7

FUNDAMENTALS OF NEURAL NETWORKS

7.1 INTRODUCTION

In 1956, the Rockefeller Foundation sponsored a conference at Dartmouth College that had as its scope

> The potential use of computers and simulation in every aspect of learning and any other feature of intelligence.

It was at this conference that the term "artificial intelligence" came into common use. Artificial intelligence can be broadly defined as

> Computer processes that attempt to emulate the human thought processes that are associated with activities that require the use of intelligence.

Generally, this definition included the fields of automatic learning, understanding natural language, vision-image recognition, voice recognition, game playing, mathematical problem solving, robotics, and expert systems. In recent years, some researchers have included neural networks and other related technologies as constituents of artificial intelligence, while others, pointing to their origin in biological sciences, have sought to avoid this association. In this text we accept neural networks as a legitimate field of artificial intelligence. Furthermore, we include genetic algorithms, fuzzy logic or fuzzy systems, wavelets, cellular automata, and chaotic systems as being within the general field of artificial intelligence.

7.2 BIOLOGICAL BASIS OF NEURAL NETWORKS

The human brain is a very complex system capable of thinking, remembering, and problem solving. There have been many attempts to emulate brain functions with computer models, and although there have been some rather spectacular achievements coming from these efforts, all of the models developed to date pale into oblivion when compared with the complex functioning of the human brain.

A *neuron* is the fundamental cellular unit of the brain's nervous system. It is a simple processing element that receives and combines signals from other neurons through input paths called *dendrites*. If the combined input signal is strong enough, the neuron "fires," producing an output signal along the axon that connects to the dendrites of many other neurons. Figure 7.1 is a sketch of a neuron showing the various components. Each signal coming into a neuron along a dendrite passes through a *synapse* or *synaptic junction*. This junction is an infinitesimal gap in the dendrite that is filled with neurotransmitter fluid that either accelerates or retards the flow of electrical charges. The fundamental actions of the neuron are chemical in nature, and this neurotransmitter fluid produces electrical signals that go to the nucleus or *soma* of the neuron. The adjustment of the impedance or conductance of the synaptic gap is a critically important process. Indeed, these adjustments lead to memory and learning. As the synaptic strengths of the neurons are adjusted, the brain "learns" and stores information.

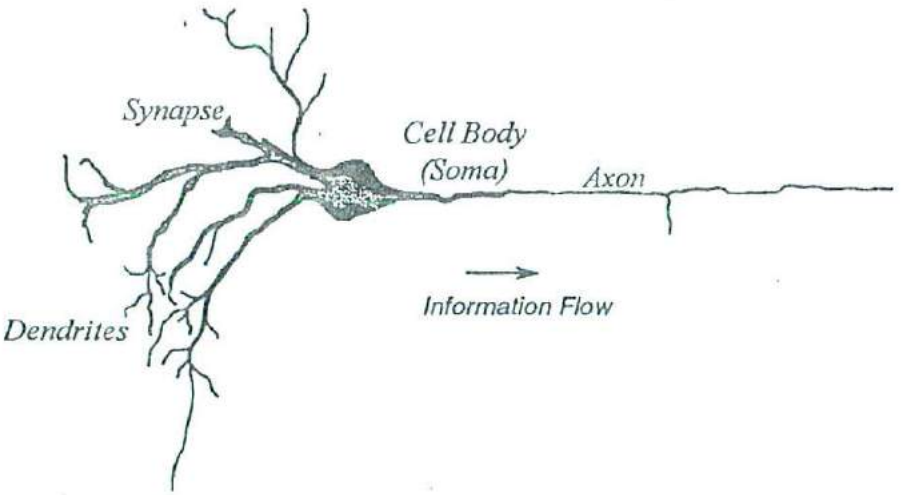


Figure 7.1 Sketch of a biological neuron showing components.

When a person is born, the cerebral cortex portion of his or her brain contains approximately 100 billion neurons. The outputs of each of these neurons are connected through their respective *axons* (output paths) to about 1000 other neurons. Each of these 1000 paths contains a synaptic junction by which the flow of electrical charges can be controlled by a neurochemical process. Hence, there are about 100 trillion synaptic junctions that are capable of having influence on the behavior of the brain. It is readily apparent that in our attempts to emulate the processes of the human brain, we cannot think of billions of neurons and trillions of synaptic junctions. Indeed, the largest of our neural networks typically contain a few thousand artificial neurons and less than a million artificial synaptic junctions.

The one area in which artificial neural networks may have an advantage is speed. When a person walks into a room, it typically takes another person about half a second to recognize them. We are told that this recognition process involves about 200–250 individual separate operations within the brain. As a benchmark for speed, this means that the human brain operates at about 400–500 hertz (Hz). Modern digital computers typically operate at clock speeds between 100 and 200 megahertz (MHz), which means that they have a very large speed advantage over the brain. However, this advantage is dramatically reduced because digital computers operate in a serial mode whereas the brain operates in a parallel mode. However, neural network chips have been developed in recent years that enable neural computers to operate in a parallel mode.

The nomenclature in the neural network field is still not standardized. You will find books and technical articles that refer to artificial neural networks as *connectionist systems* and artificial neurons as *processing elements* (PEs), *neurodes*, *nodes*, or simply *neurons*. In this text we shall use the terms *neurons* and *neural networks*, except in situations where an alternate designation would be more descriptive. Often we drop the adjective "artificial," because we deal only with artificial neurons in this text.

7.3 ARTIFICIAL NEURONS

An artificial neuron is a model whose components have direct analogs to components of an actual neuron. Figure 7.2 shows the schematic representation of an artificial neuron. The input signals are represented by $x_0, x_1, x_2, \ldots, x_n$. These signals are continuous variables, not the discrete electrical pulses that occur in the brain. Each of these inputs is modified by a *weight* (sometimes called the *synaptic weight*) whose function is analogous to that of the synaptic junction in a biological neuron. These weights can be either positive or negative, corresponding to acceleration or inhibition of the flow of electrical signals. This processing element consists of two parts. The first part simply aggregates (sums) the weighted inputs resulting in a quantity

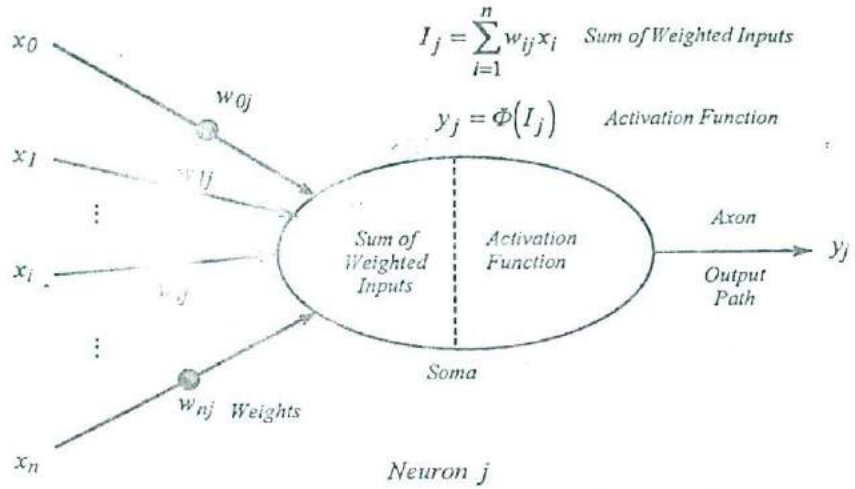


Figure 7.2 Schematic representation of an artificial neuron.

I; the second part is effectively a nonlinear filter, usually called the *activation function*,[1] through which the combined signal flows.

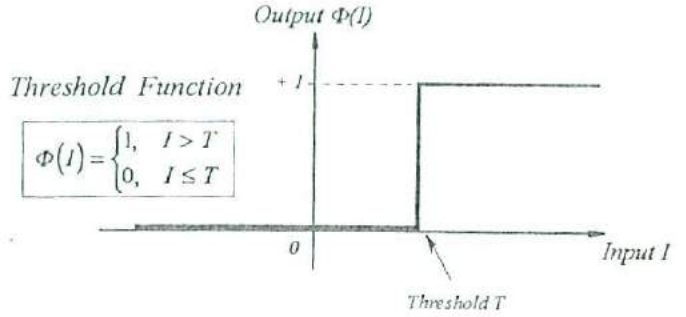
Figure 7.3 shows several possible activation functions. It may be a threshold function as shown in Figure 7.3*a* that passes information (usually a $+1$ signal) only when the output I of the first part of the artificial neuron exceeds the threshold T. It can be the signum function (sometimes called a quantizer function) shown in Figure 7.3*b* that passes negative information when the output is less than the threshold T and positive information when the output is greater than the threshold T. More commonly, the activation function is a continuous function that varies gradually between two asymptotic values, typically 0 and 1, or -1 and $+1$, called the *sigmoidal function*. The most widely used activation function is the logistic function, one of the sigmoidal activation functions, which is shown in Figure 7.3*c* and is represented by the equation

$$\Phi(I) = \frac{1}{1 + e^{-\alpha \cdot I}} \tag{7.3-1}$$

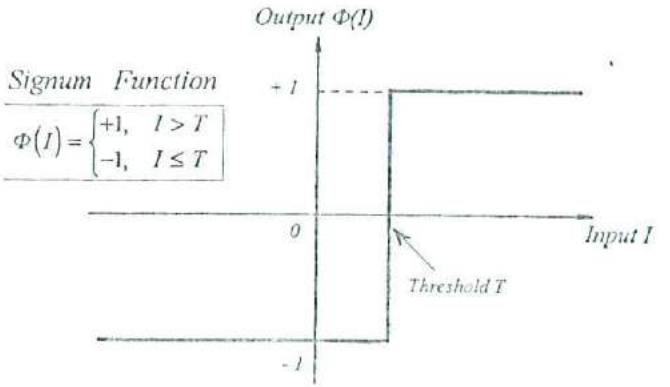
where α is a coefficient that adjusts the abruptness of this function as it changes between the two asymptotic values.

A more descriptive term for the activation function is "squashing function," which indicates that this function squashes or limits the values of the output

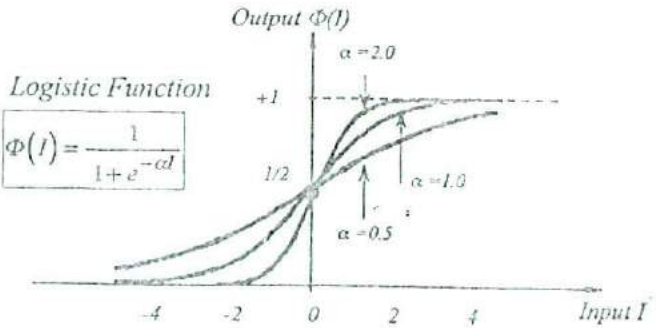
[1]A rather common name used in many books for the activation function is "transfer function." We will avoid the use of this term in the text to avoid confusion, because this term is commonly used in engineering—to describe the input–output behavior of linear systems.



(a)



(b)



(c)

Figure 7.3 Transfer functions for neurons: (a) Threshold activation function (when $T = 0$, this is called a binary activation function). (b) Signum activation function (sometimes called a quantizer'). (c) Logistic activation functions for $\alpha = 0$, 5, 1 and 2.

of an artificial neuron to values between the two asymptotes. This limitation is very useful in keeping the output of the processing elements within a reasonable dynamic range. However, there are certain situations in which a linear relation, sometimes only in the right half-plane, is used for the activation function. It should be noted, however, that the use of a linear activation function removes the nonlinearity from the artificial neuron. Without nonlinearities, a neural network cannot model nonlinear phenomena.

7.4 ARTIFICIAL NEURAL NETWORKS

An artificial neural network can be defined as

> A data processing system consisting of a large number of simple, highly interconnected processing elements (artificial neurons) in an architecture inspired by the structure of the cerebral cortex of the brain.

These processing elements are usually organized into a sequence of layers or slabs with full or random connections between the layers. This arrangement is shown in Figure 7.4, where the input layer is a buffer that presents data to the network. This input layer is *not* a neural computing layer because the nodes have no input weights and no activation functions. (Some authors do not count this layer in describing neural networks. We will count it, but we will use different symbols for the nodes in this layer where there is a need to distinguish between the different kinds of neurons.) The top layer is the output layer which presents the output response to a given input. The other layer (or layers) is called the intermediate or hidden layer because it usually

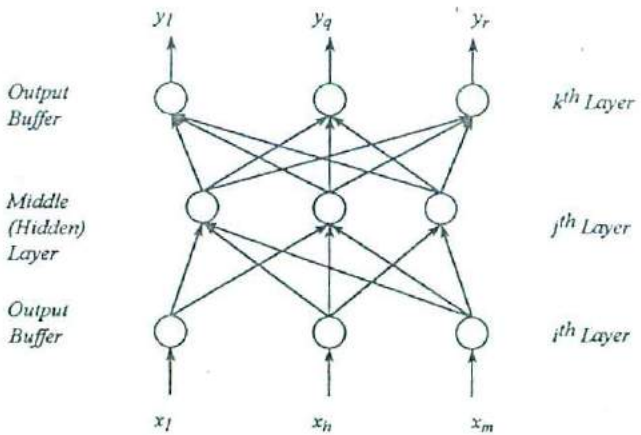


Figure 7.4 Example of an neural network architecture.

has no connections to the outside world. Typically the input, hidden, and output layers are designated the ith, jth, and kth layers, respectively.

Two general kinds of neural networks are in use: the *heteroassociative* neural network in which the output vector is different than the input vector, and the *autoassociative* neural network in which the output is identical to the input. Unless otherwise indicated, all neural networks in this book are heteroassociative.

A typical neural network is "fully connected," which means that there is a connection between each of the neurons in any given layer with each of the neurons in the next layer as shown in Figure 7.5. When there are no lateral connections between neurons in a given layer and none back to previous layers, the network is said to be a feedforward network. Neural networks with feedback connections (i.e., networks with connections from one layer back to a previous layer) are also useful and are discussed in the following chapters. Lateral connections between neurons in the same layer are also called feedback connections. In certain cases, a neuron has feedback from its

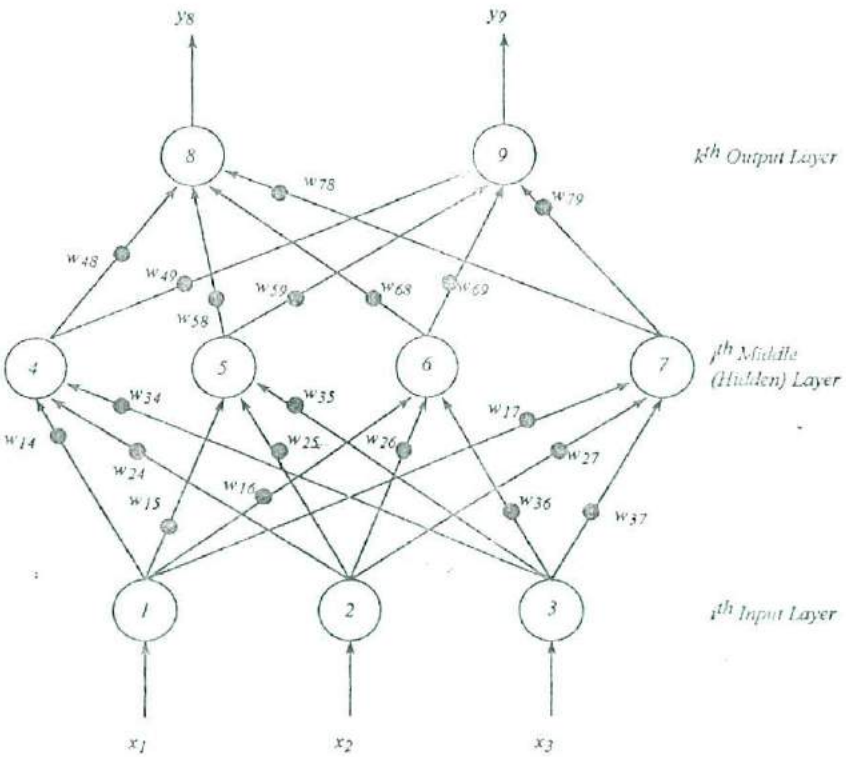


Figure 7.5 Simple feedforward neural network.

output to its own input. In all cases, these connections have weights that must be trained.

Each of the connections between neurons has an adjustable weight as shown in Figure 7.5. This simple neural network is a fully connected, feedforward network with three neurons in the input layer, four in the middle or hidden layer, and two in the output layer. The individual weights are shown as solid dots on the connection and are designated by symbols such as w_{ij}. For instance, the symbol w_{37} indicates a weight on the connection between neurons 3 and 7.

Let us consider the neural network of Figure 7.5, which has an input vector X consisting of components x_1, x_2, and x_3 and an output vector Y having components y_8, and y_9. When a signal x_1 is applied to neuron 1 in the input layer, the output x_1 goes to each of the artificial neurons in the middle or hidden layer, passing through weights w_{14}, w_{15}, w_{16}, and w_{17}. The input signal x_2 and x_3 behave in a similar manner, sending signals to neurons 4, 5, 6, and 7 through the appropriate weights, as shown in Figure 7.5.

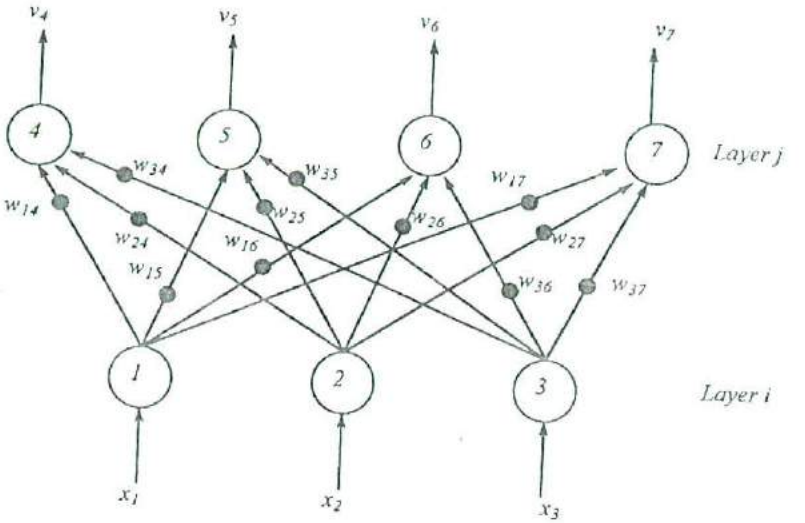
Now let us consider the behavior of neuron 4. It has three inputs from the three neurons in the input layer that have been modified by the connection weights w_{14}, w_{24}, and w_{34}. The first part of this neuron simply sums up these three weighted inputs. Then this summation is passed to the second part of the neuron, which is a nonlinear function—typically a logistic curve between 0 and 1 as shown in Figure 7.3*c*. The output of this activation function or squashing function is then sent to neurons 8 and 9 through weights w_{48} and w_{49}. Neurons 5, 6, and 7 behave in a similar manner. Neurons 8 and 9 collect the weighted inputs from neurons 4, 5, 6 and 7, sum them, and pass the sums through the activation functions to produce y_8 and y_9, the components of the output vector Y.

Vector and Matrix Notation

It is convenient to utilize vector and matrix notation in dealing with the inputs, outputs, and weights. Let us cut the neural network in Figure 7.5, just above the hidden layer as shown in Figure 7.6. The outputs of neurons 4, 5, 6, and 7 are shown to be the vector $\mathbf{V}_j$, which has components v_4, v_5, v_6, and v_7. If we limit the activation functions to linear functions, the mathematical relationships described in the previous section can be written in matrix form; that is, the column vector $\mathbf{V}_j$ is equal to the dot product of the weight matrix $\mathbf{W}_{ji}$ and the input vector $\mathbf{X}_i$. This relationship is given by

$$\begin{bmatrix} v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} = \begin{bmatrix} w_{14} & w_{24} & w_{34} \\ w_{15} & w_{25} & w_{35} \\ w_{16} & w_{26} & w_{36} \\ w_{17} & w_{27} & w_{37} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{7.4-1}$$

$$\mathbf{V}_j = \mathbf{W}_{ij} \cdot \mathbf{X}_i \tag{7.4-2}$$



$$\mathbf{V} = W_{ij} \cdot \mathbf{X}$$

Figure 7.6 Lower portion of neural network cut above the hidden layer.

In a similar manner using the upper half of the artificial neural network shown in Figure 7.7, it can be shown that the output vector Y_k is equal to the dot product of the weight matrix W_{jk} and the input vector V_j. This relationship is given by

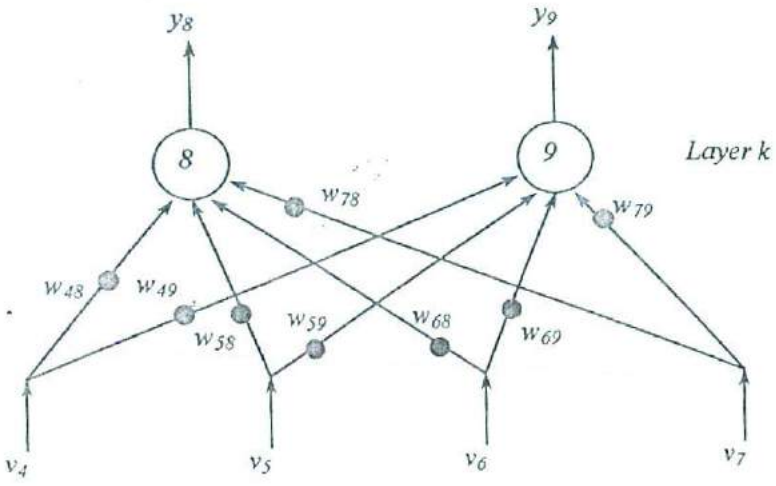
$$\begin{bmatrix} y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} w_{48} & w_{58} & w_{68} & w_{78} \\ w_{49} & w_{59} & w_{69} & w_{79} \end{bmatrix} \cdot \begin{bmatrix} v_4 \\ v_5 \\ v_6 \\ v_7 \end{bmatrix} \tag{7.4-3}$$

or

$$\mathbf{Y}_k = \mathbf{W}_{jk} \cdot \mathbf{V}_j \tag{7.4-4}$$

By combining equations (7.4-1) and (7.4-3), it is apparent that the output vector $\mathbf{Y}_k$ is equal to the dot product of the two matrices and the input vector $\mathbf{X}_i$.

$$\begin{bmatrix} y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} w_{48} & w_{58} & w_{68} & w_{78} \\ w_{49} & w_{59} & w_{69} & w_{79} \end{bmatrix} \cdot \begin{bmatrix} w_{14} & w_{24} & w_{34} \\ w_{15} & w_{25} & w_{35} \\ w_{16} & w_{26} & w_{36} \\ w_{17} & w_{27} & w_{37} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{7.4-5}$$



$$\mathbf{Y} = \mathbf{W}_{jk} \cdot \mathbf{V}$$

Figure 7.7 Upper portion of neural network cut above the hidden layer.

