.
3. *Cooking after boiling starts.* The rice is kept at a temperature above 98°C for more than 20 minutes.
4. *Water evaporation.* The extra water on the surface should evaporate.

Figure 9.37 represents the change of temperatures in these four states. Among these states, state 2 is more complex and is harder to implement. This is so because raising the temperature requires an assessment of the amount of rice and the amount of water. However, the surrounding environment, such as room temperature, water temperature, current of electricity, and shape of the interior pot, makes this process difficult. Hence the conventional type of control circuit becomes too complex. Fuzzy logic makes this process less complex and more implementable.

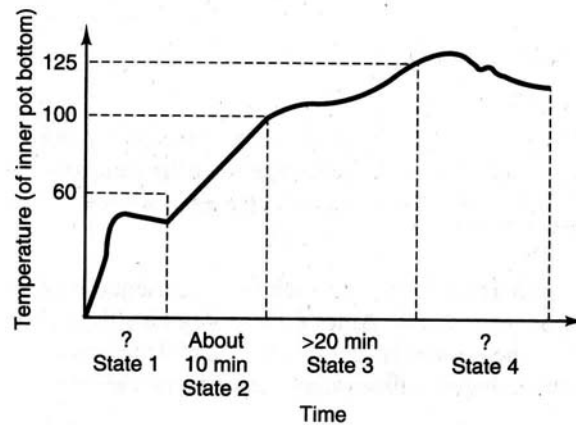


Figure 9.37 States of rice cooking.

Fuzzy logic can also control the electric power for heating based on the differences between standard (stored) data and actual data on the amount of rice, water, and temperature. For example, Figure 9.38 represents fuzzy values and their ranges for two of the variables, the difference in temperature and the difference in the amount of water. There are a number of rules for controlling this heating process. For example, a rule might be that if the differences in temperature and the amount of water are both positive then power should be increased.

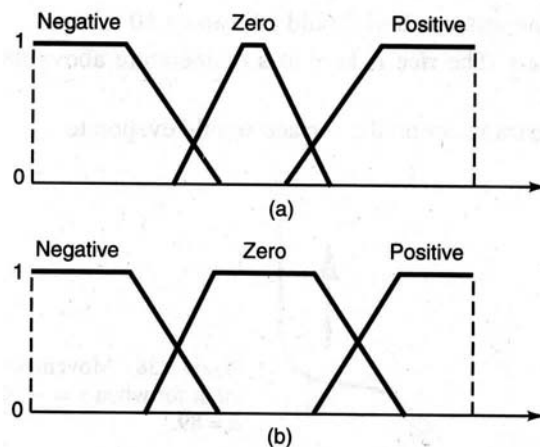


Figure 9.38 Membership functions for the rice cooking controller. (a) Difference in temperature. (b) Difference in amount of water.

In summary, fuzzy logic is making its way through many applications, ranging from home appliances to

decision-support systems. Fuzzy logic makes the development of a decision-support system easier, less complex, inexpensive, faster, and more reliable. In a mathematical model, if one equation is wrong, the whole system process fails. Fuzzy output is the effect of multiple rules, so even if one rule is faulty, the others will often compensate.

Although software implementation of fuzzy logic provides good results for some applications, dedicated fuzzy processors, called fuzzy logic accelerators, are required for implementing high-performance applications. In recent years, several fuzzy logic accelerators have been developed. Some of them are American Neurology, Inc., NLX-230, Togai Infralogic FC110, and VLSI Technology VY86C500. The general architecture of a fuzzy logic accelerator is explained in the following section.

9.4.4 Architecture of a Fuzzy Logic Accelerator

In general, there are five main units in a fuzzy logic accelerator [VLS 93]: membership function unit, rule evaluation unit, defuzzification unit, storage unit, and control unit. The function of each of these units is explained next.

Membership function unit. The membership function unit computes the degree of membership for each input value. There are different ways to implement such a unit. One way is to implement a lookup table for each input variable. Each lookup table holds the degrees of membership for possible values of an input variable. Another way is to design a unit that supports certain membership functions, such as piecewise linear, trapezoidal, and S-function. The degrees of membership are computed according to these predefined functions.

Rule evaluation unit. The rule evaluation unit evaluates the contribution of each rule on the output variables. It supports fuzzy operations such as *and*, *or*, and *not*. To understand the function of this unit, let's consider our robot example. In this example, two of the rules were

Rule 1: if(α =E and x =L) then β =ST.

Rule 2: if(α =N and x =L) then β =TR.

Suppose that we have input values 89 and -10 for direction α and distance x , respectively. The output of each rule for these input values is represented in Figure 9.39. This figure illustrates the function of the rule contribution unit. Note that the *and* operation is performed by taking the minimum of the membership degrees. The rule minimum membership degrees are used to scale the output membership functions of the output variables that are affected by the rules. The affected output membership functions are scaled in the y-direction.

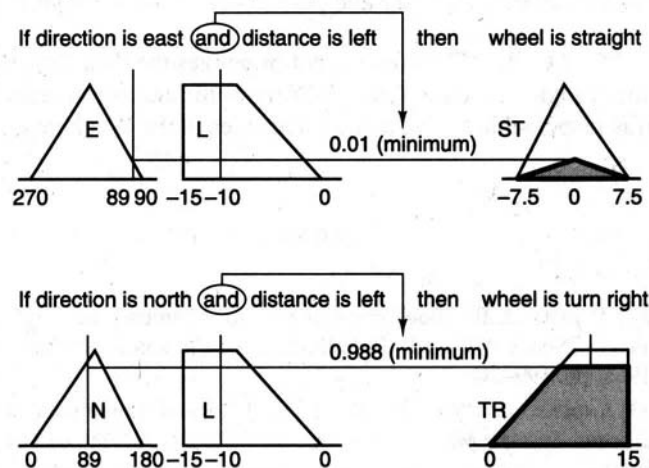


Figure 9.39 Rule evaluation steps.

Defuzzification unit. The defuzzification unit performs the defuzzification step on the scaled output

membership functions. Often, the centroid defuzzification method is implemented in fuzzy logic accelerators. As illustrated in Figure 9.40, the centroid method adds the scaled output membership functions to form a composition and then computes the center of mass on the composition.

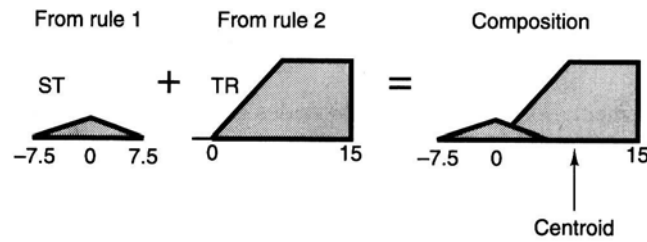


Figure 9.40 Defuzzification steps.

Storage unit. The storage unit holds the rules of the fuzzy control system. It is also used to store temporary and permanent results of the rule evaluations.

Control unit. The control unit organizes the data flow between separate units of the fuzzy logic accelerator and determines the execution order of instructions. Basically, it is responsible for all control activities in the accelerator chip.