Since the two matrices can then be reduced to a single matrix, $\mathbf{W}_{ijk}$, it follows that the output vector $\mathbf{Y}$ is equal to the dot product of the combined matrix and the input vector:

$$\mathbf{Y}_k = \mathbf{W}_{ij} \cdot \mathbf{W}_{jk} \cdot \mathbf{X}_j = \mathbf{W}_{ijk} \cdot \mathbf{X}_i \qquad (7.4\text{-}6)$$

The limitation of linear activation functions means that the relationships given in equations (7.4-1) through (7.4-6) are severely limited. Indeed, this indicates that a three-layer perceptron with linear activation functions in the middle and output layers can be replaced with a two-layer network with a linear activation function in the output layer. Nevertheless, this process has introduced the concept of the weight matrices, which is very useful in many situations.

Neural Networks and Feedback

The feedforward neural network shown in Figure 7.5 operates in a simple straightforward manner. When the vector $\mathbf{X}_i$ is applied to the input layer, the calculations for weighting inputs, as well as summing and filtering, are rapidly carried out for each neuron as the process moves from the input to the middle layer and on to the output layer. However, when there are feedback

connections, either between neurons in the same layer or from one layer to an earlier layer, the process is much more complicated. In a neural network where the mathematical operations are performed almost instantaneously, information reverberates around the network, across layers and within layers, until some sort of convergence status is reached. When the mathematical operations are implemented serially, the process is more complicated. The outputs for the feedforward connections are performed first, then the calculations for the feedback connections are performed, then the calculations for the feedforward connections are again performed using the results of the previous calculations, and this process continues until equilibrium values are reached. Under many circumstances, artificial neural networks with feedback connections can be very useful. However, about 80% of the neural network applications today utilize feedforward neural networks.

Neural Networks in Perspective

Neural networks have profound strengths and weaknesses, and these must be recognized if they are to be used properly. Although neural networks are sometimes called neural computers, they are in fact not computers; but rather, they are basically memories that memorize results, just as the human brain memorizes certain results. For instance, a person memorizes the fact that the product of four times six is twenty-four, and this fact is stored in the person's memory for life. On the other hand, the cheapest digital calculator actually calculates the product every time the numbers are entered.

Neural networks use memory-based storage of information in ways that are different and more flexible than simple storage in a look-up table. In the neural network, as in the brain, the storage of information is distributed throughout the network. Although this makes it hard to keep things separate that should be kept separate, it does give rise to the networks' ability to make generalizations that are so important to the practical applications of neural networks. Furthermore, the loss of a few neurons (real or artificial) does not materially affect the information stored.

Linear Associator Neural Network

The most elementary neural network is a "linear associator" that, along with its learning rules, can be used to demonstrate the abilities and limitations of neural networks. We start with the fundamental assumption that information is stored by a pattern or a set of activities of many neurons that is often represented as a "state vector." Hence, the output of the network is the result of the interaction of many neurons (sometimes called neural computing), not just the response of a single neuron. As discussed earlier, the fundamental neuron sums the weighted inputs and then subjects this sum to a nonlinear activation function, typically a sigmoidal function, to keep the output of the neuron within a reasonable range.

The architecture of a linear associator is a set of input neurons that are connected to a set of output neurons (i.e., a two-layer neural network). Any particular output y_j (one component of the output state vector **Y**) can be computed from the activities of all the various inputs x_i and the strengths of the weights on the connections. Mathematically, the output from the summing unit is equal to the *inner product* (dot product) between the weight matrix and the input vector. In the linear associator, the activation function is a linear function. While this simplifies the network considerably, care must be taken to ensure that the outputs do not exceed the range of the output neuron.

In simple terms, the operation of a linear associator involves the input of a pattern that then produces the output pattern that we want (i.e., the "right" answer). For this to happen, we have to train the weights of the linear associator to give the desired pattern. This can be accomplished by presenting the network with training vector pairs (inputs and desired outputs) and utilizing an appropriate training rule. Any of the different training rules discussed later can be used to perform this training. In theory, the initial weights can have any values, but experience indicates that starting with small randomized weights is advantageous.

Suppose that we have one set of neurons projecting to another set through modifiable weights as shown in Figure 7.8. When the activation functions of the neurons are linear, this network is a linear associator. What this means is

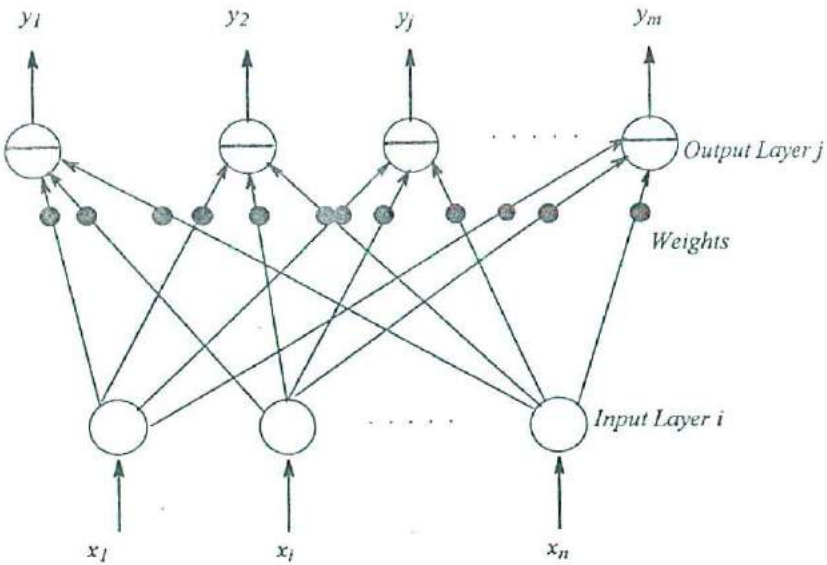


Figure 7.8 Sketch of a linear associator.

that after the neural network is trained, presentation of an input pattern to the input layer will produce the desired (associated) output pattern. This is represented mathematically by the following equation:

$$\mathbf{Y}_i = \mathbf{W} \cdot \mathbf{X}_i \tag{7.4-7}$$

where W is the trained weight matrix, and $\mathbf{X}_i$ and $\mathbf{Y}_i$ are the ith input and output vectors, respectively.

One of the unique and advantageous features of the linear associator is its ability to store more than one relationship simultaneously. This is discussed and demonstrated in a later section. The problems with the linear associator is that it is not very accurate, especially if too many items are stored in the associator. Second, simple networks that use *Hebbian learning*[2] cannot compute some functions that may be desired. This leads to the concept of comparing the output with the desired output and using the difference (error) as a basis for adjusting the weights, such as is the case in *Widrow–Hoff learning*. In effect, this procedure constitutes a form of "supervised" learning that is discussed in the next chapter.

7.5 LEARNING AND RECALL

Neural networks perform two major functions: *learning* and *recall*. Learning is the process of adapting the connection weights in an artificial neural network to produce the desired output vector in response to a stimulus vector presented to the input buffer. Recall is the process of accepting an input stimulus and producing an output response in accordance with the network weight structure. Recall occurs when a neural network globally processes the stimulus presented at its input buffer and creates a response at the output buffer. Recall is an integral part of the learning process since a desired response to the network must be compared to the actual output to create an error function.

The learning rules of neural computation indicate how connection weights are adjusted in response to a learning example. In *supervised learning*, the artificial neural network is trained to give the desired response to a specific input stimulus. In *graded learning*, the output is "graded" as good or bad on a numerical scale, and the connection weights are adjusted in accordance with the grade.

In *unsupervised learning* there is no specific response sought, but rather the response is based on the networks ability to organize itself. Only the input stimuli are applied to the input buffers of the network. The network then organizes itself internally so that each hidden neuron responds strongly to a different set of input stimuli. These sets of input stimuli represent

[2] *Hebbian learning* as well as other learning paradigms are presented in Chapter 9.

clusters in the input space (which often represent distinct real-world concepts or features).

The vast majority of learning in engineering applications involves supervised learning. In this case a stimulus is presented at the input buffer representing the input vector, and another stimulus is presented at the output buffer representing the desired response to the given input. This desired response must be provided by a knowledgeable teacher. The difference between actual output and desired response constitutes an error, which is used to adjust the connection weights. In other cases, the weights are adjusted in accordance with criteria that are prescribed by the nature of the learning process, as in competitive learning or in Hebbian learning.

There are a number of common supervised learning algorithms utilized in neural networks. Perhaps the oldest is Hebbian learning, named after Donald Hebb, who proposed a model for biological learning (Hebb, 1949) where a connection weight is incremented if both the input and the desired output are large. This type of learning comes from the biological world, where a neural pathway is strengthened each time it is used. "Delta rule" learning takes place when the error (i.e., the difference between the desired output response and the actual output response) is minimized, usually by a least squares process. Competitive learning, on the other hand, occurs when the artificial neurons compete among themselves, and only the one that yields the largest response to a given input modifies its weight to become more like the input. There is also random learning in which random incremental changes are introduced into the weights, and then either retained or dropped, depending upon whether the output is improved or not (based on whatever criteria the user specifies).

In the recall process, a neural network accepts the signal presented at the input buffer and then produces a response at the output buffer that is determined by the "training" of the network. The simplest form of recall occurs when there are no feedback connections from one layer to another or within a layer (i.e., the signals flow from the input buffer to the output buffer in a "feedforward" manner). In a feedforward network the response is produced in one cycle of calculations by the computer.

Supervised Learning

In order to demonstrate supervised learning, let us modify the neural network shown in Figure 7.5, to include a desired output pattern, a comparator, and a weight adjusting algorithm. This arrangement is shown in Figure 7.9, where the desired output is represented by the vector **Z** with components z_8 and z_9. The inputs to the comparator are the desired output pattern **Z** and the actual output pattern **Y**. The error coming from the comparator—that is, the difference between **Y** and **Z**—is then utilized in the weight-adjusting algorithm to determine the amount of the adjustment to be made in the weights in both layers.

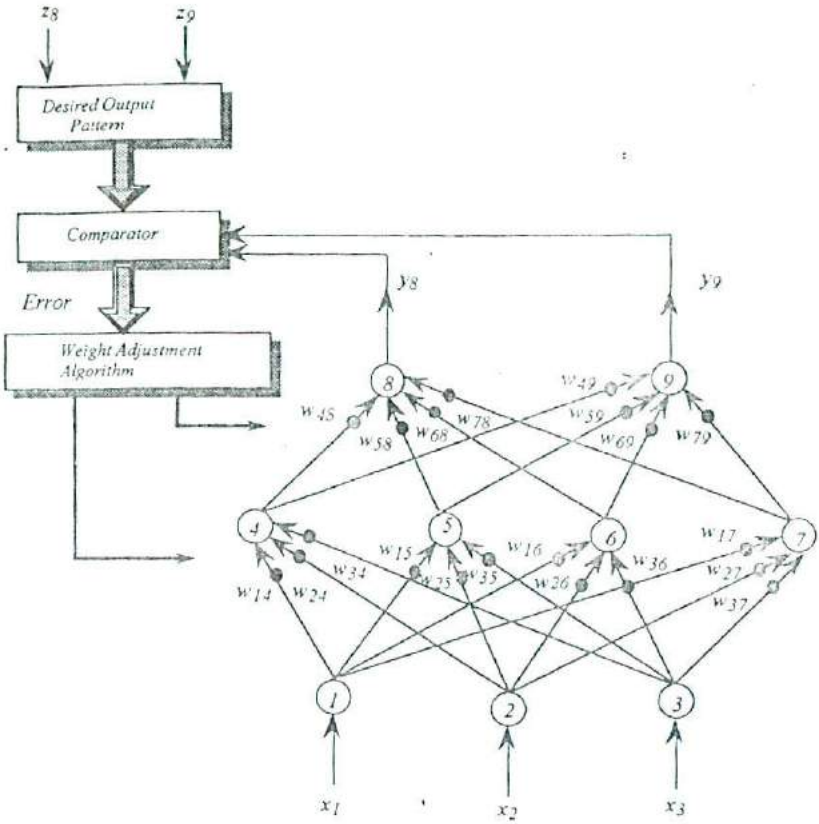


Figure 7.9 A neural network with supervised learning.

In order to start the process, let us randomly adjust all the weights in the neural network in Figure 7.9 to small random values, and then consider the training pair **X** and **Z** with components x_1, x_2, x_3 and z_8, z_9, respectively. When the vector **X** is applied to the neural network, it produces an output vector **Y**, which is compared with the vector **Z** to produce the error. The weight-adjusting algorithm then modifies the weights in the direction that reduces this error. When the input vector **X** is again applied, it produces a new **Y**, which is compared with **Z**, and the error is applied to the weight-adjusting algorithm again to adjust the weights. This process is repeated over and over until the error is reduced to some specified value or an irreducible small quantity. At that point the output vector **Y** and the desired output vector **Z** are substantially equivalent, and the neural network is said to have been trained to map input vector **X** into the desired output vector **Z**. This is

the essence of supervised training. Of course, we must specify and explain the mechanism by which the weights are adjusted before there is a complete understanding of this process.

Example 7.1 Mapping the Alphabet to a Five-Bit Code. In order to understand how a neural network training process works, let us consider the arrangements shown in Figure 7.10. On the left we have a 7×5 matrix array of inputs that are restricted to either 0 or 1. In the center we have a neural network, with the input layer on the left having 35 input artificial neurons. Each of these 35 neurons is connected to one of the inputs from the 7×5 matrix array. On the right we have a 5×1 matrix array of outputs, each of them connected to one of the neurons in the output layer of the neural network in the center. The hidden layer in the neural network in this case has 20 artificial neurons, a number that was chosen arbitrarily. The input vector on the left, **X**, has 35 components $(x_1, x_2, \ldots, x_{35})$ and the output vector **Y** on the right has five components $(y_1, y_2, y_3, y_4, y_5)$. In effect, we are going to map the pattern contained in the 7×5 matrix on the left into a pattern on the right contained by the 5×1 matrix. In a sense, this is a form of data compression where the data contained in the 35-bit matrix on the left is mapped into the five-bit matrix on the right. The compression ratio in this case is 7 : 1.

Let us introduce a pattern to represent an uppercase letter A in the 7×5 matrix on the left, where the shaded areas in Figure 7.10 represent 1s and the unshaded areas represent 0s. Suppose we want to map this pattern into the five-bit pattern in the matrix on the right, which is shown to be (1, 0, 1, 0, 1). The artificial neural network has an input pattern representing the A, and the desired output pattern is represented by the five-bit matrix on the right. To carry out this mapping, we must adjust the weights in both the

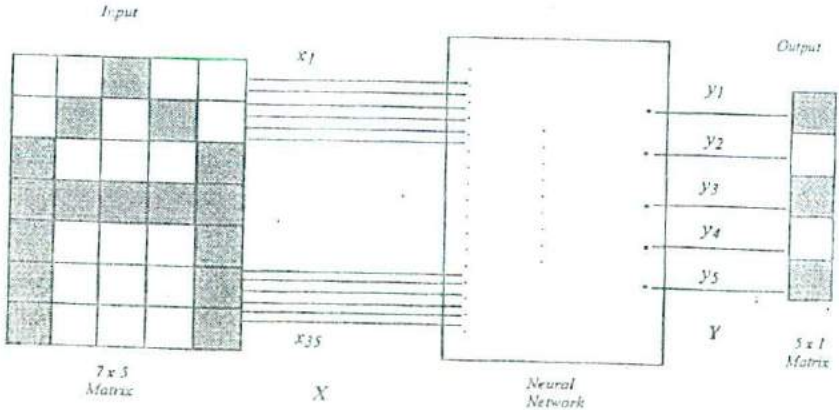


Figure 7.10 Input–output mapping of the letter *A*.

connections between the input and hidden layers and the connection between the hidden and output layers.

If the neural network is fully connected, we have 700 (35×20) connections with 700 weights between the input and hidden layers. In the connections between the hidden and output layers, we have another 100 (20×5) weights, giving a total of 800 weights that must be adjusted. In effect, we can think of this arrangement as having 800 degrees of freedom because of the 800 adjustable weights. It is very clear that we do not need 800 degrees of freedom to map a 35-bit input into a five-bit output. What this means is that there are hundreds, if not thousands, of different combinations of these 800 weights that will permit this neural network to carry out this mapping.

In order to start the training for this mapping, all the weights in the neural network are set to small random values, usually between -0.3 and $+0.3$. Then the training process is started. This involves applying the pattern from the 35-bit matrix on the left to the input layer, multiplying these inputs by 700 connection weights between the input and hidden layer, and then summing the 35 weighted inputs going into each of the 20 neurons in the hidden layer. These 20 sums then pass through the nonlinear activation function to produce the 20 outputs that go to each of the five neurons in the output layer. Each of these 20 outputs is multiplied by the appropriate weights, summed by each output neuron and passed through the nonlinear activation function to produce the five outputs. (Note that these outputs are not 0s and 1s, but rather numerical values between 0 and 1. Therefore, an interpretation of the outputs is needed. For instance, an output greater than 0.9 could be considered as a 1; an output less than 0.1 could be considered as a 0; and any value in between 0.1 and 0.9 could be considered as indeterminate.) These outputs are then compared with the desired output shown in the 5×1 matrix on the right. The difference between the actual output of the neural network and the desired output becomes the error vector that is then used to adjust both layers of weights in such a way that the overall error is reduced. Then the process is repeated over and over again until eventually, every time an A is applied to the input, the desired output is produced by the neural network within limits prescribed by some specific criteria. At this point we say that the neural network is trained and is capable of mapping a 35-bit representation of A into a five-bit representation of the A.

Now let us consider the arrangement in Figure 7.11, where we have a 35-bit representation of a B as an input to the neural network and a five-bit representation of the B as a desired output, which in this case is (0, 1, 0, 1, 0). If we use the neural network we have just trained for an A and apply the B at the input matrix, we can continue the same procedure used before to calculate the output of the neural network and compare it with the desired output. Although there is a small probability that we might get the right output initially, the most likely outcome is that the actual output and the desired output will be quite different (i.e., some of the outputs will be wrong and others will be between 0.1 and 0.9 and hence indeterminate). This

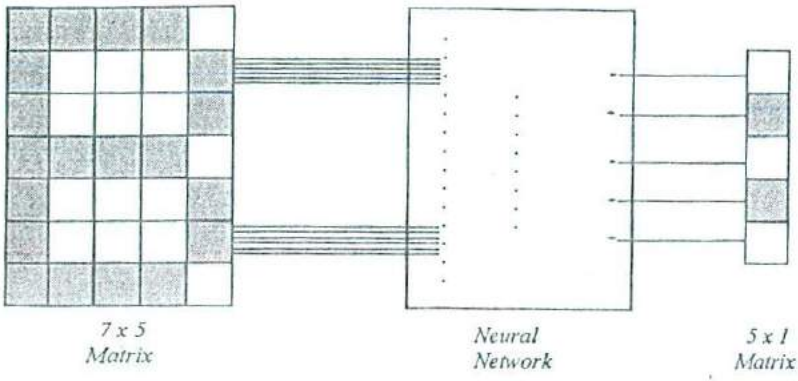


Figure 7.11 Input–output mapping of the letter *B*.

produces another error signal that becomes the basis for adjusting the weights further beyond the training provided for the A input. We continue this training process until every time we apply B at the input matrix on the left, we get the desired (0, 1, 0, 1, 0) output at the five-bit matrix at the right. At this point we have trained the neural network to map a 35-bit representation of a B into a five-bit representation.

Now suppose we again apply an A to this neural network that has been trained for an A and a B. Are we likely to get the desired output? Maybe, maybe not. If not, we can carry out additional training, until we achieve the desired results. Then we can apply the B again. Will we get the desired output? Maybe; maybe not. If not, we can carry out more training. This process of going back and forth between the A and B can be continued until every time we apply an A to the input matrix we get the desired (1, 0, 1, 0, 1) output and every time we apply a B to the input matrix we also get the desired (0, 1, 0, 1, 0) output. Now we have a network that is capable of mapping both an A and a B into the five-bit representations we specified.

Now let us apply a C to the input matrix as shown in Figure 7.12 and specify the desired output matrix as being a (1, 0, 0, 1, 0). When we apply the C, there is a high probability that we will not get the desired output that we have chosen. So we start the training process again and continue it until every time that we apply a C to the left-hand matrix, we get the desired (1, 0, 0, 1, 0) output. We now have a network that is capable of mapping the 35-bit representation of a C into the desired five-bit representation.

If we now apply the A to this trained network, will we get the desired output? Maybe, maybe not. If not, we perform more training until we achieve the desired result. Then we can apply a B. If we don't get the desired output, we carry out additional training. Then we can apply a C. If we don't get the

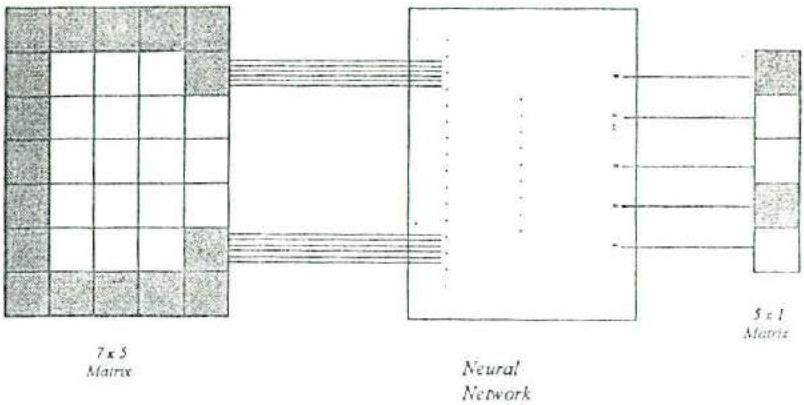


Figure 7.12 Input–output mapping of the letter *C*.

desired output, we carry out more training. We repeat this process over and over until every time we put in an A, a B, or a C, we get the desired output. At this point the neural network has been trained to map an A or B and a C into the desired representation that we have chosen.

At this point we could continue the process with D, E, and F and work our way through the alphabet. Since we have a five-bit binary output, the total number of possible mappings is 2^5, or 32. Hence, we can represent the whole alphabet plus six punctuation symbols. However, this is not a very efficient process to go through the complete training process for one symbol before starting the training process for another symbol. A more realistic and appropriate way would be to choose the 32 training sets (i.e., an A and its five-bit representation, a B and its five-bit representation, etc.) and, after randomizing the weights, to apply all 32 training sets, one after the other until we go all the way through the 32 letters and punctuation symbols once. This set of 32 input and desired output pairs, known as an *epoch*, is applied again and again until all 32 letters or symbols are mapped into the five-bit codes we specified.

Overall error is a better determination of the status of the training than it is of whether all the outputs are correct or not. This is simply the summation of all the errors between outputs of the neural network and the corresponding desired outputs (0s and 1s) for all pairs in the epoch. Ideally, this overall error should approach zero. If it does not, additional training should be carried out. However, if there is any noise in the inputs and/or outputs, an overall error of zero is never attained. Indeed, it is possible to overtrain a neural network until it fits the noise pattern rather than the underlying relationship. □

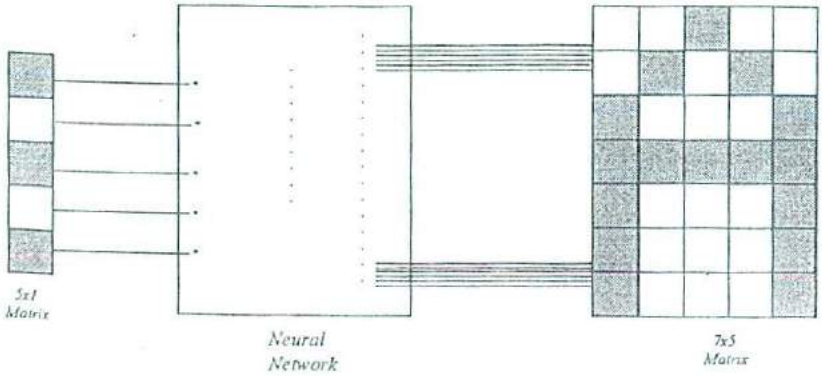


Figure 7.13 Inverse mapping of the letter *A*.

Example 7.2 Data Compression and Expansion. Now let us consider the arrangement shown in Figure 7.13. Here we have an input matrix that is a five-bit representation of an A and an output that is a 35-bit representation of the letter A. Is it possible to train a neural network to map a five-bit representation into a 35-bit representation? Yes, it is. The process is exactly the same as we went through in mapping the reverse arrangement. In this case, we have data expansion instead of data compression.

We can even have an arrangement that combined the networks shown in Figures 7.10 and 7.13—that is, a 35-bit representation that is compressed into a five-bit representation of an A and then is expanded back out to a 35-bit representation of the letter A as shown in Figure 7.14. Why would we want such an arrangement? Suppose we were sending information down a narrow-band data channel. We could compress the data (in this case by a factor of seven), send it down the channel, and then expand it back to the original symbol. This process is used in many practical situations. □

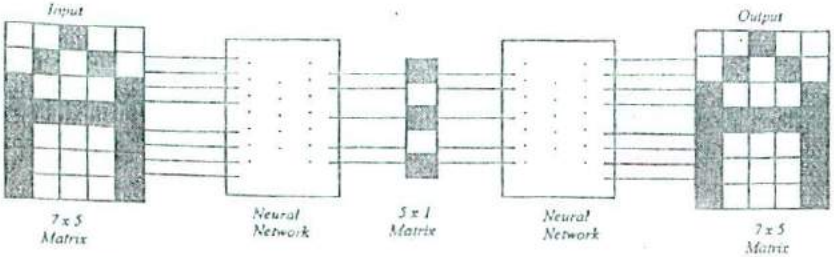


Figure 7.14 Compression and expansion using neural networks.

Example 7.3 Distortion Correction. A variation of the compression–expansion arrangement is shown in Figure 7.14, where the input to the neural network is exactly the same as the output. This arrangement is called an *autoassociative neural network*, which simply means that the input and the output are exactly the same. In this case, we randomly adjust the weights, apply the 35-bit representation of A as the input, apply the same 35-bit representation of the A on the right as the desired output, and start the training the process that we have used in the previous examples until we can consistently get an output that is equal to the desired output. Why would we want to do this? Suppose, after training the network, instead of using the 35-bit representation of an A as the input to the trained ANN, we put in a representation of a distorted A. The output of the neural network would be an undistorted A, because this is the only output pattern the neural network is trained to produce.

Suppose we go further and train this autoassociative network to represent all 26 letters of the alphabet plus the six punctuation symbols that we discussed earlier. It would then be reasonable to expect that every time that you applied a distorted symbol as the input, you would get the correct symbol as the output. In general, this is true, but there are exceptions. Suppose we put in a distorted B with the distortion in the lower right-hand side. This network might produce a B as the output or it might produce an *R*. The choice by the network would depend upon whether the distorted B input was closer, in a least square sense, to the B or the R that was used in the training. The same is true for other, similar combinations of letters—for example, Q & O, R & P, C & G, and perhaps others. □

7.6 FEATURES OF ARTIFICIAL NEURAL NETWORKS

What makes neural networks different from artificial intelligence or traditional computing? Generally, there are four features that are associated with artificial neural networks:

- They learn by example.
- They constitute a distributed, associative memory.
- They are fault-tolerant.
- They are capable of pattern recognition.

Neural networks are not the only systems capable of *learning by example*, but this feature certainly is an important characteristic of neural networks. Indeed, one of the most important characteristics of artificial neural networks is the ability to utilize examples taken from data and to organize the information into a form that is useful. Typically, this form constitutes a model that represents the relationship between the input and output variables. In

essence, this is what we were doing with the mapping exercises that we went through in the last section.

A neural network memory is both *distributed* and *associative*. By distributed, we mean that the information is spread among all of the weights that have been adjusted in the training process. These connection weights are the memory units of neural networks, and the values of the weights represent the current state of the knowledge of the network. Hence, each individual unit of knowledge is distributed across all the memory units in the network. Furthermore, it shares these memory units with all other items of information stored in the network.

The memory in a neural network is also *associative*. This means that if the trained network is presented with a partial input, the network will choose the closest match in the memory to that input and generate an output that corresponds to a full input. This is the process that was discussed with the autoassociative network in Figure 7.15, where the presentation of partial input vectors to the network resulted in their completion.

Neural networks are also fault-tolerant, since the information storage is distributed over all the weights. For instance, in the example in Figure 7.10, the information is distributed over 800 weights. Hence, the destruction or misadjustment of one or a few of these 800 weights does not significantly influence the mapping process between the inputs and outputs. In general, the amount of distortion is approximately equal to the fraction of the weights that have been destroyed.

Furthermore, even when a large number of the weights are destroyed, the performance of the neural network degrades gradually. While the performance suffers, the system does not fail catastrophically because the information is not contained in just one place but is, instead, distributed throughout the network. When neural networks are implemented in hardware, they are

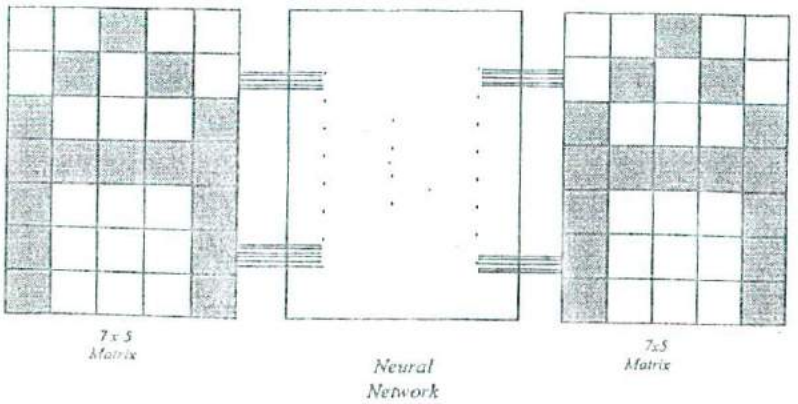


Figure 7.15 Input/output mapping in an autoassociative neural network.

very fault-tolerant, as contrasted to von Neumann-type computers where the failures of a single component can, in theory, lead to catastrophic results. For this reason, neural networks show great promise for use in environments where robust, fault-tolerant pattern recognition is necessary in a real-time mode, and the incoming data may be distorted or noisy. Such applications might include: nuclear power plants, missile guidance systems, space probes, or any system that is inaccessible for repair or where continuous performance is critical.

Pattern recognition requires the neural network to match large amounts of input information simultaneously and generate a categorical or generalized output with a reasonable response to noisy or incomplete data. Neural networks are good pattern recognizers, even when the information comprising the patterns is noisy, sparse, or incomplete. For a complex system with many sensors and possible fault types, real-time response is a difficult challenge to both human operators and expert systems. While the training time for a neural network may be long, once it has been trained to recognize the various conditions or states of a complex system, it only takes one cycle of the neural network to detect or identify a specific condition or state.

Neural computing networks consist of interconnected units that act on data instantly in a massive parallel manner. Indeed, when a neural network is implemented in hardware, such computation occurs virtually instantaneously. Such a neural computer provides an approach that is closer to human perception and recognition than that of conventional computers, and it can produce reasonable results with noisy or incomplete inputs.

7.7 HISTORICAL DEVELOPMENT OF NEURAL NETWORKS[3]

Artificial intelligence had its beginning at the Dartmouth Summer Research Conference in 1956, which was organized by Marvin Minsky (learning machines), John McCarthy (symbolic languages), Nathaniel Rochester (neural systems), and Claude Shannon (information theory). This conference led to the development of computer programs capable of making machines perform human-like or intelligent tasks and to the development of machines that used mechanisms modeled after studies of the brain to become "intelligent." The conference inspired Frank Rosenblatt to develop his concept of the perceptron, a generalization of the 1943 McCulloch–Pitts concept of the functioning of the brain by adding learning. The McCulloch–Pitts abstract model of a

[3]The history of the development of neural networks has been well documented by a number of books in the past few years: Caudill and Butler (1989, 1992), DARPA (1988), Hecht-Nielsen (1989), Maren, Pap, and Harston (1990), Miller, Sutton, and Werbos (1990), Nelson and Illingsworth (1990), Pao (1989z0, Simpson (1990), Wasserman (1989, 1993), and White and Sofge (1992). We will limit this review to descriptions of Rosenblatt's Perceptron, Minsky and Papert's review entitled *Perceptrons*; and Widrow's ADALINE because all had a profound influence on the development of neural networks.

brain cell was based on the theory that the probability of a neuron firing depended on the input signals and the voltage thresholds in the soma. It introduced the idea of a step threshold, but it did not have the ability to learn.

The first learning machine was actually built by Minsky and Dean in 1951 (before the Dartmouth conference) at the Massachusetts Institute of Technology. It had 40 processing elements, which, when described in neural network terms, were neurons with synapses that adjusted their weights according to their success in performing a specific task. Each neuron or processing element required six vacuum tubes and a motor/clutch/control system. The machine utilized Hebbian learning and was able to learn enough that it could "run a maze." It worked surprisingly well, considering the state of electronics and the understanding of the learning process at that time.

Rosenblatt's Perceptron

After the Dartmouth conference, Frank Rosenblatt of Cornell Aerolaboratory developed a computational model for the retina of the eye, called the "perceptron." The perceptron (see Figure 7.16) was inspired by the

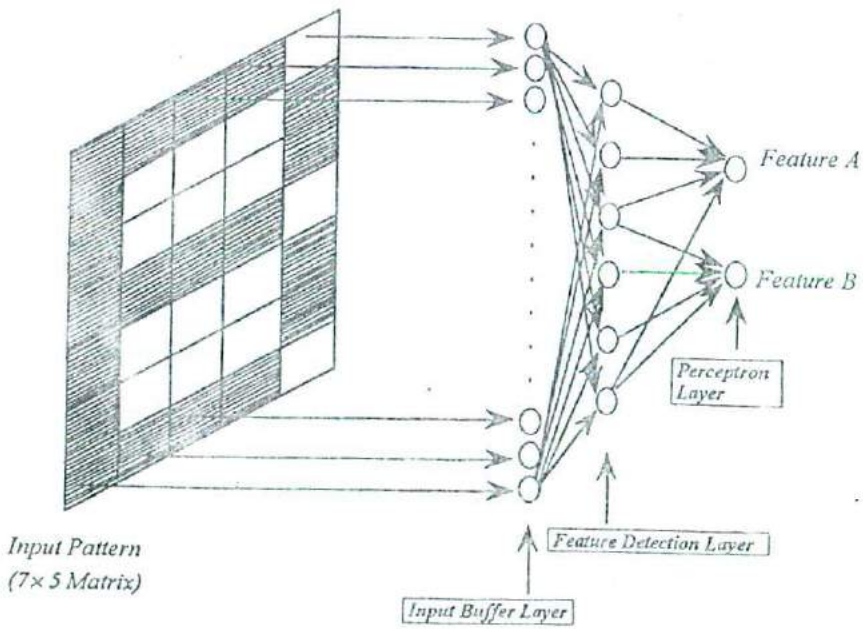


Figure 7.16 Diagram of the perceptron.

McCulloch–Pitts model and incorporated Hebbian learning, which he summarized as follows:

> When the synaptic input and the neuron output were both active, the strength of the connection is enhanced.

The perceptron was a pattern classification system that could identify both abstract and geometrical patterns. The first perceptron was primarily an optical system that had a grid of 400 photocells connected to associator units in the input buffer, which collected electrical impulses from the photocells. The photocells were randomly connected to the associators and received optical stimuli. The output of the sensors were connected to a hard-wired genetically predetermined set of processing elements (called demons) that recognized particular types of patterns. The output of each demon was connected to a threshold logic unit which had no output until a certain level and type of input was received. Then the output rose linearly with the input. This concept was inspired by the observation that the neuron does not fire until the balance of input activity exceeds some threshold, and that the firing rate is increased in proportion to certain characteristics of the input. It was quite robust and capable of some learning, it possessed a great deal of plasticity (i.e., information could be retained after some of the cells had been destroyed), and it was capable of making limited generalizations. It could properly categorize patterns despite noise in the input.

Rosenblatt studied both the two-layer and three-layer perceptrons. He was able to prove that the two-layer perceptron could separate inputs into two classes only if the two classes were linearly separable. In some systems, supervised learning was used in which the weights were adjusted in proportion to the error between the desired and actual output. While his attempts to extend the learning procedure to the three-layer perceptrons were encouraging, he could not find a mathematical basis for distributing credit (or blame) for the output errors between the two layers of weights. Hence, there was no mathematical basis for making corrections to the weighting functions. Actually Amari had solved the credit assignment problem in 1967, but it went unnoticed, because it was published in the Japanese literature. Had Amari's work been more widely known, it could have mitigated the impact of the critical book entitled *Perceptrons* by Minsky and Papert discussed later in this section.

The perceptron paradigm was designed to explain and model the pattern recognition capabilities of the visual system. The perceptron was a feedforward network without any feedback, without connections between neurons in the same layer, and without any randomness about the operation of the network. It was basically a three-layer network in which the input layer was a buffer (fanout) layer that mapped a rectangular pixelized sensor pattern to a linear array. The second layer, consisting of a set of feature detectors or feature demons, was either fully or randomly connected to the input layer.

This layer used either linear or nonlinear threshold activation functions to condition the outputs. The output layer contained "pattern recognizers" or "perceptrons." The weights of the inputs to the second layer were randomized and then fixed while the weights of the output layer were "trainable." The artificial neurons in the output or perceptron layer each had an input tied to a bias with a value of +1. The activation functions on the neurons in the output layer sometimes were "threshold-linear" functions in which the output signal is zero until the sum of the weighted inputs becomes positive, at which time the output increased to the weighted summation of the inputs. An alternate activation function sometimes used was a threshold function in which the output was zero if the weighted sum was zero or negative and equal to one if the weighted summation of the input was positive.

The basic learning algorithm procedure for training the perceptron is as follows:

- If the output is correct, leave the weights unchanged.
- If the output should be 1 but is instead 0, increment the weights on the active input lines (an active input line is defined as one that has a positive input).
- If the output should be 0 but is instead 1, decrement the weights on the active input lines.

The amount that the weights were changed depended upon the learning scheme that has been chosen. The three basic types of learning used in the perceptrons were as follows:

- A fixed increment or decrement.
- A variable amount of increment or decrement based upon the error (defined as the difference between the weighted sum and the desired output).
- A combination of both a fixed increment and an increment proportional to the error.

To classify a wide variety of shapes, the number of feature neurons must be quite large. By selective use of feedback, it is possible to radically reduce the number of neurons required. Another scheme used with the perceptron involved (a) the segmentation of the image into smaller pieces and (b) the creation of neurons that were specific to particular areas.

Minsky and Papert's Perceptrons

In the mid-1960s, Marvin Minsky began studying the "limitations" of the perceptrons, because of concern that Rosenblatt was making claims that were not being substantiated. (Fierce competition between Minsky and Rosenblatt

is alleged to have extended back to the time when both were students at the Bronx High School of Science, which was probably the top technical high school in the United States at that time.) He and Seymour Papert showed that the two-layer perceptron was rather limited because it could only work problems with a linearly separable solution space. The exclusive-or (*XOR*) problem was cited as an elementary system that the perceptron was unable to solve. They emphasized the inability of the perceptron to assign credit for the errors to the different layers of weights. After their book entitled *Perceptrons* was published in the late 1960s, virtually all support for research in the neural networks field was ended by the various U.S. funding agencies.

A quotation from *Perceptrons* is indicative of the nature of the criticism by Minsky and Papert.

> The perceptron has shown itself to be worthy of study despite (and even because of!) its severe limitations. It has many features to attract attention: its linearity; its intriguing learning theorem; its clear paradigmatic simplicity as a kind of parallel computer. There is no reason to suppose that any of these virtues carry over to the many layered version. Nevertheless, we consider it an important research problem to elucidate (or reject) our intuitive judgment that the extension is sterile. Perhaps some power conversion theorem will be discovered, or some profound reason for the failure to produce an interesting 'learning theorem' for the multi-layered machine will be found.

The criticism in *Perceptrons*, while generally fair in the contexts of the state of knowledge at that time, was absolutely wrong in one respect. The virtues cited by Minsky and Papert for the two-layer network indeed did carry over to the many-layered version, and in fact a three-layer perceptron was capable of separating linearly inseparable variables, including the XOR problem. Rosenblatt died in a boat accident shortly after publication of *Perceptrons*, and unfortunately, the criticism of Minsky and Papert was never properly refuted at that time.

Widrow's Adaline

Adaline (*ada*ptive *lin*ear *e*lement) is a neural network that adapts a system to minimize the "error" signal using supervised learning. It acts as a filter to sort input data patterns into two categories. Up until the last decade, it was perhaps the most successful application of neural networks, because it is used in virtually all high-speed modems and telephone switching systems to cancel out the echo of a reflected signal in a transmission line or corridor. It was invented by Bernard Widrow and M. E. (Ted) Hoff of Stanford University in the early 1960s. (Hoff is also generally credited with being the inventor of the microprocessor as we know it today and was the founder of Intel Corporation.)

The basic design of the Adaline is shown in Figure 7.17. This arrangement is substantially the same as that for the perceptron discussed earlier, with the

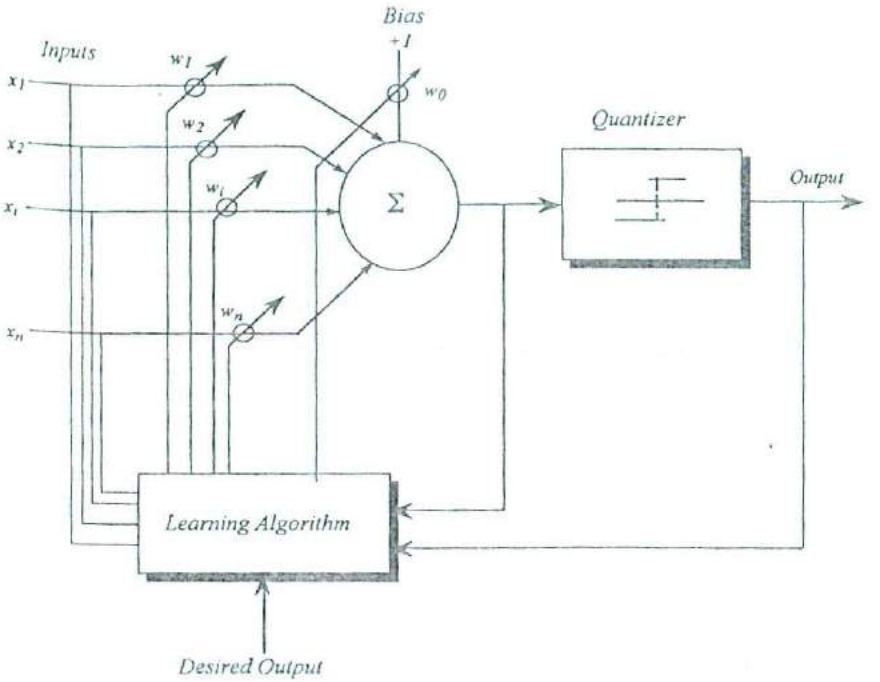


Figure 7.17 Diagram of an Adaline processing element.

expectation that the quantizer is a threshold-type nonlinear function with +1 and −1 as the limiting values (i.e., if the summation of the weighted inputs is positive, the output of the system will be +1; and if it is zero or negative, the output will be −1). The learning algorithm uses the difference between the desired output and the output of the summation (not the output of the quantizer) to produce the error ϵ function used to adjust the weights. In contrast, Rosenblatt used the difference between the actual output of the system and the desired output as the error functions to adjust the weights in the perceptron.

Prior to the beginning of the training, all weights must be adjusted to random values. With the Adaline, an input pattern is presented to the processing elements that filters it for a specific category. If the input matches the category, the processing element output is +1, and if it does not match the category, the processing element output is −1. The learning rule is the "Delta Rule," also known as the "Widrow–Hoff Learning Rule," which is, in fact, a least mean squares minimization of error procedure and is discussed extensively in Section 8.2. It involves the adjustment of weights according to

the error in the processing element, computing a "delta" vector

$$\Delta w_i = \frac{\eta \cdot \epsilon \cdot x_i}{|\mathbf{X}|^2} \tag{7.7-1}$$

where η is the learning constant, ϵ is the error, x_i is the ith input (-1, or $+1$), and $\mathbf{X}$ is the input vector. This Widrow–Hoff learning rule is discussed in Section 8.2 of the next chapter.

The learning algorithm for Adaline involves the application of the input (which may be noisy) to the single processing element or neuron. The application of the desired output and a computation of the error, defined as the difference between the weighted sum prior to the quantizer and the desired output, provides the input to the learning module. Then each weight is adjusted so that the error is equally distributed among the weights (including the weight for the bias). Equation (7.7-1) becomes

$$\Delta w_i = \frac{\eta \cdot \epsilon \cdot x_i}{|\mathbf{X}|^2} = \frac{\epsilon \cdot x_i}{(N+1)|\mathbf{X}|^2} \tag{7.7-2}$$

where $(N + 1)$ is the number of inputs plus the bias input and $1/(N + 1)$ replaces the learning constant η. This means that the error is uniformly assigned to the $(N + 1)$ inputs.

Since all inputs are $+1$ or -1, this adds or subtracts a fixed amount to the weight for each element input, depending on the sign. This process is repeated over and over again for each set of inputs in the epoch, and the epochs are repeated until the error is reduced to the desired value. Since both the desired output and actual inputs are binary, it is possible to have complete agreement between the desired outputs and the quantizer output even though the error (the difference between the desired value and the neuron output before the quantizer) has a substantial value. Further training beyond the time when the quantizer output is equal to the desired output is performed because the minimization of the error makes the system more tolerant of noise fluctuations in the input signals. This algorithm has been shown to guarantee convergence, provided that a set of weights exist that will minimize the error in a least squares sense.

Most of the time, the convergence of the learning process in the Adaline is very fast. However, the nature of the initial randomization can have a major effect on the speed of convergence. In a limited number of cases, convergence will not occur at all. Some of the real-world problems dealing with the Adaline occur where input patterns may not be perfect examples of the categories they represent. For instance, suppose we consider separating "circle" from "noncircle." The pertinent question is, How perfect does the circle have to be before it is considered a circle; or, alternately, How much deviation from a perfect circle is necessary before a figure is considered as a

noncircle? Another restriction associated with the Adaline, as it was originally conceived, is that it is capable of classifying only linearly separable patterns. Later versions involving multilayers of Adalines proved more powerful and capable of separating input space even though the variables were not linearly separable.

One of the major applications by Widrow of the Adaline is in adaptive noise reduction. Every telephone has different transfer characteristics which can change during a single transmission. The use of an adaptive network to adjust the input signals spectrum so as to keep the *signal-to-noise-ratio* high for the given state of the line was one of the early applications. Other applications of the Adaline by Widrow and his students at Stanford University include: (1) adaptive antenna arrays, (2) adaptive blood pressure regulation, (3) adaptive filtering, (4) seismic signal pattern recognition, (5) weather forecasting, (6) long-distance and satellite telephone adaptive echo cancellation, (7) cancellation of correlated interference in acoustical and electronic instruments, (8) separation of a fetal heartbeat from its mother's heartbeat, and (9) signal equalization in all high-speed modems in use today.

Widrow's Madaline

A Madaline (which is an acronym for "Many Adalines") involves the use of several Adalines as the middle layer of a three-layer neural network. The input layer, as in the case of the perceptron, is an input buffer to ensure that all inputs go to each of the Adalines, and the output layer is a single unit that combines the outputs of all Adalines in a prescribed way. Sometimes this output unit gives a $+1$ when the majority of the inputs are $+1$, and a -1 when they are not (i.e., voting majority). In other cases, it will give a $+1$ only when all of the output of all Adaline's are $+1$ (an "*AND*" output). In another situation, the output unit will give a $+1$ when any of the outputs of the Adaline are $+1$ (an "*OR*" output).

Since Madaline has a binary output, it can only be used to discriminate between two classes. It is possible of course to use many independent Madalines to discriminate between more than two classes. One Madaline is needed for each pair of classes added. Typically, the final classification is the class that constitutes the most outputs. A typical Madaline network architecture is shown in Figure 7.18. The Madaline combiner unit does not have a bias input, and the weights on the input to the Madaline unit are fixed; that is, they are adjusted initially to represent the importance of the specific Adaline output. Here again the stability of the Madaline relates to the stability of individual Adalines, and convergence is seldom a problem. Adaline elements in a Madaline network evolve as detectors for a specific input features. This is particularly useful when the Madaline is used in control systems.

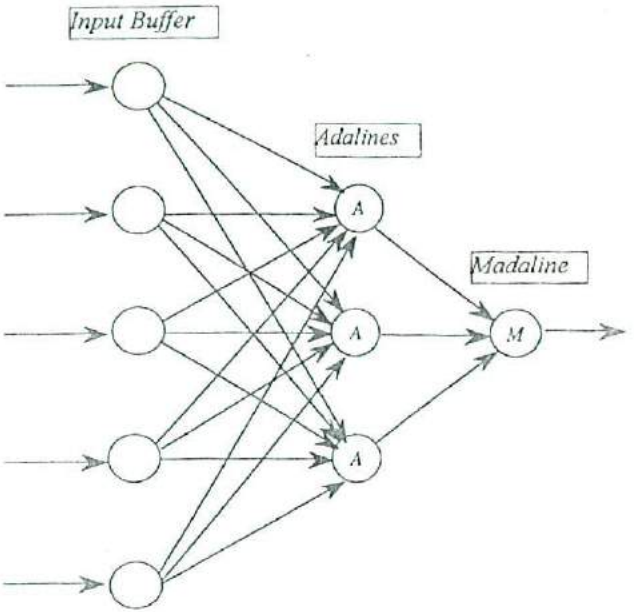


Figure 7.18 Diagram of a Madaline network architecture.

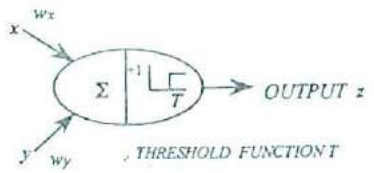
7.8 SEPARATION OF NONLINEARLY SEPARABLE VARIABLES

The ability of an Adaline to separate linearly separable variables can be readily demonstrated with a processing element or neuron that involves two inputs (x and y) and two weights (w_x and w_y). The neuron shown in Figure 7.19*a* sums the two weighted inputs—that is,

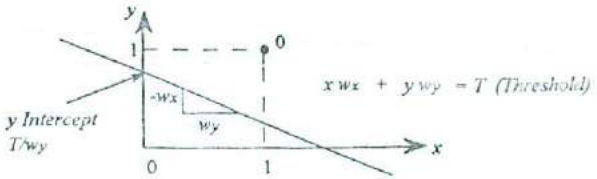
$$I = xw_x + yw_y \tag{7.8-1}$$

The output z is equal to 1 if the sum is greater than the threshold value T, and it is equal to 0 (or -1 in an Adaline) if the sum is less than or equal to the threshold value. A special case occurs when the output is equal to the threshold T (i.e., the case that divides the two regions). Equation (7.8-1) becomes

$$xw_x + yw_y = T \tag{7.8-2}$$



(a)



$x \cdot w_x + y \cdot w_y > T$ *System Fires* ($z = 1$)

$x \cdot w_x + y \cdot w_y < T$ *System does not Fire* ($z = 0$)

$x \cdot w_x + y \cdot w_y = T$ *Dividing Line*

$$y = -\frac{w_x}{w_y}x + \frac{T}{w_y}$$

(b)

Figure 7.19 Separation of variables by a neural network: (*a*) Processing element with two inputs. (*b*) Division of *x*–*y* plane by a processing element.

which can be rearranged to

$$y = \left[-\frac{w_x}{w_y}\right]x + \left[\frac{T}{w_y}\right] \tag{7.8-3}$$

This is the equation of a straight line where the slope is equal to $[-w_x/w_y]$ and the y intercept is $[T/w_y]$, which divides the plane into values that are below the threshold and above the threshold as shown in Figure 7.19*b*.

This concept can be extended further with a three-layer network as shown in Figure 7.20, where the first layer is a buffer with two inputs x and y; the middle layer has two neurons fully connected to the buffer layer with weights on each connection. The output layer is a single processing element whose input weights are set at 0.5 and whose threshold is set at 0.75. This configuration represents a logic "and" function, where both processing elements in the middle layer must produce a 1 to give a 1 in the output layer. The two processing elements in the middle layer are threshold functions with

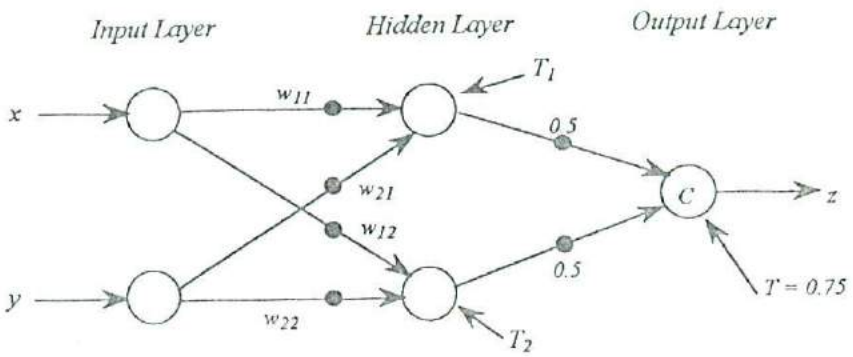


Figure 7.20 Three-layer network for separating nonlinearly separable variables.

thresholds T_1 and T_2. Hence, the outputs are either 0 or 1, depending upon whether the summation of the weighted inputs is less than or greater than the threshold values. As in the previous paragraphs, each of these threshold values effectively allow the plane to be divided.

The equations for the cases where the outputs of the two middle layer neurons are equal to the threshold values T_1 and T_2 are

$$xw_{11} + yw_{21} = T_1 \tag{7.8-4}$$

$$xw_{12} + yw_{22} = T_2 \tag{7.8-5}$$

which can be rearranged into the classical equation form for a straight line:

$$y = -\frac{w_{11}}{w_{21}}x + \frac{T_1}{w_{21}} \tag{7.8-6}$$

$$y = -\frac{w_{12}}{w_{22}}x + \frac{T_2}{w_{22}} \tag{7.8-7}$$

It is readily apparent that the use of two processing elements in the hidden layer provides for a double division of the plane as shown in Figure 7.21. The location and orientation of these two lines are determined by six quantities, namely, the values of the four weights and the two thresholds. Since only four parameters are needed to define these two lines unambiguously, there is a wide range of values of weights and threshold values that will define any two particular lines.

Since the outputs of the two neurons in the middle layer are either 0 or 1, it is apparent that both outputs must be 1 if the output layer is to produce a 1. It is readily shown that this corresponds to coordinates x and y being located in only one particular "quadrant" produced by these two intersecting

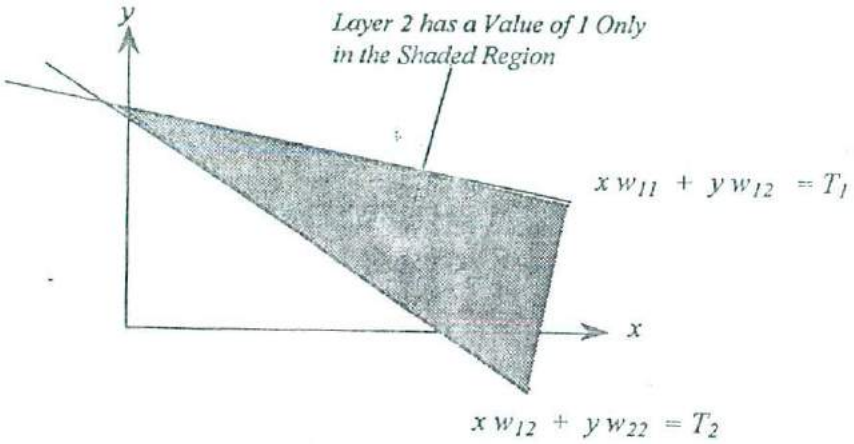


Figure 7.21 Division of x–y plane by two neurons.

lines. Hence, only one "quadrant" of this area will have a value of 1 while the other three "quadrants" will have a value of 0.

The use of three hidden processing elements in the middle layer further subdivides the plane with three lines, producing a triangular closed region. Writing the equations for the outputs of the processing elements in the middle layer of the neural networks shown in Figure 7.22 and setting them equal to the thresholds give the three dividing lines:

$$v_3 = xw_{13} + yw_{23} = T_3 \tag{7.8-8}$$

$$v_4 = xw_{14} + yw_{24} = T_4 \tag{7.8-9}$$

$$v_5 = xw_{15} + yw_{25} = T_5 \tag{7.8-10}$$

where T_3, T_4, and T_5 are the thresholds. Again, these equations can be put in the classical form for a straight line where the coefficient of the x terms are the slopes of the three lines and the three constant terms involving thresholds T_3, T_4, and T_5 are the y intercepts of the three straight lines:

$$y = -\frac{w_{13}}{w_{23}}x + \frac{T_3}{w_{23}} \tag{7.8-11}$$

$$y = -\frac{w_{14}}{w_{24}}x + \frac{T_4}{w_{24}} \tag{7.8-12}$$

$$y = -\frac{w_{15}}{w_{25}}x + \frac{T_5}{w_{25}} \tag{7.8-13}$$

which are shown in Figure 7.23.

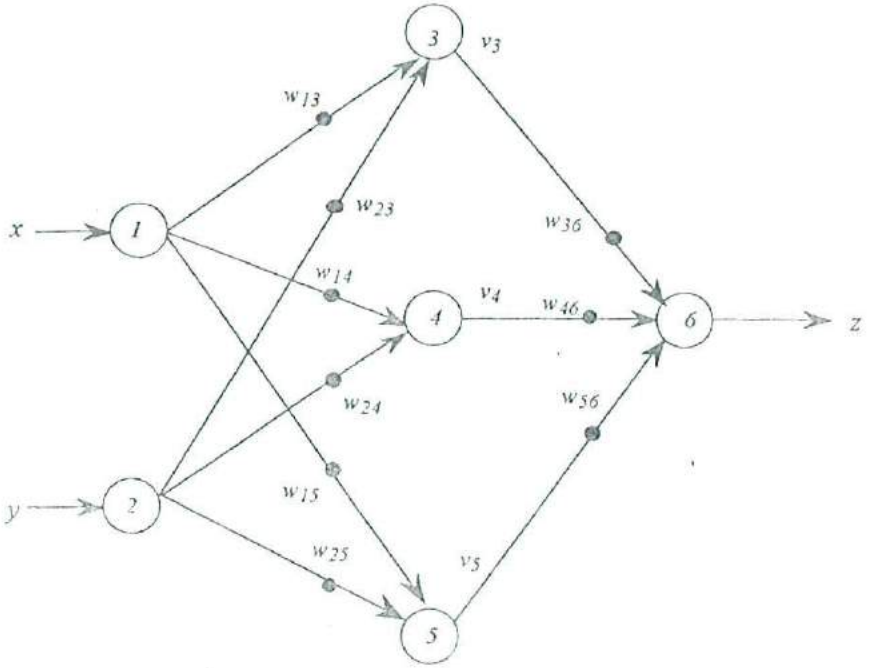


Figure 7.22 Three-layer network with three neurons in the hidden layer divides the plane with three lines.

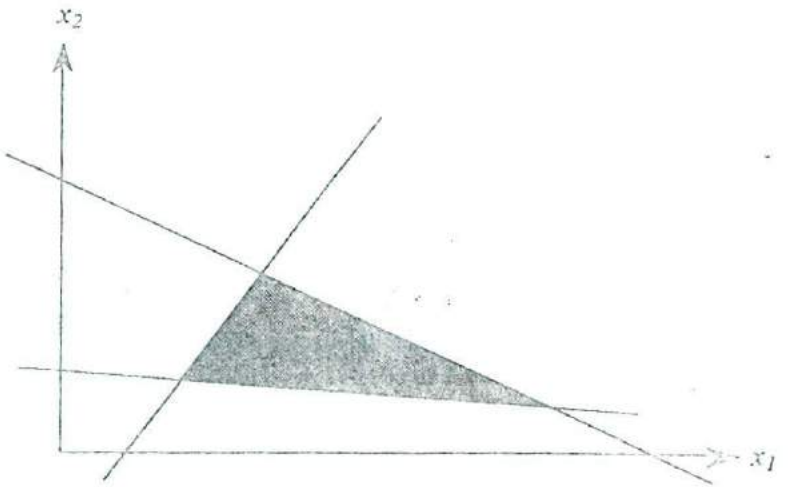
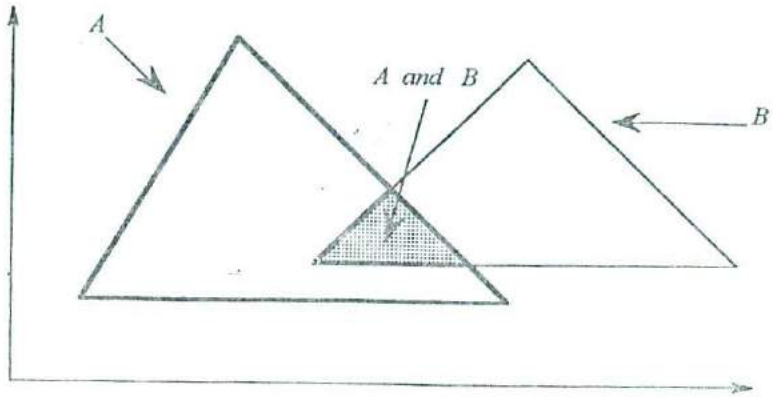
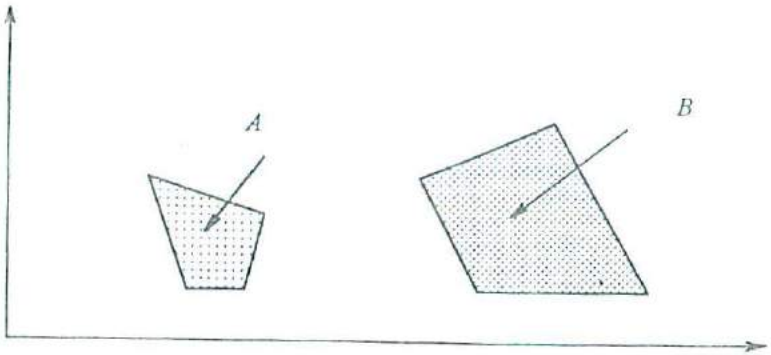


Figure 7.23 Plane separated by three lines to provide a closed triangular region.



(a)



(b)

Figure 7.24 Areas enclosed by use of many neurons: (*a*) Closed triangular area with re-entrant area created by six lines (six nodes in middle layer of Figure 7.22). (*b*) Separate closed areas created by eight lines (eight nodes in middle layer of Figure 7.22).

It is readily apparent that the use of additional processing elements in the hidden layer allows us to generate virtually any type of enclosed area desired, ranging from an approximation of a circle to a convex polygon with re-entrant regions as shown in Figure 7.24*a*. Indeed, not all of the outputs of the middle layer need to overlap, and hence it is possible to use such an artificial neural network to enclose multiple regions as shown in Figure 7.24*b*.

REFERENCES

Caudill, M., and Butler, C., *Naturally Intelligent Systems*, MIT Press, Cambridge, MA, 1989.

Caudill, M., and Butler, C., *Understanding Neural Networks: Computer Explorations*, Vols. 1 and 2, MIT Press, Cambridge, MA, 1992.

DARPA (Defense Advanced Research Projects Agency), *DARPA Neural Network Study*, AFCEA International Press, Fairfax, VA, 1988.

Hebb, D., *Organization of Behavior*, John Wiley, New York, NY, 1949.

Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesley, Reading, MA, 1989.

Maren, A. J., Harston, C. T., and Pap, R. M., *Handbook of Neural Computing Applications*, Academic Press, New York, 1990.

Miller, W. T., Sutton, R. S., and Werbos, P. J., *Neural Networks for Control*, MIT Press, Cambridge, MA, 1990.

Nelson, M. M., and Illingsworth, W. T., *A Practical Guide to Neural Networks*, Addison-Wesley, Reading, MA, 1990.

Pao, Y. H., *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley, Reading, PA, 1989.

Simpson, P., *Artificial Neural Systems*, Pergamon, Elmsford, NY, 1990.

Wasserman, P. D., *Neural Computing: Theory and Practice*, Van Nostrand Reinhold, New York, 1989.

Wasserman, P. D., *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, New York, NY, 1993.

White, D. A., and Sofge, D. A., *Handbook of Intelligent Control*, Van Nostrand Reinhold, New York, 1992.

PROBLEMS

1. In the linear associator of Figure 7.8, the input vector x is a three component vector $(0.3, -0.7, 0.2)$ and the output vector is a four component vector $(-0.8, -0.3, 0.6, 0.9)$. Calculate the weights.

2. Discuss the differences between Widrow's Adaline and Madaline networks and Rosenblatt's perceptron. How do they differ as far as error input is concerned?

3. The three-layer network of Figure 7.22 divides the plane with three lines forming a triangle. Calculate the weights that will give a triangle with its vertices at (x, y) coordinates $(0, 0)$, $(1, 3)$, and $(3, 1)$.

4. Design a three-layer network as shown in Figure 7.20 to separate the non-linearly separable variables for the "exclusive-nor" function having the

following truth table:

		Input x	
		0	1
Input y	0	1	0
	1	0	1

5. The five-bit code for the letter Q is 01011. Develop a storage matrix W and correct it so that chaotic oscillations will not occur. Show that this storage matrix can produce a correct memory state, even when an erroneous code for q is applied. Use the erroneous representation of Q to be 01010. Show all steps involved.

6. A weight matrix M is given by

$$M = \begin{vmatrix} 1 & 2 & 3 & 2 \\ -2 & 1 & 3 & -1 \\ 3 & 1 & 2 & -3 \end{vmatrix}$$

Draw a 2-layer neural network in which the given matrix represents the weights.

8

BACKPROPAGATION AND RELATED TRAINING ALGORITHMS

8.1 BACKPROPAGATION TRAINING

Backpropagation is a systematic method for training multiple (three or more)-layer artificial neural networks. The elucidation of this training algorithm in 1986 by Rumelhart, Hinton, and Williams (1986) was the key step in making neural networks practical in many real-world situations. However, Rumelhart, Hinton, and Williams were not the first to develop the backpropagation algorithm. It was developed independently by Parker (1982) in 1982 and earlier by Werbos (1974) in 1974 as part of his Ph.D. dissertation at Harvard University. Nevertheless, the backpropagation algorithm was critical to the advances in neural networks because of the limitations of the one- and two-layer networks discussed previously. Indeed, backpropagation played a critically important role in the resurgence of the neural network field in the mid-1980s. Today, it is estimated that 80% of all applications utilize this backpropagation algorithm in one form or another. In spite of its limitations, backpropagation has dramatically expanded the range of problems to which neural network can be applied, perhaps because it has a strong mathematical foundation.

Prior to the development of backpropagation, attempts to use perceptrons with more than one layer of weights were frustrated by what was called the "weight assignment problem" [i.e., how do you allocate the error at the output layer between the two (or more) layers of weights when there is no firm mathematical foundation for doing so?]. This problem plagued the neural network field for over two decades and was cited by Minsky and Papert as one of the criticisms of multilayer perceptrons. Ironically, this need not have been the case, because Amari developed a method for allocating

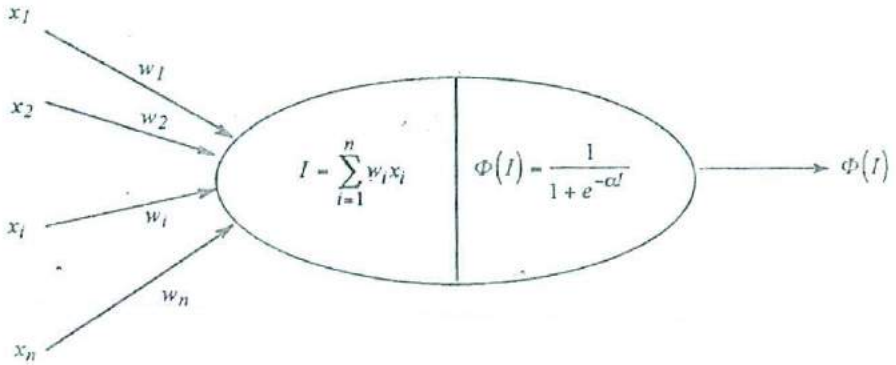


Figure 8.1 Diagram of a neuron.

weights in the 1960s that was not widely disseminated (Amari, 1972). Even more ironic is the fact that Rosenblatt's method of using a random distribution of the weight values in the middle neuron layer and adjusting only the weights for the output neuron layer has been shown to provide adequate training of the network in most cases.

Let us consider a typical neuron as shown in Figure 8.1, with inputs x_i weights w_i, a summation function in the left half of the neuron, and a nonlinear activation function in the right half. The summation of the weighted inputs designated by I is given by

$$I = x_1 w_1 + x_2 w_2 + \cdots + x_n w_n = \sum_{i=1}^{n} x_i w_i \tag{8.1-1}$$

The nonlinear activation function used is the typical sigmoidal function and is given by

$$\Phi(I) = \frac{1}{(1 + e^{-\alpha I})} = (1 + e^{-\alpha I})^{-1} \tag{8.1-2}$$

This function is, in fact, the logistic function, one of several sigmoidal functions which monotonically increase from a lower limit (0 or -1) to an upper limit ($+1$) as I increases. A plot of a logistic function is shown in Figure 8.2*a*, in which values vary between 0 and 1, with a value of 0.5 when I is zero. An examination of this figure shows that the derivative (slope) of the curve asymptotically approaches zero as the input I approaches minus infinity and plus infinity, and it reaches a maximum value of $\alpha/4$ when I equals zero as shown in Figure 8.2*b*. Since this derivative function will be utilized in backpropagation, let us reduce it to its most simple form. If we

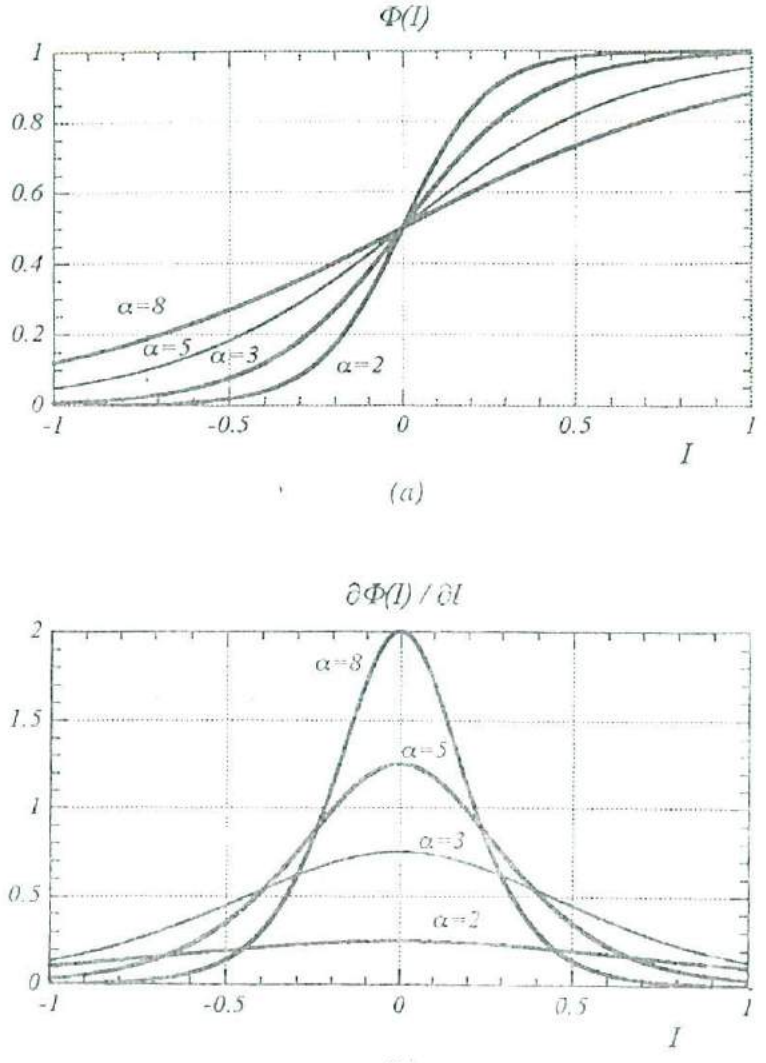


Figure 8.2 (*a*) Logistic activation function ($\alpha = 2, 3, 5$, and 8) and (*b*) its first derivative (slope).

take a derivative of equation (8.1-2), we get

$$\frac{\partial \Phi(I)}{\partial I} = (-1)(1 + e^{-\alpha I})^{-2} e^{-\alpha I}(-\alpha)$$

$$= \alpha e^{-\alpha I}(1 + e^{-\alpha I})^{-2} = \alpha e^{-\alpha I}\Phi^2(I) \qquad (8.1\text{-}3)$$

If we solve equation (8.1-2) for $e^{-\alpha I}$, substitute it into equation (8.1-3), and simplify, we get

$$\frac{\partial \Phi(I)}{\partial I} = \alpha \frac{1 - \Phi(I)}{\Phi(I)} \Phi^2(I) = \{\alpha[1 - \Phi(I)]\Phi(I)\} = \alpha(1 - \Phi)\Phi \tag{8.1-4}$$

where $\Phi(I)$ has been simplified to Φ by dropping (I).

It is important to point out that multilayer networks have greater representational power than single-layer networks only if nonlinearities are introduced. The logistic function (also called the "squashing" function) provides the needed nonlinearity. However, in the use of the backpropagation algorithm, any nonlinear function can be used if it is everywhere differentiable and monotonically increasing with I. Sigmoidal functions, including logistic, hyperbolic tangent, and arctangent functions, meet these requirements. The arctangent function, denoted as $\tan^{-1}$, has the form

$$\Phi(I) = \frac{2}{\pi} \tan^{-1}(\alpha I) \tag{8.1-5}$$

where the factor $2/\pi$ reduces the amplitude of the arctangent function so that it is restricted to the range -1 to $+1$. The constant α determines the rate at which the function changes between the limits of -1 and $+1$ and to the slope of the function at the origin is $2\alpha/\pi$. It influences the shape the arctangent function in the same way that α influences the logistic function in Figure 8.2*a*. The arctangent function has the same sigmoidal shape as shown in Figure 8.3*a*. The derivative is

$$\frac{\partial \Phi(I)}{\partial I} = \frac{2}{\pi}\left[\frac{\alpha}{1 + \alpha^2 I^2}\right] \tag{8.1-6}$$

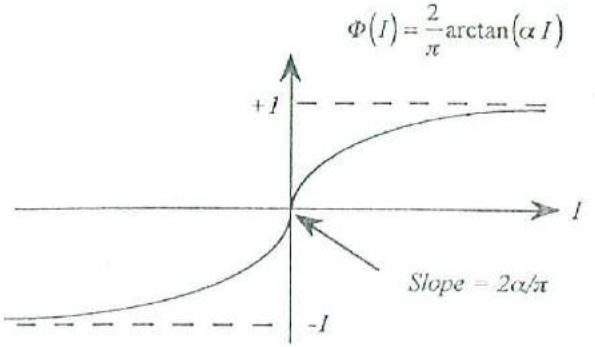
which would be used in place of equation (8.1-4) if the arctangent replaced the logistic activation function.

The hyperbolic tangent function has the form

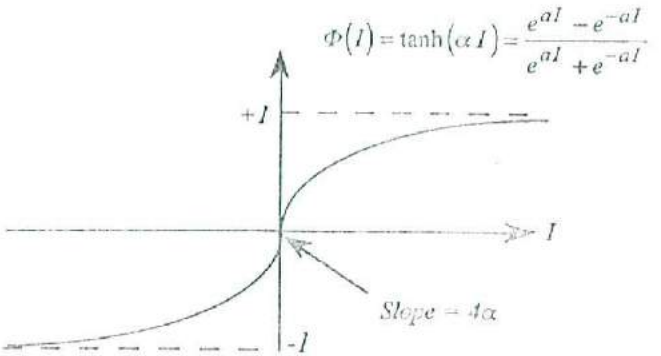
$$\Phi(I) = \tanh(\alpha I) = \frac{e^{\alpha I} - e^{-\alpha I}}{e^{\alpha I} + e^{-\alpha I}} \tag{8.1-7}$$

and its shape is shown in Figure 8.3*b*. Its derivative is

$$\frac{\partial \Phi(I)}{\partial I} = \alpha \operatorname{sech}^2(\alpha I) \tag{8.1-8}$$



(a)



(b)

Figure 8.3 Alternate activation functions for backpropagation: (*a*) Arctangent. (*b*) Hyperbolic tangent.

The slope of $\Phi(I)$ at the origin is 4α, and it determines the rate at which the function changes between the limits of -1 and $+1$ in the same general way that α influences the shape of the logistic function in Figure 8.2*a*.

The use of a sigmoidal (squashing) function provides a form of "automatic gain control"; that is, for small values of I near zero, the slope of the input–output curve is steep, producing a high gain, since all sigmoidal activation functions have derivatives with bell shapes of the type shown in Figure 8.2*b*. As the magnitude of I becomes greater in a positive or negative

direction, the gain decreases. Hence, large signals can be accommodated without saturation. This is shown in Figure 8.2*a*.

8.2 WIDROW–HOFF DELTA LEARNING RULE

The Widrow–Hoff delta learning rule can be derived by considering the node of Figure 8.4, where T is the target or desired value vector and I is defined by equation (8.1-1) as the dot product of the weight and input vectors and is given by

$$I = \sum_{i=1}^{n} w_i x_i \tag{8.2-1}$$

For this derivation, no quantizer or other nonlinear activation function is included, but the result presented here is equally valid when such nonlinear elements are included.

From Figure 8.4, we see the error function ε as a function of all weights w_i, and we see the squared error ε^2 to be

$$\varepsilon = (T - I) \tag{8.2-2}$$

$$\varepsilon^2 = (T - I)^2 \tag{8.2-3}$$

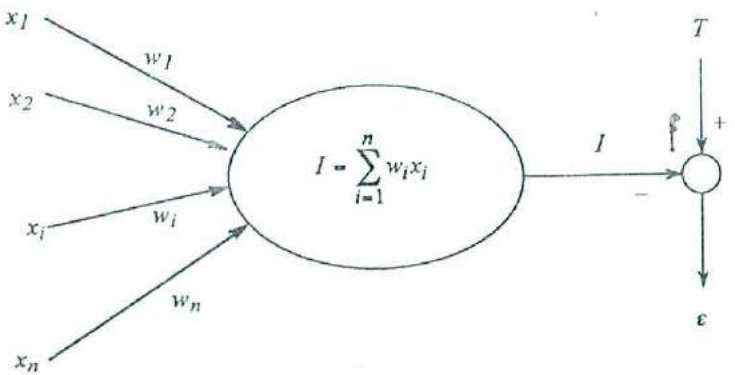


Figure 8.4 Neuron without activation function but with a target value T and an error ε.

The gradient of the square error vector is the partial derivatives with respect to each of these i weights:

$$\frac{\partial \varepsilon^2}{\partial w_i} = -2(T - I)\frac{\partial I}{\partial w_i} = -2(T - I)x_i \tag{8.2-4}$$

Since this gradient involves only the ith weight component, the summation of equation (8.2-1) disappears.

For demonstration purposes, let us consider a neuron with only two inputs, x_1 and x_2. The square error is now given by

$$\begin{aligned}\varepsilon^2 &= [T - w_1x_1 - w_2x_2]^2 \\ &= T^2 + w_1^2x_1^2 + w_2^2x_2^2 - 2Tw_1x_1 - 2Tw_2x_2 + 2w_1x_1w_2x_2 \\ &= w_1^2[x_1^2] + w_1[-2x_1(T - w_2x_2)] + [(T - w_2x_2)^2] \\ &= w_2^2[x_2^2] + w_2[-2x_2(T - w_1x_1)] + [(T - w_1x_1)^2]\end{aligned} \tag{8.2-5}$$

The minimum square error occurs when the partial derivatives of square error with respect to the weights w_1 and w_2 are set equal to zero:

$$\frac{\partial \varepsilon^2}{\partial w_1} = -2[T - w_1x_1 - w_2x_2]x_1 = 0 \tag{8.2-6}$$

$$\frac{\partial \varepsilon^2}{\partial w_2} = -2[T - w_1x_1 - w_2x_2]x_2 = 0 \tag{8.2-7}$$

Since x_1 and x_2 cannot be zero, the quantities in the brackets, which are identical for both equations, must be zero. This gives

$$T - w_1x_1 - w_2x_2 = 0 \tag{8.2-8}$$

from which the location of the minimum in the w_1 and w_2 dimensions are

$$w_1 = \frac{T - w_2x_2}{x_1} \tag{8.2-9}$$

$$w_2 = \frac{T - w_1x_1}{x_2} \tag{8.2-10}$$

Substitution of either of these values into equation (8.2-5) gives the minimum square error to be zero. Technically, this is correct, but in the real world the minimum square error is never equal to zero because of nonlinearities, noise,

and imperfect data. The presence of noise with a sigmoidal activation function will give a minimum square error that is not zero which we designate as $\varepsilon^2_{\min}$.

Examination of equation (8.2-5) shows that plots of ε^2 versus w_1 or w_2 will be parabolic in shape. The parabolic curve of squared error ε^2 versus w_1 is shown in Figure 8.5 for two cases of minimum square error: zero and $\varepsilon^2_{\min}$. For both cases, the minimum square error occurred at a value of w_1 given by equation (8.2-9). An identical result can be obtained for squared error versus w_2, where the minimum value occurs at the value of w_2 given by equation (8.2-10). Hence, the minimum square error surface for the two dimensional weight case is a paraboloid of revolution with the ε^2 axis located at (w_1, w_2).

A geometrical interpretation of the delta rule is that it involves a gradient descent algorithm to minimize the square error. When the square error is viewed in three dimensions $(w_1, w_2, \varepsilon^2)$ the square error surface is a paraboloid of revolution with the weight vector descending toward the minimum value along a gradient vector on the surface of the paraboloid. The projection of this gradient vector on the w_1–w_2 plane is the delta vector as shown in Figure 8.6. The delta rule moves the weight vector along the negative gradient of the curved surface toward the ideal weight vector

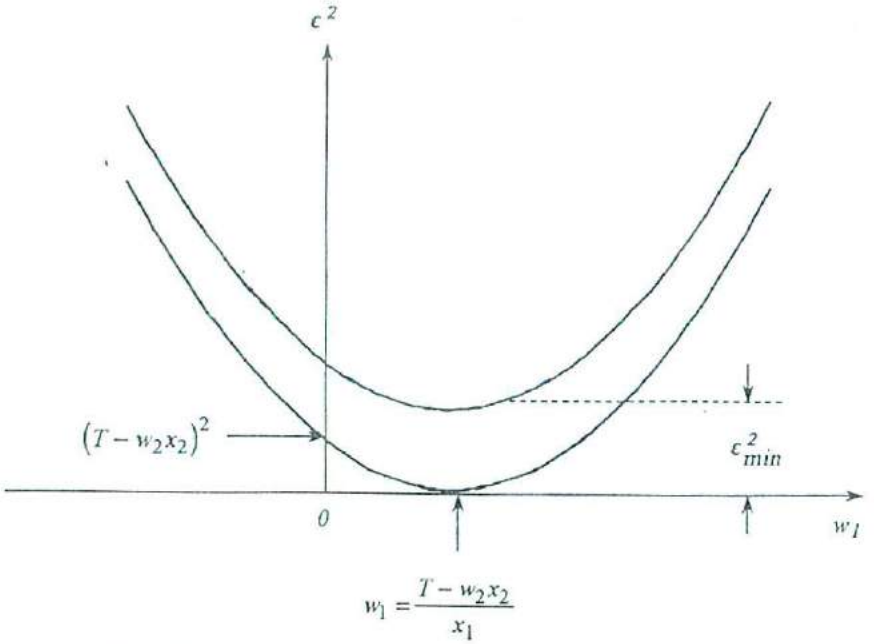


Figure 8.5 Minimization of square error during Widrow–Hoff training.

position. Because it follows the gradient, it is called a *gradient descent* or *steepest descent* algorithm. Since the gradient is the most efficient path to the bottom of the curved surface, the delta rule is the most efficient way to minimize the square error. There is, however, one caveat that must be added here: This statement is true only if the weight vector is descending toward a global minimum. If there are local minima, which are common with multidimensional problems, other techniques must be used to ensure that a solution (i.e., a weight configuration) is not trapped in one of these local minima.

The Widrow–Hoff delta training rule provides that the change in each weight vector component is proportional to the negative of its gradient:

$$\Delta w_i = -K\frac{\partial \varepsilon^2}{\partial w_i} = K \cdot 2(T - i)x_i = 2K\varepsilon x_i \qquad (8.2\text{-}11)$$

where K is a constant of proportionality. The negative sign is introduced because a minimization process is involved. It is common to normalize the

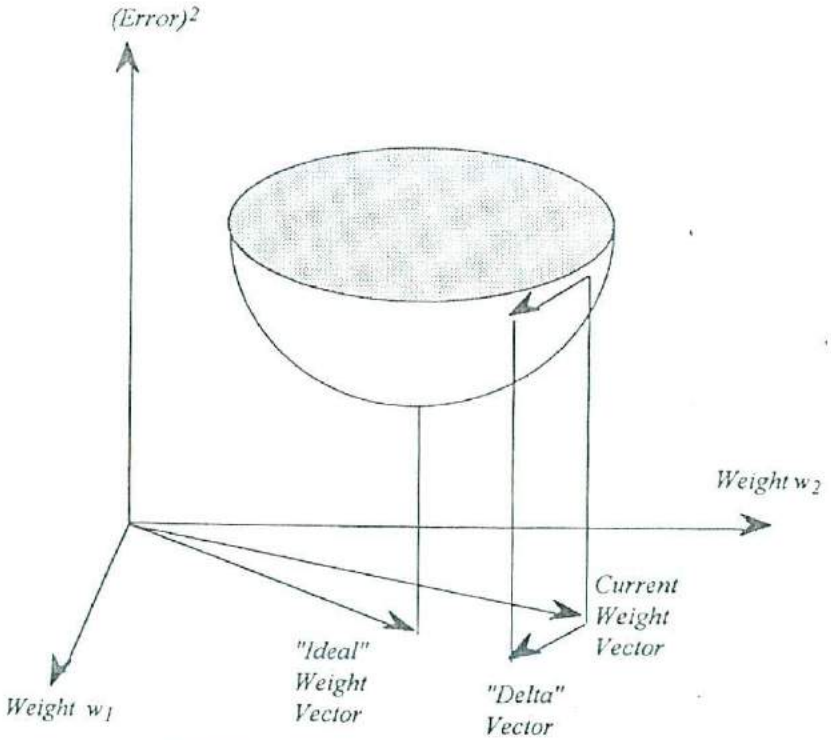


Figure 8.6 Geometric interpretation of delta rule.

input vector component x_i by dividing by $|\mathbf{X}|^2$. Equation (8.2-11) now becomes

$$\Delta w_i = \left[2K|\mathbf{X}|^2\right]\frac{\varepsilon x_i}{|\mathbf{X}|^2} = \frac{\eta \varepsilon x_i}{|\mathbf{X}|^2} \tag{8.2-12}$$

which agrees with equation (7.6-1) if we define the learning constant η to be equal to the terms in the brackets:

$$\eta = 2K|\mathbf{X}|^2 \tag{8.2-13}$$

8.3 BACKPROPAGATION TRAINING FOR A MULTILAYER NEURAL NETWORK[1]

Before discussing the details of the backpropagation process, let us consider the benefits of the middle layer(s) in an artificial neural network. A network with only two layers (input and output) can only represent the input with whatever representation already exists in the input data. Hence, if the data are discontinuous or nonlinearly separable, the innate representation is inconsistent, and the mapping cannot be learned. Adding a third (middle) layer to the artificial neural network allows it to develop its own internal representation of this mapping. Having this rich and complex internal representation capability allows the hierarchical network to learn any mapping, not just linearly separable ones.

Some guidance to the number of neurons in the hidden layer is given by Kolmogorov's theorem as it is applied to artificial neural networks. In any artificial neural network, the goal is to map any real vector of dimension m into any other real vector of dimension n. Let us assume that the input vectors are scaled to lie in the region from 0 to 1, but there are no constraints on the output vector. Then, Kolmogorov's theorem tells us that a three-layer neural network exists that can perform this mapping exactly (not an approximation) and that the input layer will have m neurons, the output layer will have n neurons, and the middle layer will have $2m + 1$ neurons. Hence, Kolmogorov's theorem guarantees that a three-layer artificial neural network will solve all nonlinearly separable problems. What it does not say is that (1) this network is the most efficient one for this mapping, (2) a smaller network cannot also perform this mapping, or (3) a simpler network cannot perform the mapping just as well. Unfortunately, it does not provide enough detail to find and build a network that efficiently performs the mapping we want. It does, however, guarantee that a method of mapping does exist in the form of an artificial neural network (Poggio and Girosi, 1990).

[1]The analysis presented here is the classical approach in which the hidden and output layer neurons have sigmoidal activation functions. An alternate approach in which the output neurons have linear activation functions is presented in Section 8.7.

Let us consider the three-layer network shown in Figure 8.7, where all activation functions are logistic functions. It is important to note that backpropagation can be applied to an artificial neural network with any number of hidden layers (Werbos, 1994). The training objective is to adjust the weights so that the application of a set of inputs produces the desired outputs. To accomplish this the network is usually trained with a large number of input–output pairs, which we also call examples.

The training procedure is as follows:

1. Randomize the weights to small random values (both positive and negative) to ensure that the network is not saturated by large values of weights. (If all weights start at equal values, and the desired performance requires unequal weights, the network would not train at all.)
2. Select a training pair from the training set.
3. Apply the input vector to network input.
4. Calculate the network output.
5. Calculate the error, the difference between the network output and the desired output.
6. Adjust the weights of the network in a way that minimizes this error. (This adjustment process is discussed later in this section.)

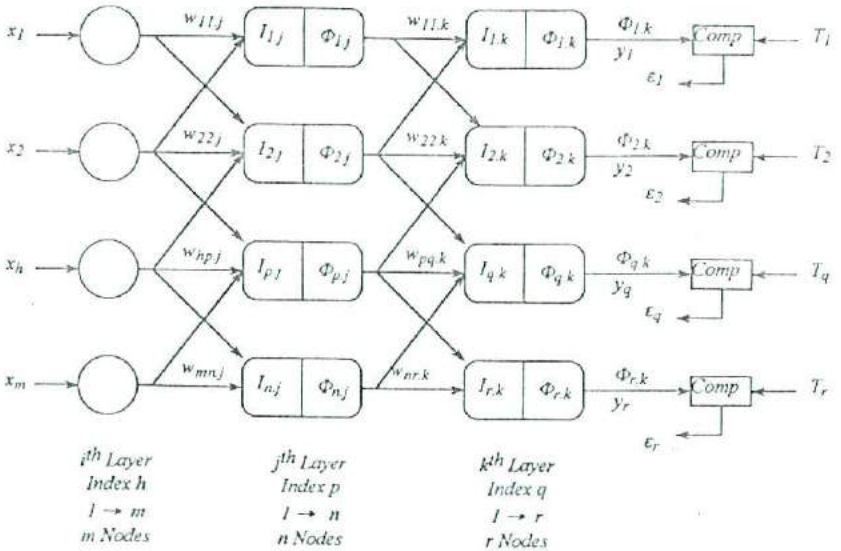


Figure 8.7 Sketch of multilayer neural network showing the symbols and indices used in deriving the backpropagation training algorithm.

7. Repeat steps 2–6 for each pair of input–output vectors in the training set until the error for the entire system is acceptably low.

Training of an artificial neural network involves two passes. In the forward pass the input signals propagate from the network input to the output. In the reverse pass, the calculated error signals propagate backward through the network, where they are used to adjust the weights. The calculation of the output is carried out, layer by layer, in the forward direction. The output of one layer is the input to the next layer. In the reverse pass, the weights of the output neuron layer are adjusted first since the target value of each output neuron is available to guide the adjustment of the associated weights, using the *delta rule*. Next, we adjust the weights of the middle layers. The problem is that the middle-layer neurons have no target values. Hence, the training is more complicated, because the error must be propagated back through the network, including the nonlinear functions, layer by layer.

Calculation of Weights for the Output-Layer Neurons

Let us consider the details of the backpropagation learning process for the weights of the output layer. Figure 8.8 is a representation of a train of neurons leading to the output layer designated by the subscript k with neurons p and q, outputs $\Phi_{p,j}(I)$ and $\Phi_{q,k}(I)$, input weights $w_{hp,j}$ and $w_{pq,k}$, and a target value T_q. The notation (I) in $\Phi_{q,k}(I)$ will be dropped for convenience. The output of the neuron in layer k is subtracted from its target value and squared to produce the square error signal, which for a layer k neuron is

$$\varepsilon = \varepsilon_q = \left[T_q - \Phi_{q,k}\right] \tag{8.3-1}$$

since only one output error is involved. Hence

$$\varepsilon^2 = \varepsilon_q^2 = \left[T_q - \Phi_{q,k}\right]^2 \tag{8.3-2}$$

The delta rule indicates that the change in a weight is proportional to the

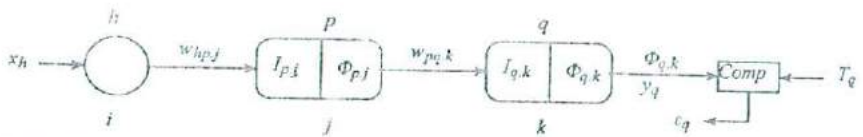


Figure 8.8 Representation of a train of neurons for calculating the change of weight for an output-layer neuron in backpropagation.

rate of change of the square error with respect to that weight—that is,

$$\Delta w_{pq,k} = -\eta_{p,q} \frac{\partial \varepsilon_q^2}{\partial w_{pq,k}} \tag{8.3-3}$$

where $\eta_{p,q}$ is constant of proportionality called *learning rate*. To evaluate this partial derivative, we use the chain rule of differentiation:

$$\frac{\partial \varepsilon_q^2}{\partial w_{pq,k}} = \frac{\partial \varepsilon_q^2}{\partial \Phi_{q,k}} \frac{\partial \Phi_{q,k}}{\partial I_{q,k}} \frac{\partial I_{q,k}}{\partial w_{pq,k}} \tag{8.3-4}$$

Each of these terms are evaluated in turn. The partial derivative of equation (8.3-2) with respect to $\Phi_{q,k}$ gives

$$\frac{\partial \varepsilon_q^2}{\partial \Phi_{q,k}} = -2\left[T_q - \Phi_{q,k}\right] \tag{8.3-5}$$

From equation (8.1-4), we get

$$\frac{\partial \Phi_{q,k}}{\partial I_{q,k}} = \alpha \Phi_{q,k}\left[1 - \Phi_{q,k}\right] \tag{8.3-6}$$

From Figure 8.7 we see that $I_{q,k}$ is the sum of the weighted inputs from the middle layer—that is,

$$I_{q,k} = \sum_{p=1}^{n} w_{pq,k} \Phi_{p,j} \tag{8.3-7}$$

Taking the partial derivative with respect to $w_{pq,k}$ gives

$$\frac{\partial I_{q,k}}{\partial w_{pq,k}} = \Phi_{p,j} \tag{8.3-8}$$

Since we are dealing with one weight, only one term of the summation of equation (8.3-7) survives. Substituting equations (8.3-5), (8.3-6), and (8.3-8) into equation (8.3-4) gives

$$\frac{\partial \varepsilon_q^2}{\partial w_{pq,k}} = -2\alpha\left[T_q - \Phi_{q,k}\right]\Phi_{q,k}\left[1 - \Phi_{q,k}\right]\Phi_{p,j} = -\delta_{pq,k}\Phi_{p,j} \tag{8.3-9}$$

where $\delta_{pq.k}$ is defined as

$$\delta_{pq.k} \equiv 2\alpha\left[T_q - \Phi_{q.k}\right]\Phi_{q.k}\left[1 - \Phi_{q.k}\right]$$
$$= 2\varepsilon_q \frac{\partial \Phi_{q.k}}{\partial I_{q.k}} \tag{8.3-10}$$

Substituting equations (8.3-9) into equation (8.3-3) gives

$$\Delta w_{pq.k} = -\eta_{p.q} \frac{\partial \varepsilon_q^2}{\partial w_{pq.k}} = -\eta_{p.q}\delta_{pq.k}\Phi_{p.j} \tag{8.3-11}$$

$$w_{pq.k}(N+1) = w_{pq.k}(N) - \eta_{p.q}\delta_{pq.k}\Phi_{p.j} \tag{8.3-12}$$

where N is the number of the iteration involved. An identical process is performed for each weight of the output layer to give the adjusted values of the weights. The error term $\delta_{pq.k}$ from equation (8.3-10) is used to adjust the weights of the output layer neurons using equation (8.3-11) and (8.3-12). It is useful to discuss why the derivative of the activation function is involved in this process. In equation (8.3-10) we have calculated an error which must be propagated back through the network. This error exists because the output neurons generate the wrong outputs. The reasons are (1) their own incorrect weights and (2) the middle-layer neurons generate the wrong output. To assign this blame, we backpropagate the errors for each output-layer neuron, using the same interconnections and weights as the middle layer used to transmit its outputs to the output layer.

When a weight between a middle-layer neuron and an output-layer neuron is large and the output layer neuron has a very large error, the weights of the middle layer neurons may be assigned a very large error, even if that neuron has a very small output and thus could not have contributed much to the output error. By applying the derivative of the squashing function, this error is moderated, and only small to moderate changes are made to the middle-layer weights because of the bell-shaped curve of the derivative function shown in Figure 8.2*b*.

Calculation of Weights for the Hidden Layer Neurons

Since the hidden layers have no target vectors, the problem of adjusting the weights of the hidden layers stymied workers in this field for years until backpropagation was put forth. Backpropagation trains hidden layers by propagating the adjusted error back through the network, layer by layer, adjusting the weight of each layer as it goes. The equations for the hidden layer are the same as for the output layer except that the error term $\delta_{hp.j}$ must be generated without a target vector. We must compute $\delta_{hp.j}$ for each

neuron in the middle layer that includes contributions from the errors in each neuron in the output layer to which it is connected. Let us consider a single neuron in the hidden layer just before the output layer, designated with the subscript p (see Figure 8.8). In the forward pass, this neuron propagates its output values to the q neurons in the output layer through the interconnecting weights $w_{pq.k}$. During training, these weights operate in reverse order, passing the value of $\delta_{pq.k}$ from the output layer back to the hidden layer. Each of these weights is multiplied by the value of the neuron through which it connects in the output layer. The value of $\delta_{hp.j}$ needed for the hidden-layer neuron is produced by summing all such products.

The arrangement in Figure 8.9 shows the errors that are backpropagated to produce the change in $w_{hp.j}$. Since all error terms of the output layer are involved, the partial derivative involves a summation over the r outputs. The procedure for calculating $\delta_{hp.j}$ is substantially the same as calculating $\delta_{pq.k}$. Let us start with the derivative of the square error with respect to the weight for the middle layer that is to be adjusted. Then, in a manner analogous to equation (8.3-3), the delta rule training gives

$$\Delta w_{hp.j} = -\eta_{h.p} \frac{\partial \varepsilon^2}{\partial w_{hp.j}} = -\eta_{h.p} \sum_{q=1}^{r} \frac{\partial \varepsilon_q^2}{\partial w_{hp.j}} \tag{8.3-13}$$

where the total mean square ε^2 is now defined by

$$\varepsilon^2 = \sum_{q=1}^{r} \varepsilon_q^2 = \sum_{q=1}^{r} \left[T_q - \Phi_{q.k}\right]^2 \tag{8.3-14}$$

since several output errors may be involved. The learning constant $\eta_{h.p}$ is

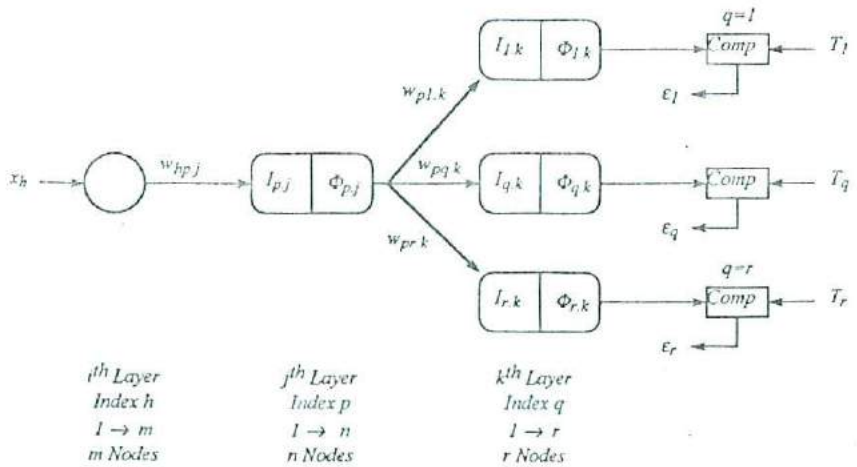


Figure 8.9 Representation of a train of neurons for calculating the change of weight for a middle (hidden) layer neuron in backpropagation.

usually, but not necessarily, equal to $\eta_{p,q}$. Again, we can evaluate the last term of equation (8.3-13) using the chain rule of differentiation, which gives

$$\frac{\partial \varepsilon^2}{\partial w_{hp,j}} = \sum_{q=1}^{r} \frac{\partial \varepsilon_q^2}{\partial \Phi_{q,k}} \frac{\partial \Phi_{q,k}}{\partial I_{q,k}} \frac{\partial I_{q,k}}{\partial \Phi_{p,j}} \frac{\partial \Phi_{p,j}}{\partial I_{p,j}} \frac{\partial I_{p,j}}{\partial w_{hp,j}} \tag{8.3-15}$$

Each of these terms is similar to those in equation (8.3-4) and are evaluated in the same manner.

The first two terms are already given by equations (8.3-5) and (8.3-6), which are

$$\frac{\partial \varepsilon_q^2}{\partial \Phi_{q,k}} = -2(T_q - \Phi_{q,k}) = -2\varepsilon_q \tag{8.3-16}$$

$$\frac{\partial \Phi_{q,k}}{\partial I_{q,k}} = \alpha \Phi_{q,k}(1 - \Phi_{q,k}) \tag{8.3-17}$$

Taking the partial derivative of equation (8.3-7)

$$I_{q,k} = \sum_{p=1}^{n} w_{pq,k} \Phi_{p,j} \tag{8.3-18}$$

with respect to $\Phi_{p,j}$ gives

$$\frac{\partial I_{q,k}}{\partial \Phi_{p,j}} = w_{pq,k} \tag{8.3-19}$$

The summation over p disappears because only one connection is involved. Changing subscripts on equations (8.3-6) to correspond to the middle layer gives

$$\frac{\partial \Phi_{p,j}}{\partial I_{p,j}} = \alpha \Phi_{p,j}[1 - \Phi_{p,j}] \tag{8.3-20}$$

Changing subscripts on equation (8.3-7) and substituting the ith-layer input x_h for the jth-layer input $\Phi_{p,j}$ gives

$$I_{p,j} = \sum_{h=1}^{m} w_{hp,j} x_h \tag{8.3-21}$$

Taking the partial derivative of equation (8.3-21) gives

$$\frac{\partial I_{p,j}}{\partial w_{hp,j}} = x_h \tag{8.3-22}$$

Again, the summation over h in equation (8.3-21) disappears because only one connection is involved. Substitution of equations (8.3-17) through (8.3-20) into equation (8.3-15), use of equation (8.3-14) and the definition of $\delta_{pq,k}$ equation (8.3-10) gives

$$\frac{\partial \varepsilon^2}{\partial w_{hp,j}} = \sum_{q=1}^{r} (-2)\alpha(T_q - \Phi_{q,k})\left[\Phi_{q,k}(1 - \Phi_{q,k})\right] w_{pq,k}\alpha\left[\Phi_{p,j}(1 - \Phi_{p,j})\right] x_h$$

$$= -\sum_{q=1}^{r} \delta_{pq,k} w_{pq,k} \frac{\partial \Phi_{p,j}}{\partial I_{p,j}} x_h \tag{8.3-23}$$

If we define $\delta_{hp,j}$ as

$$\delta_{hp,j} \equiv \delta_{pq,k} w_{pq,k} \frac{\partial \Phi_{p,j}}{\partial I_{p,j}} \tag{8.3-24}$$

then equation (8.3-23) becomes

$$\frac{\partial \varepsilon^2}{\partial w_{hp,j}} = -\sum_{q=1}^{r} \delta_{hp,j} x_h \tag{8.3-25}$$

Since the change in weights as given in equation (8.3-13) is proportional to the negative of the rate of change of the square error with respect to that weight, then, substitution of equation (8.3-23) and (8.3-24) into equation (8.3-13) gives

$$\Delta w_{hp,j} = -\eta_{h,p} \frac{\partial \varepsilon^2}{\partial w_{hp,j}} = \eta_{h,p} \sum_{q=1}^{r} \delta_{pq,k} w_{pq,k} \frac{\partial \Phi_{p,j}}{\partial I_{p,j}} x_h$$

$$= \eta_{h,p} x_h \sum_{q=1}^{r} \delta_{hp,j} \tag{8.3-26}$$

and hence

$$w_{hp,j}(N+1) = w_{hp,j}(N) + \eta_{h,p} x_h \sum_{q=1}^{r} \delta_{hp,j} \tag{8.3-27}$$

If there are more than one middle layer of neurons, this process moves through the network, layer by layer to the input, adjusting the weights as it

goes. When finished, a new training input is applied and the process starts the whole process again. It continues until an acceptable error is reached. At that point the network is trained.

Example 8.1 Updating Weights Through Backpropagation. A simple, fully connected feedforward neural network is shown in Figure 8.10, where bias inputs of +1 and adjustable weights w_{1C}, w_{1D} and w_{1E} have been added to neurons C, D, and E, respectively. (See Section 8.4 for a discussion of bias.) All neurons have the same logistic activation function with $\alpha = 1$ and the same learning constants with $\eta = 0.5$.

The desired output of neuron E is 0.1. The weights are randomized to the values shown, and training is started. Then the backpropagation process of learning is applied in the backward direction and the process is carried through one cycle, i.e., the change in each of the six weights and the new values of the weights are calculated.

For the weights between layers j and k, substitution of equation (8.3-9) into equation (8.3-11) gives the changes to be

$$\Delta w_{pq,k} = -\eta_{p,q}\left[-2\alpha\left[T_q - \Phi_{q,k}\right]\Phi_{q,k}\left[1 - \Phi_{q,k}\right]\Phi_{p,j}\right] \quad \text{(E8.1-1)}$$

For the weights between layers i and j, substitution of equation (8.3-23) into

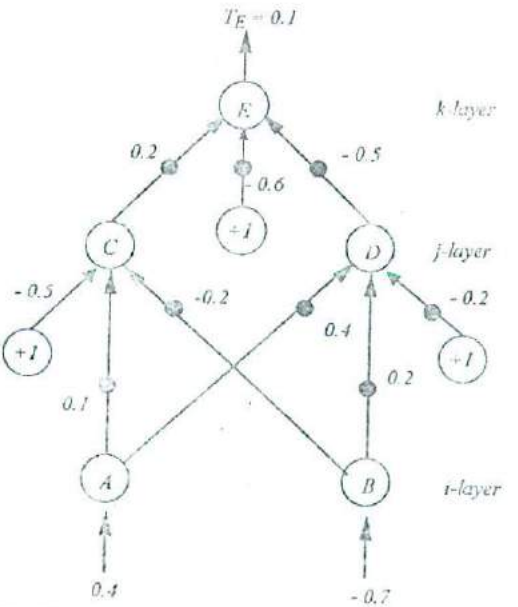


Figure 8.10 Weight adjustment (training) in a simple neural network.

equation (8.3-26) gives the changes to be

$$\Delta w_{hpj} = -\eta_{h,p}\left[\sum_{q=1}^{r} -2\alpha(T_q - \Phi_{q,k})\Phi_{q,k}(1-\Phi_{q,k})w_{pq,k}\alpha\Phi_{p,j}(1-\Phi_{p,j})x_h\right] \tag{E8.1-2}$$

The subscripts h, p, q, and indices m, n, and r are defined in Figure 8.7. Subscript 1 refers to the +1 bias terms..

First, calculate the outputs of each neuron using the logistic function with $\alpha = 1$.

$$I_C = 0.4 \times 0.1 + (-0.7) \times (-0.2) + 1 \times (-0.5) = 0.04 + 0.14 - 0.50$$
$$= -0.32 \qquad \Phi(I_C) = 0.42$$
$$I_D = 0.4 \times 0.4 + (-0.7) \times 0.2 + 1 \times (-0.2) = 0.16 - 0.14 - 0.2$$
$$= -0.18 \qquad \Phi(I_D) = 0.46$$
$$I_E = 0.42 \times 0.2 + 0.46 \times (-0.5) + 1 \times (-0.6) = 0.08 - 0.23 - 0.60$$
$$= -0.75 \qquad \Phi(I_E) = 0.32$$

Substitution of these values and other network parameters into equations (E8.1-1) and (E8.2-2) gives

$$\Delta w_{CE} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 0.42$$
$$= -0.020$$
$$\Delta w_{DE} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 0.46$$
$$= -0.022$$
$$\Delta w_{1E} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 1$$
$$= -0.048$$
$$\Delta w_{AC} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 0.20$$
$$\times 1 \times 0.42 \times (1.00 - 0.42) \times 0.4$$
$$= -0.00093 = -0.001$$
$$\Delta w_{AD} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32)$$
$$\times (-0.50) \times 1 \times 0.46 \times (1.00 - 0.46) \times 0.4$$
$$= 0.00238 = 0.002$$
$$\Delta w_{BC} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 0.20$$
$$\times 1 \times 0.42 \times (1.00 - 0.42) \times (-0.7)$$
$$= 0.00163 = 0.002$$

$$\Delta w_{BD} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times (-0.50) \times 1 \times 0.46 \times (1.00 - 0.46) \times -0.7 = -0.00142 = -0.001$$

$$\Delta w_{1C} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times 0.20 \times 1 \times 0.42(1.00 - 0.42) \times 1 = -0.0023 = -0.002$$

$$\Delta w_{1D} = -0.5 \times (-2) \times 1 \times (0.10 - 0.32) \times 0.32 \times (1.00 - 0.32) \times -(0.50) \times 1 \times 0.46(1.00 - 0.46) \times 1 = 0.0059 = 0.006$$

Adding these changes to the original weights gives the new weights.

$$w_{CE} = 0.200 - 0.020 = 0.180$$
$$w_{DE} = -0.500 - 0.022 = -0.522$$
$$w_{IE} = -0.600 - 0.048 = -0.648$$
$$w_{AC} = 0.100 - 0.001 = 0.099$$
$$w_{AD} = 0.400 + 0.002 = 0.402$$
$$w_{BC} = -0.200 + 0.002 = -0.198$$
$$w_{BD} = 0.200 - 0.001 = 0.199$$
$$w_{1C} = -0.500 - 0.002 = -0.502$$
$$w_{1D} = -0.200 + 0.006 = -0.194$$

This process is repeated until all sample pairs in the epoch have been utilized. After these weight changes have been calculated, the total square error is then calculated. If it is more than the specified amount, the learning algorithm is again applied to the network using another epoch of training data. A better alternative is to continue the training process until monitoring of the total square error for a test set of data starts to increase, even though the total square error for the training set continues to decrease. □

8.4 FACTORS THAT INFLUENCE BACKPROPAGATION TRAINING

Adding a bias (a +1 input with a training weight, which can be either positive or negative) to each neuron is usually desirable to offset the origin of the activation function. This produces an effect equivalent to adjusting the threshold of the neuron and often permits more rapid training. The weight

of the bias is trainable just like any other weight except that the input is always +1.

Momentum

Another technique to reduce training time is the use of *momentum*, because it enhances the stability of the training process. Momentum is used to keep the training process going in the same general direction analogous to the way that momentum of a moving object behaves. This involves adding a term to the weight adjustment that is proportional to the amount of the previous weight change. In effect, the previous adjustment is "remembered" and used to modify the next change in weights. Hence, equation (8.3-11) now becomes

$$\Delta w_{pq,k}(N+1) = -\eta_{pq}\,\delta_{pq,k}\Phi_{p,j} + \mu\,\Delta w_{pq,k}(N) \qquad (8.4\text{-}1)$$

where μ is the *momentum coefficient* (typically about 0.9). This relationship is shown in Figure 8.11. The new value of the weight then becomes equal to the previous value of the weight plus the weight change of equation (8.3-11), which includes the momentum term. Equation (8.2-12) now becomes

$$w_{pq,k}(N+1) = w_{pq,k}(N) + \Delta w_{pq,k}(N+1) \qquad (8.4\text{-}2)$$

This process works well in many problems, but not so well in others. Another way of viewing the purpose of momentum is to overcome the effects of local

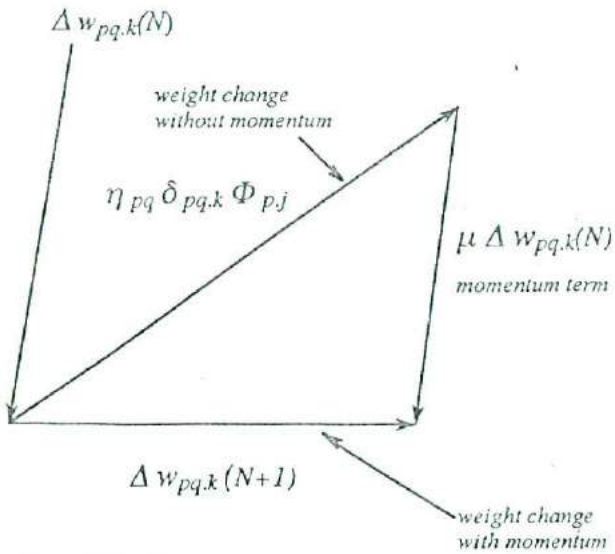


Figure 8.11 Influence of momentum upon weight change.

minima. The use of the momentum term will often carry a weight change process through one or more local minima and get it into a global minima. This is perhaps its most important function.

There is a substantial number of advanced algorithms or other procedures that have been proposed as means of speeding up the training of backpropagation networks. Sejnowski and Rosenberg (1987) proposed a similar momentum method that used exponential smoothing. However, the results were mixed. In some cases it improved the speed of the training, whereas in other cases it did not. Parker (1987) proposed a method called the "second-order" backpropagation that used the second derivative to produce a more accurate estimation of the correct weight change. The computational requirements were greater and were generally viewed as not being cost effective compared to other methods. It was, however, clear that higher-order (greater than 2) backpropagation systems were not effective. Stornetta and Huberman (1987) pointed out that the 0–1 range of sigmoidal function is not optimal for binary inputs. Since the magnitude of a weight adjustment is proportional to the output level of the neuron from which it originates, a level of 0 results in no modification. With binary inputs, half of the outputs (on the average) will be zero, and weights do not train. The proposed solution was to change the input range of the activation function from $-1/2$ to $+1/2$ by adding a bias of $-1/2$. They demonstrated that for binary functions this procedure reduces the training time by 30–50%. Today a more common method of accomplishing this is to use the arctan or hyperbolic tangent activation function.

Despite some spectacular results, it is clear that backpropagation is not a panacea. The main problem is the long and sometimes uncertain training time. Some artificial neural networks have been known to require days or weeks of training, and in some cases the network simply will not train at all. This may be the result of a poor choice of training coefficients or perhaps the initial random distribution of the weights. However, in most cases failure to train is usually due to local minima or *network paralysis*, where training virtually ceases due to operation in the flat region of the sigmoid function.

Stability

The proof of convergence of backpropagation by Rumelhart, Hinton and Williams (1986) used infinitesimal weight adjustments. This is impractical because it requires infinite training time. In the real world, if the step size is too small, the training is too slow; if the step size is too large, instability may result. However, recent efforts that involve the use of large steps initially with automatic reduction as the training proceeds have been quite successful in reducing training time.

Another issue is temporal instability. If a network is to learn the alphabet, it is of no value to learn the letter B if it destroys the learning of letter A. The network must learn the entire training set without disrupting what is

already learned. Rumelhart's convergence requires the network to process all training examples before adjusting any weights. Furthermore, backpropagation may not be useful if the network faces a continuously changing environment where the inputs are continuously changing, because the process may never converge. There are alternate networks discussed later that are useful in such situations.

Adjusting α Coefficient in Sigmoidal Term

Sometimes weights become very large in value and force the neurons to operate in a region where the sigmoidal function is very flat—that is, its derivative is very small. Since the error sent back for training in backpropagation is proportional to the derivative of the sigmoid function, very little training takes place. This network paralysis can sometimes by avoided by reducing the training coefficient, which unfortunately results in extending the training time.

A better method of coping with network paralysis if to adjust the α coefficient on the exponential term in the logistic term. By decreasing α, we effectively spread out the sigmoidal function over a wider range. Values of I that gave $\Phi(I)$ of 0.99 now gives smaller values, like 0.75 or perhaps 0.35, depending upon the value of α. The training process is now operating in a range where the derivative of the sigmoidal is much greater, and hence training will proceed much faster.

For large negative values of I, the logistic activation function squeezes the $\Phi(I)$ values close to zero. Use of hyperbolic tangent and arctangent activation functions spreads these values down into the range between 0 and -1, thereby eliminating the network paralysis. The use of a trainable bias term as an input to each neuron, which is standard practice in most commercial neural network software, is also useful in avoiding network paralysis.

Dealing with Local Minima

Perhaps the major problem of backpropagation is local minima. Since backpropagation employs a form of gradient descent, it is assumed that the error surface slope is always negative and hence constantly adjusting weights toward the minimum. However, error surfaces often involve complex, high-dimensional space that is highly convoluted with hills, valleys, folds, and gullies. It is very easy for the training process to get trapped in a local minimum. One of the most practical solutions involves the introduction of a shock—that is, changing all weights by specific or random amounts. If this fails, then the most practical solution is to re-randomize the weights and start the training over.

Another alternative is to utilize *simulated annealing*, a technique that is used to search for global minima in a search surface in which states are updated based on a statistical rule rather than deterministically. This update

rule changes to become more deterministic as the search progresses. (*Simulated annealing* is discussed later in Chapter 9.) The procedure is to use backpropagation until the process seems to stall. Then *simulated annealing* is used to continue training until the local minimum has been left behind. Then the simulated annealing is stopped and the backpropagation training continues until a global minimum is reached. In most cases, only a few simulated annealing cycles of this two-stage process are needed. If the mean square error of the outputs stalls in its descent, then the annealing process may have to be used again. The final training step in this process is backpropagation to minimize the overall error of the process.

Learning Constants

Choosing the correct learning constant η is important in backpropagation training. First, η cannot be negative because this would cause the change of the weight vector to move away from the ideal weight vector position. Of course, if η is equal to 0, then no learning takes place. Therefore, η must always be positive. It can be shown both analytically and experimentally that if η is greater than 2, then the network is unstable, and if η is greater than 1, then the weight vector will overshoot the ideal position and oscillate, rather than settle into a solution. Hence, η should be in the range between 0 and 1.

If η is large (0.8 or thereabouts), then the weight vector will take relatively large steps and will find the minimum faster. However, if the input data patterns are not highly compacted around the "ideal" example, this will cause the network to jump wildly each time a new input pattern is presented. If the value of η is small (0.2 or thereabouts), then the weight vector will take small steps toward the "ideal" position, and will not vary wildly if the input data patterns are not very close to an "ideal" example. However, the network will require a longer time to learn the patterns with many iterations of the data needed.

As a compromise, large values of η are used when the input data patterns are close to the ideal whereas small values of η are used when they are not. When the nature of the input data patterns are not known, then it is better to use a moderate value of η. As suggested earlier, an even better method is to change the value of η as the network learns, beginning with a large value initially and reducing it as the learning progresses. Then the leaning process is not distracted by minor variations in the input data. It is important to remember that real data patterns are never perfect examples of a category, and that the separating hyperplanes cannot always separate all the input data into a specific number of categories.

Variations of the Standard Backpropagation Algorithm

In addition to such standard techniques are adding momentum, adjusting learning rate, adjusting the exponential decay constant in the sigmoidal

function, and using other activation functions, there are a number of other variations that are often useful in many situations. Most of them modify the standard backpropagation algorithm in one way or another.

Cumulative Update of Weights

A variation of backpropagation training (called "cumulative backpropagation") that seems to be helpful in speeding up training is the cumulative update of weights. In this case, the individual weight changes for each weight are accumulated for an epoch of training, summed, and then the cumulative weights changes are made in the individual weights. This procedure significantly reduces the amount of computation involved, and there usually is no noticeable effect on the final training of the network.

Fast Backpropagation

This variation of backpropagation introduced by Tariq Samad of Honeywell (Samad, 1988) involves the following changes in standard backpropagation. A multiple of the error at layer k is added to the k-layer activation value $\Phi_{q,k}$ prior to doing the weight update for weights on connections between the j and k layers. This can dramatically increase the speed of training, usually by more than an order of magnitude. Furthermore, it has been shown in one case to reach convergence when standard backpropagation training failed to do so after 10,000 iterations.

Quickprop Training

Fahlman's quickprop training algorithm is one of the more effective algorithms in overcoming the step-size problem in backpropagation. The $\partial \varepsilon^2 / \partial w$ values are computed as in standard backpropagation (Fahlman, 1988). However, a second-order method related to Newton's method is used to upgrade the weights in place of simple gradient descent. Fahlman reports that quickprop consistently outperforms backpropagation, sometimes by a wide margin.

Quickprop's weight update procedure depends on two approximations: (1) Small changes in one weight produce relatively little effect on the error gradient observed at other weights, and (2) the error function with respect to each weight is locally quadratic. The slopes and weights for current and previous iterations are used to define a parabola. The algorithm then goes to the minimum point of this parabola as the next weight. This process continues going through all weights for an epoch. If the error is sufficiently small, the training process is terminated; if not, training continues for another epoch.

Use of Different Error Functions

The error function defined in equation (8.3-2) is proportional to the square of the Euclidean distance between the desired output and the actual output of the network for a particular input pattern. As an alternative, we can substitute any other error function whose derivatives exist and can be calculated as the output layer. Errors of third and fourth order have been used to replace the traditional square error criterion. NeuralWorks[2] has suggested cubic and quadratic errors of the forms

$$\varepsilon^2 = \sum_{q=1}^{r} \left[|T_q - \Phi_{q,k}| \right]^3 \tag{8.4-3}$$

$$\varepsilon^2 = \sum_{q=1}^{r} \left[T_q - \Phi_{q,k} \right]^4 \tag{8.4-4}$$

which have local errors, analogous to equation (8.3-10), of

$$\delta_{pq,k} = -3(T_q - \Phi_{q,k})^2 \frac{\partial \Phi_{q,k}}{\partial I_{q,k}} \tag{8.4-5}$$

$$\delta_{pq,k} = -4(T_q - \Phi_{q,k})^3 \frac{\partial \Phi_{q,k}}{\partial I_{q,k}} \tag{8.4-6}$$

It is not clear whether the benefits of cubic and quadratic error functions compensate for the additional complexity introduced.

Delta-Bar-Delta Networks

Most changes of the standard backpropagation algorithm involve one of two methods: (1) Incorporate more analytical information to guide the search such as second-order backpropagation, which has not proven particularly successful, and (2) use heuristics (often intuition) that are reasonably accurate. The delta-bar-delta algorithm is an attempt introduced by R. A. Jacobs (Jacobs, 1988) to improve the speed of convergence using heuristics. Empirical evidence suggests that each dimension of the weight space may be quite different in terms of the overall error surface. By using past values of the gradient, heuristics can be used to imply the curvature of the local error surface, from which intelligent steps can be taken in weight optimization. Since parameters for one weight dimension may not be appropriate for all weight dimensions, each neuron has its own learning rate which is adjusted over time (i.e., reduced as the training progresses). This method has proven to be effective in reducing the time required for training neural networks.

[2]Copyright held by NeuralWare Inc., Pittsburgh, PA.

Extended Delta-Bar-Delta

Minai and Williams (1990) have extended the delta-bar-delta algorithm by (1) adding a time-varying momentum term and (2) adding a time-varying learning rate. The rates of change in momentum and learning rate decrease exponentially with weighted gradient components so that greater increases will be applied in areas of small slope or curvature than in the areas of high curvature. To prevent wild jumps and oscillations in weight space, ceilings are placed on the individual learning and momentum rates.

8.5 SENSITIVITY ANALYSIS IN A BACKPROPAGATION NEURAL NETWORK

Sensitivity analysis is an extremely useful tool in many practical applications. The introduction of a small perturbation in one of the input neurons usually produces perturbations in the outputs of all neurons connected, directly or indirectly, to that input neuron (Guo and Uhrig, 1992; Hashem, 1992). The ratio of the magnitude of the perturbation in the output of a specific output neuron to the perturbation in the input of a specific input node is defined as the *sensitivity*. A perturbation of x_h of the multilayer neuron network in Figure 8.7 produces perturbations in all values of y_q. Hence, the sensitivity $\sigma_{q,h}$ is given by

$$\sigma_{q,h} = \frac{\Delta y_q}{\Delta x_h} = \frac{\partial y_q}{\partial x_h} \tag{8.5-1}$$

where the ratio of the Δ perturbations are replaced with partial derivatives (after taking the appropriate limit). Using the notation of Figure 8.7, equation (8.5-1) becomes

$$\sigma_{q,h} = \frac{\partial y_q}{\partial x_h} = \frac{\partial \Phi_{q,k}}{\partial x_h} = \frac{\partial \Phi_{q,k}}{\partial I_{q,k}} \frac{\partial I_{q,k}}{\partial \Phi_{p,j}} \frac{\partial \Phi_{p,j}}{\partial I_{p,j}} \frac{\partial I_{p,j}}{\partial x_h} \tag{8.5-2}$$

The first term is given by equation (8.3-6) to be

$$\frac{\partial \Phi_{q,k}}{\partial I_{q,k}} = \alpha \Phi_{q,k}\left[1 - \Phi_{q,k}\right] \tag{8.5-3}$$

The second term is found by taking the partial derivative of equation (8.3-7) with respect to $\Phi_{p,j}$, which gives

$$\frac{\partial I_{q,k}}{\partial \Phi_{p,j}} = \sum_{p=1}^{n} w_{pq,k} \tag{8.5-4}$$

The third term is given by equation (8.3-20) to be

$$\frac{\partial \Phi_{p,j}}{\partial I_{p,j}} = \alpha \Phi_{p,j}\left[1 - \Phi_{p,j}\right] \tag{8.5-5}$$

The fourth term is found by taking the partial derivative with respect to x_h of equation (8.3-21) to be

$$\frac{\partial I_{p,j}}{\partial x_h} = w_{hp,j} \tag{8.5-6}$$

Again, the summation disappears because only one input is involved. Substituting these four terms into equation (8.5-2) gives

$$\sigma_{q,h} = \alpha \Phi_{q,k}\left[1 - \Phi_{q,k}\right] \sum_{p=1}^{n} w_{pq,k}\, \alpha \Phi_{p,j}\left[1 - \Phi_{p,j}\right] w_{hp,j} \tag{8.5-7}$$

which becomes

$$\sigma_{q,h} = \alpha^2 \Phi_{q,k}\left[1 - \Phi_{q,k}\right] \sum_{p=1}^{n} w_{pq,k} \Phi_{p,j}\left[1 - \Phi_{p,j}\right] w_{hp,j} \tag{8.5-8}$$

Experimental Evaluation of Sensitivity Coefficients

An experimental evaluation of sensitivity coefficients, sometimes called the "dither" method, is possible after a neural network has been trained. It involves the introduction of small perturbations of each input x_i, one at a time, of about 0.5% in both the positive and negative directions. The resultant perturbations of each output y_j in each direction for each input perturbation is measured and averaged. The sensitivity coefficient is then taken as

$$\sigma_{q,h} = \frac{\overline{\Delta y_q}}{\overline{\Delta x_h}} \tag{8.5-9}$$

where the averages are taken over the perturbations in the positive and negative directions. If there are h inputs and q outputs, then there are hq sensitivity coefficients that can be evaluated experimentally.

In general, the usefulness of such measurements are limited. The principal problem is that the values are valid only for the particular location in problem space represented by the values of the inputs and outputs before they are perturbed.

8.6 AUTOASSOCIATIVE NEURAL NETWORKS

Although any layer in an neural network can have any number of neurons, the most common backpropagation network starts with a large number of neurons in the input layer and has relatively few neurons in the output layer. The reason is that many problems involve a complex description of a situation or condition as the input, and a limited number of classes or conditions as the output. There is no rule prescribing the number in neurons of a middle or hidden layer of a three-layer neural network. Kolmogorov's theorem gives us a number of neurons that guarantee the existence of a mapping function between the input and output, but as indicated earlier, this may not be the optimal or most appropriate number of neurons in any given situation. As a rule of thumb, the number of neurons in the middle layer should be less than the number of data sets in an epoch so that the neural network does not *memorize* the various input data sets; that is, a particular neuron in the hidden layer becomes associated with a particular data set of an epoch.

Autoassociative neural networks, in which the output is trained to be identical to the input, have many unusual characteristics that can be exploited in many applications. Autoassociative neural networks as defined here are feedforward, fully connected, multilayer perceptrons usually (but not always) trained using backpropagation (Masters, 1993). The number of neurons in the hidden layer(s) may be greater or smaller than the number in the input and output. All neurons in the middle layer(s) must have nonlinear sigmoidal (logistic, arctangent, or hyperbolic tangent) activation function, but the output layer may have either a linear or nonlinear activation function. Kramer (1991, 1992) has investigated the special case where the middle layer has fewer neurons (which Kramer called a "bottleneck") and reported features that lend themselves to diagnostic and monitoring as well as to the identification of nonlinear principal components. This bottleneck layer prevents a simple one-to-one of "straight through" mapping from developing during the training of the network.

Let us consider the autoassociative backpropagation network shown in Figure 8.12, where there are 100 neurons in the input and output layers and 40 neurons in the hidden (bottleneck) layer. We start with the desired output of the output layer being exactly equal to input to the input layer, and we proceed with the training using backpropagation. The data for the training set must be chosen so that individual input–output values cover the range over which the network will have to accept inputs in the future. Eventually, within some tolerable error, the input and output of the network are the same. This indicates that the information contained in the input vector which has 100 components is approximately equal to the information contained in the output vector which also has 100 components. Furthermore, the information in the input vector passes through the middle layer, where it is represented by a 40-component vector. This compression of the information into

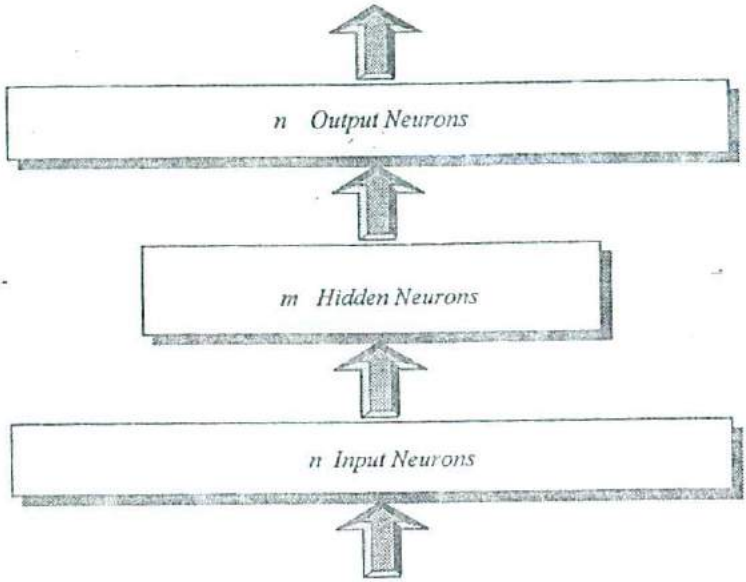


Figure 8.12 Autoassociative neural network.

40 components is a very useful property for certain specific applications. In effect, the 100-dimensional data set has been reduced to a 40-dimensional data set and then reexpanded to a 100-dimensional data set with the error minimized in a least square sense. This means that the 100-dimensional input must be reproduced at the output with only 40 independent variables represented by the outputs of the neurons in the middle layer. In effect, least squares training induces the network to model correlations and redundancies in the input data in order to reproduce the input data at the output with minimal distortion under the dimensional restriction of the "bottleneck" layer.

A quantity that is often of interest is the compressed representation of the input variable consisting of 40 values in the hidden layer. The values from this middle layer are often extracted and utilized as a compressed representation of the input information. Since we are now effectively dealing with a two-layer network (the input layer and the middle layer, which is now an output layer), the validity of this representation is dependent upon the nature of the input data. There is no "hidden" layer to capture the patterns of features of the input data and present it in an organized manner to the middle layer. The same situation exists with respect to the conversion of a

40-dimension representation of the data contained in the middle layer to a 100-dimension representation of the data contained in the output layer. These situations can be overcome by putting in two "intermediate" layers, one between the input and the bottleneck layer and another between the middle layer and the output layer, to give a five-layer neural network. Such an arrangement is shown in Figure 8.13. The second and fourth layers, each having 135 neurons, are effectively feature extraction layers. Kramer calls them "mapping" and "demapping" layers, respectively. These layers have more nodes than the input and output layers so that they are capable of representing nonlinear functions of arbitrary form. However, care must be taken to avoid memorization by the neurons in these layers. With this configuration, the signal now coming from the third (middle) layer of this five-layer neural network now is an appropriate 40-dimensional representation of the input data. In the case of this five-layer autoassociative neural network, the second and fourth layers must have nonlinear activation functions, but the middle and output layers may have either linear or nonlinear activation functions. Kramer indicates that to capture linear functionality efficiently, linear bypasses can be allowed from the input layer to the bottleneck layer and from the bottleneck layer to the output layer, but not across the bottleneck layer.

The functioning of autoassociative neural networks should not be confused with associative memory of the type illustrated in the last chapter, where a neural network that was trained to recognize the letters of the alphabet could recognize distorted versions of the letters. In effect, the distorted letter was closer (in a least squares sense) to the corresponding undistorted letter stored in the associative memory. In contrast, the autoassociative neural network has no stored quantities and no discrete classes, and its outputs are continuous variables.

Once the autoassociative neural network of Figure 8.13 has been trained, it can be split into two separate neural networks. The first consists of the input, mapping, and bottleneck layers as shown in Figure 8.14*a*, in which the input is separated into a reduced-dimensional representation of the input. If all neurons in this autoassociative neural network had linear activation functions, the output of the middle layer would be the linear principal components. With nonlinear activation functions, these components represent nonlinear principle components of the input, and the number of principal components obtained is equal to the number of neurons in the bottleneck layer.

The second network consists of the bottleneck, demapping, and output layers as shown in Figure 8.14*b*, in which the reduced-dimensional representation of the middle layer is expanded into a good representation of the original input signal. This separation can occur only after training has taken place; otherwise there are no target values available for the neurons in the middle layer.

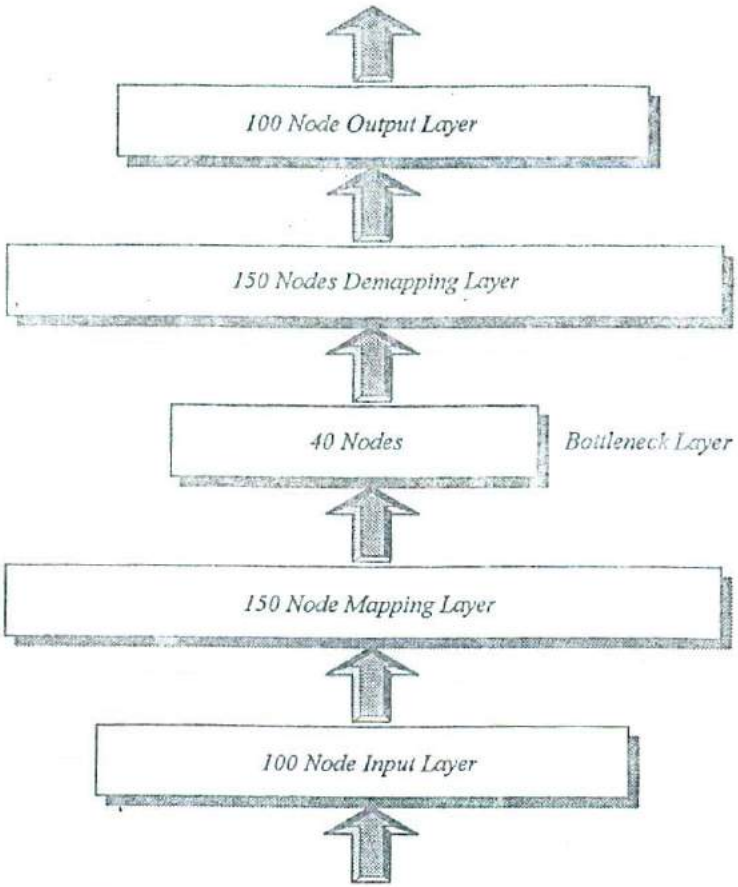


Figure 8.13 Five-layer autoassociative neural network.

Use of an Autoassociate Neural Network for Filtering

The process described above is a form of "filtering" where the amount of information lost is related inversely to the number of neurons in the "bottleneck" layer. If the input is a time series taken from an fluctuating analog signal with a sampling rate of 100 samples per second, then the 100 input components represents 1 sec of data. (Similarity, if the sampling rate is 5 or 1000 samples per second, the input data represents 20 or 0.1 sec of data, respectively.) The training proceeds by using data sets consisting of successive groups of 100 samples, shifted by one sample, which are entered into the neural network software or hardware as both the input and desired output. Each successive group of 100 values consists of the next sample value and the

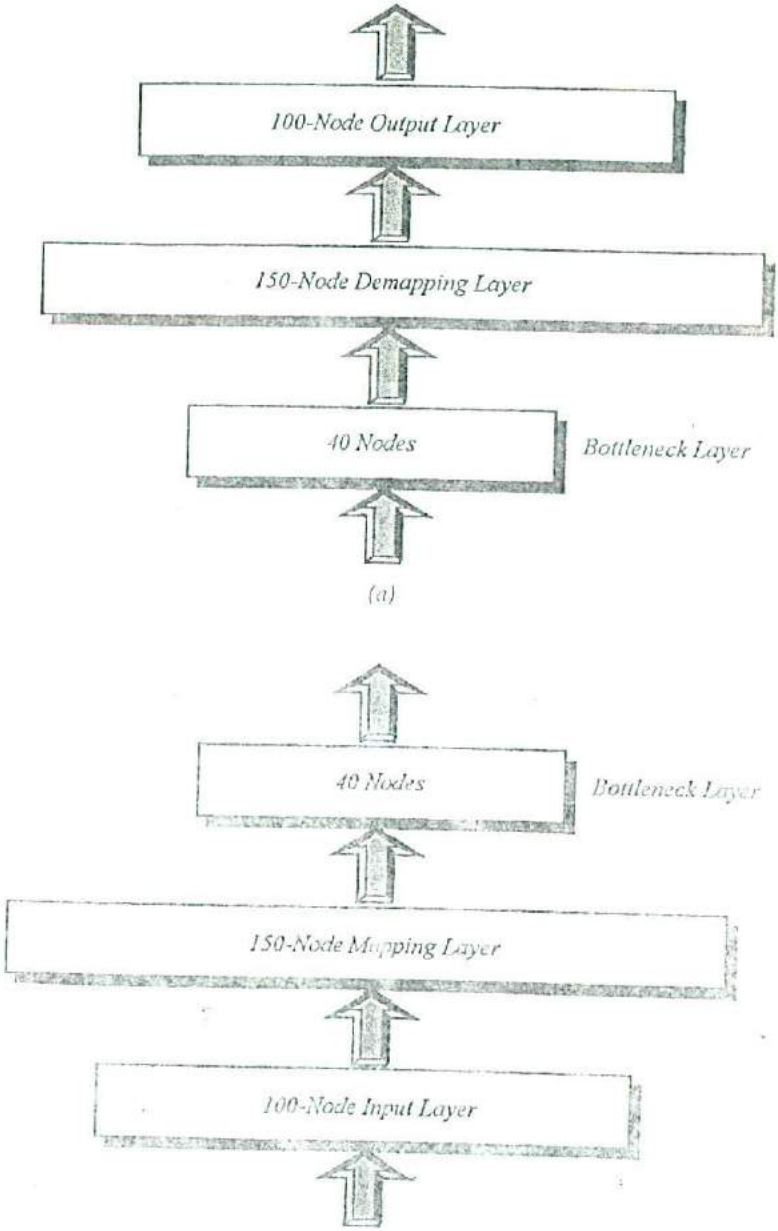


Figure 8.14 Five-layer autoassociative neural network broken into three-layer networks.

preceding 99 sample values. The oldest sample is dropped. The neural network then undergoes another training cycle, and then the data are shifted again to include the next data point, and the training process proceeds. Care must be taken to ensure that the range of variables to which the 100 inputs and outputs are subjected during training covers the range of input data expected in the future. When this training is complete, a time series can be introduced into the neural network by introducing successive 100-sample groups of data to the input layer. Because the bottleneck forces the variable to be represented by fewer dimensions, some information is lost. In this case, it is the high-frequency information that is lost since more dimensions are required to represent a high-frequency variable. In effect, the autoassociative neural network behaves as a low-pass filter.

It is in this filtering application that the true nature of the autoassociative network process is revealed. In effect, the high-frequency components in the signal are eliminated by the middle layer. The smaller the number of neurons in the middle layer, the lower the cutoff frequency and the greater the number of high-frequency components that are eliminated. Hence, it is very clear that the input and output signals can never be identical, because information has been filtered out of the input signal. The output can be only a low-frequency approximation of the original input signal. As the number of neurons in the hidden layer decreases, the cutoff frequency is reduced further, thereby eliminating more of the higher-frequency components.

Use of Autoassociative Neural Networks in Systemwide Monitoring

Neural networks, in combination with other artificial intelligence technologies, offer means of interpreting data and measurements in ways that are not otherwise possible. The unique characteristics of three- and five-layer autoassociative neural networks in which the outputs are trained to emulate the inputs over an appropriate dynamic range have been explored and found to be useful in systemwide monitoring. Many (typically 10–20) variables of complex systems (power plants, chemical or manufacturing processes, social systems, etc.) that have some degree of correlation (typically > 0.3) with each other constitute the inputs. Hence, each output receives some information from almost every input. During training to make each output equal to the corresponding input, the interrelationships between all the input variables and each individual output are embedded in the connection weights of the network. As a result, any specific output, even the corresponding output shows only a small fraction of the input change over a reasonably large range. This characteristic allows the autoassociative neural network to detect drift, deterioration, or failure of a sensor by simply comparing each input with the corresponding output.

Upadhyaya and Eryurek (1992) have demonstrated the feasibility of such an application using data from the EBR-2 (Experimental Breeder Reactor

#2). A necessary prerequisite for such an application is that the various inputs have some degree of correlation. As indicated above, when all of the inputs to an autoassociative neural network are correlated to some degree, then each output is dependent on all the inputs. Hence, the deterioration of one input signal will have only a slight influence on the outputs. The change in the channel that corresponds to that input would be larger than the changes in the other channels, but significantly smaller than the change in the input because of the influence of the correlation with the inputs from the other channels.

In principle, all that is necessary is to detect a deteriorating sensor or instrumentation channel is to compare each input with the corresponding output, calculate the difference, compare it to an allowable difference, and trip an alarm when the difference exceeds the allowable difference. To avoid false alarms, several small deviations beyond the limit may be required in a specified time to trip an alarm. A single large deviation, of course, should trip the alarm. In most practical applications, especially when noise is present, a more sophisticated technique to detect error, such as "sequence probability ratio test," (Wald, 1945) should be employed to minimize the number of missed and false-positive alarms.

An alternative interpretation of the existence of differences between the inputs and the corresponding outputs of the autoassociative neural network might be that the input–output relationship of the system from which the signals come may have changed due to system failure or changes of some sort in the system. All of the results reported in the experimental work discussed in this section are based on the assumption that the underlying system does not change and that only the sensors and related instrumentation channels are being validated. However, in the real world, systems change with time—sometimes slowly, sometimes rapidly—while still behaving normally. Sometimes the changes are anticipated; sometimes these changes come as a surprise. If the changes occur during the training period (or can be artificially introduced), then the relationship between the variables for different conditions can be trained into an autoassociative neural network. This is the case with power ascension from 45% to 100% of full power in the Experimental Breeder Reactor-2 as shown in Example 8.2 (Upadhyaya and Eryurek, 1992).

Corrected Readings from Deteriorating or Failed Sensors

One of the unique advantages of using autoassociative neural networks is the ability to obtain the correct reading for a sensor that has failed. Since the specific sensor that has failed can be identified as discussed above, all that is needed is to carry out an adjustment of the input to the input neuron representing that sensor input to bring the outputs of the autoassociative neural network back to their original values (in a minimization of least squares difference sense). Multiple failures can also be handled in the same

way with a multidimensional search for the proper input values, provided that the number of correct sensors is greater than the number of neurons in the bottleneck layer.

Robust Autoassociative Neural Networks

To improve the behavior of the autoassociative neural network, a technique involving the addition of uniform random noise up to 10% to each input, one at a time, while retaining the noise-free values for the desired output, can be employed. This technique is analogous to adding noise to a neural network input to avoid "memorization" and to speed training. Application of this process to all input–output pairs of neurons during training can produce a very robust autoassociative neural network in which the outputs are virtually immune to input change up to 10% of the range of the input (Wrest, 1996).

A critically important issue is how to deal with changing plant configurations and conditions that are not trained into the autoassociative neural network. Fortunately, such changes would be readily detected in most cases by the comparison of outputs with inputs. Differences in more than one input–output pair are almost invariably associated with changes in the system rather than sensor failure, because simultaneous failures of more than one sensor are very rare. Clearly a change in configuration not trained into the autoassociative neural network requires immediate additional training or retraining. Another important issue that needs to be investigated is how the retrained networks relate to the previously trained network. It may be advantageous to retain all consecutive network configuration to have an "audit trail" for the calibration and drift detection. For a slowly changing condition, especially one that is cyclic in nature, is better to train over a whole cycle when possible so that the influence of this quantity is included in the trained network.

Example 8.2 Behavior of an Autoassociative Neural Network as a Plantwide Monitoring System. This autoassociative technique of plant-wide monitoring was applied to data from 18 signals (8 from the primary system and 10 from the secondary system) from the EBR-2 during ascension in power, and the results of these measurements for the primary system are shown in Figure 8.15. Data were collected for the eight primary variables (defined in Table 8.1) as EBR-2 increased in power output from 45% (run #1) to 100% (run #150) of full power. All variables were normalized and scaled into the interval 0.1–0.9. The autoassociative neural network was trained using data collected during the power ascension. Some variables changed rather significantly during the power ascension (between run #1 and run #150) while others changed very little. (The lines connecting the points are used only to indicate that these points belong to the same run.) In all cases, the values predicted by the trained network were within about 0.5% of the measured

value. However, when an error was deliberately introduced into variable #1 (as indicated with the point connected with "dashed" lines in Figure 8.15), the corresponding value predicted by the trained network did not change significantly. It is this characteristic behavior of autoassociative neural networks that allows monitoring of many variables to be carried out simultaneously by simply comparing network inputs and outputs. □

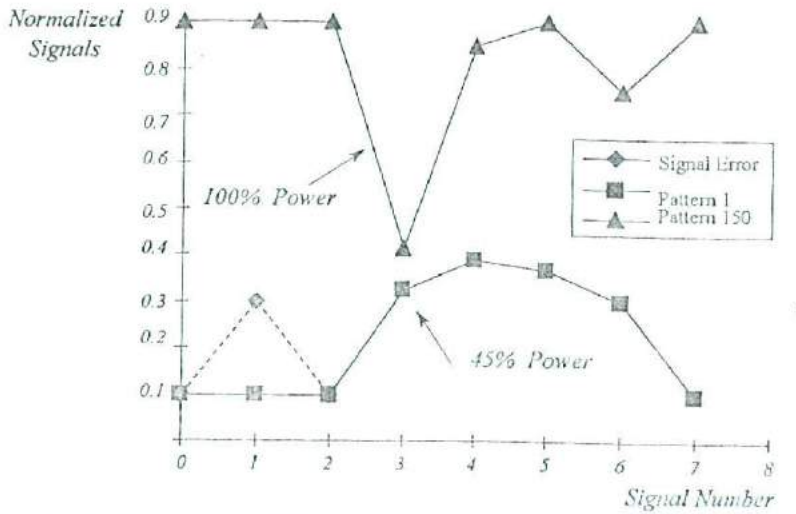


Figure 8.15 Comparison of actual output with output predicted by autoassociative neural network at 45% power and 100% power. (Dashed lines indicate effect of an error deliberately introduced into input signal #1.)

Table 8.1 EBR-2 variables of the primary coolant system monitored by an autoassociative neural network

Variable Number-- Variable
0 -- Power Level (%)
1 -- Core Exit Temperature (°F)
2 -- Control Rod Position (°F)
3-- Primary Pump Flow rate (%)
4-- High Pressure Plenum Sodium Temp. (°F)
5-- Low Pressure Plenum Sodium Temp. (°F)
6-- IHX Primary Outlet Sodium Temp. (°F)
7 --Core Upper Sodium Temp. (°F)

Inferential Measurements Using Neural Networks

Inferential measurement, as the name implies, is the inferring of a measurement value from its relationship, usually its physical correlation, with other variables (Guha, 1992). Reasons for an inferential measurement are that the quantity cannot be measured directly, measurements are difficult or expensive, a sensor is failing or has failed, the measurement process itself is deteriorating, or comparison of an inferred value with an actual value will assist in the identification and diagnoses of problems. The mapping ability of neural networks are ideal for such inferential measurements because they can map plant characteristics to the quantity whose measurement is to be inferred. Typically, the neural network is a simple multilayer perceptron with only a few (three to five) inputs and a single output. The number of neurons in the middle layer is usually not important as long as memorization and overtraining are avoided. The inputs must have some degrees of correlation with the quantity to be inferred, because using an input with no relation to the output would only deteriorate the quality of the measurement.

Examples of where inferential measurements have been used advantageously include the following examples:

1. Inferential measurements of nitrous oxide emissions from a gas-line pumping station have been used to demonstrate compliance with regulatory requirements. This avoided the placement of a chemical analysis unit at each of many pumping stations that often are located at remote sites as well as the need for technical personnel required to carry out the measurements.
2. Inferential measurements of feedwater flow in a nuclear power plant (an important quantity in the thermal power calibration) have been carried out using a neural network to map four related inputs to the flow. The neural network is trained using data gathered immediately after the venturi flowmeter has been cleaned and calibrated. As the venturi fouls due to water chemistry phenomena, a 1–2% difference develops between the inferred (correct) value and the (incorrect) measurement by the fouled flowmeter.
3. Inferential measurements are also being used for sensor validation. Again the process is one of mapping several related inputs to a single measured quantity. If the actual measurement deviates significantly from the value predicted by the trained neural network, then sensor failure or deterioration may be involved.

8.7 AN ALTERNATE APPROACH TO NEURAL NETWORK TRAINING

In Section 8.3, the traditional approach to backpropagation training of a neural network was examined. The neural network was a traditional three-

layer network with the input layer as a buffer layer and the hidden and output layers having nonlinear activation functions (a logistic function in this case). Such a network has been shown by Kolmogorov (1965) to be capable of mapping any arbitrary function into any other arbitrary function, i.e., it is a *universal approximator*. This holds true for neural networks with several hidden layers, each having neurons with nonlinear activation functions. While such configurations were shown to be sufficient for the arbitrary mappings of a universal approximator, they were not shown to be necessary.

In the past few years, a simpler network configuration has been shown to be equally effective in performing arbitrary mappings (Cybenko, 1989; Funahashi, 1989; Hornik, 1989). This network has one or more hidden layers with nonlinear activation functions, but the output layer has a linear activation function. Because of these linear activation functions of the neurons in the output (kth) layer, its output vector Φ_k is proportional to the summation vector $\mathbf{I}_k$. Since it is common practice to set the constant of proportionality equal to unity (because this constant would simply scale the weights associated with the output layer), we now have Φ_k equal to $\mathbf{I}_k$, which in turn is equal to the desired output vector T if we set the error vector ε equal to zero.

Such a configuration was explored extensively almost a decade ago by Lapedes and Farber and found to be very useful, especially when the output of the network were analog variables (Lapedes, 1988). This configuration is proposed here as an alternative to conventional backpropagation training that takes advantage of the linearity of the output of the output layer of neurons to speed up training. Generally, this procedure is implemented using a matrix type software program such as MATLAB®[3] so that a set of training vector pairs (X, T), the corresponding layer summation and output vectors $\mathbf{I}_j$, $\mathbf{I}_k$, Φ_j, and Φ_k, and the error vector ε are handled as matrices of vectors for the z pairs of training samples.

Let us apply the methodology and notation of Section 8.3 to such a network with a single hidden layer. (It is equally applicable to the last layer weight matrix in neural networks with more than one hidden layer.) The process starts in the same manner as cumulative backpropagation with the randomization of the weights, the application of the input matrix X of the training set (X, T) containing z patterns to the input layer, and the calculation of the output matrix of the hidden (j-th) layer Φ_j.

Let us assume as a starting point that the error matrix ε is zero. Now we have the target matrix T equal to the output Φ_k, which in turn is equal to $\mathbf{I}_k$ because of the linear activation function. We now have a deterministic input–output relationship for the output layer weight matrix W_k. The middle layer output matrix Φ_j (which we have just calculated) is the input and T

[3]MATLAB is a registered trademark owned by MathWorks Inc., Natick, MA.

(which is obtained from the training set) is the output, both related by

$$T = \Phi_k = W_k^* \Phi_j \tag{8.7-1}$$

which can be solved for the weight matrix W_k either exactly by matrix inversion or approximately by regression. Because T and Φ_j either represent or were calculated from experimental data, Φ_j usually cannot be inverted, and regression is really the only practical method for solving for W_k.

This regression calculation of W_k is a first approximation of the weight matrix W_k. It is dependent on the weight matrix W_j that was created by a randomization process. The critical question then is whether a neural network with such a combination of W_j and W_k can model adequately the process from which the training data set was obtained. Masters (1993, p. 170) has pointed out that

> The essence of neural networks is that they activate hidden neurons based on patterns in the input data. What we are really interested in is the weights that connect the inputs to the hidden layer (and interconnect hidden layers if more than one is used). Once these weights are determined, computation of the weights that lead to the output layer is almost an afterthought.

This view is further enhanced by recent work of Lo (1996) that indicates that the essence of the representation of the model of a system is contained in the weights associated with the hidden layer(s) (whose neurons have nonlinear activation functions), a view consistent with the concept of the hidden layer being a "feature detection" layer as described in Chapter 7. Hence, we need do more than create the hidden layer weights by a randomization process. Indeed, randomization of weights is only a convenient starting condition that avoids some training problems and is useful only when associated with a long training process (e.g., backpropagation) that allows these randomized hidden layer weights to be adjusted sufficiently that they represent adequately the system model. Hence, we need to proceed with a modified form of backpropagation in which a regression method is used to solve for the weight matrix W_k. The weight matrix W_j is then iteratively determined using backpropagation.

One reason that conventional backpropagation training is very slow to converge is that the error terms are propagated back through the output layer weights to the hidden layer to provide an error correction. When these error terms are backpropagated through the non-optimal output layer weights, the changes in the hidden layer weights are far from optimal. Hence, the use of a regression method to provide a good first approximation of W_k, albeit based on randomized weights, greatly speeds up the training process.

Use of Regression to Solve for the Weight Matrix W_k

We can solve for the weight matrix W_k in equation (8.7-1) using a least square error regression method. Given the input matrix Φ_j, the output matrix

I_k and the weight matrix W_k, a standard regression equation for equation (8.7-1) is

$$\Phi_k = W_k \Phi_j \tag{8.7-2}$$

which we can use to solve for the weight matrix using the general least squares procedure

$$W_k = \left(\Phi_j \Phi_j'\right)^{-1} \Phi_j' I_k \tag{8.7-3}$$

In this solution, the mean squared error is minimized with extremely high precision. The weight values produced by this algorithm may contain very large values (5 or 6 orders of magnitude) that result in very poor network generalization. Instead of learning the general trend of the data, the network has also learned the noise. The fit to the training data would be excellent, but the generalization would be very poor. There are several methods by which this least-squares operation may be carried out properly; the method suggested as best by Masters (1993), "Singular Value Decomposition," is discussed below.

At this point, we have a neural network whose state of training is much better than the typical neural network after the first iteration of backpropagation training. We now proceed with conventional backpropagation as described in Section 9.3, modified to include the regression method for calculating W_k. This combination of a good starting state and an algorithm for optimally calculating one layer of weights speeds up the training process dramatically. Application of this methodology has sped up the training of neural networks 40-fold (Uhrig, 1996), and resulted in networks with superior generalization capabilities. After this initial pass through the network, we start again with the application of the input matrix X of the training set to the buffer input layer and calculate our way through the network to produce the hidden layer output matrix Φ_j. Then we use the regression method to calculate the changes in W_k and conventional backpropagation to calculate the change in W_j. Then the weight matrices are updated, and the input matrix X is applied to the input layer, starting a new cycle of this hybrid training process. As the training proceeds, the features of model of the system under study is embedded in the weight matrix W_j associated with the hidden layer(s).

Singular Value Decomposition

The hidden layer output matrix Φ_j is broken down into its singular value composition given by

$$\Phi_j = USV^T \tag{8.7-4}$$

where

$\Phi_j = z \times n$ matrix of hidden layer outputs

$\mathbf{U} = z \times n$ matrix of principle components

$\mathbf{S} = n \times n$ diagonal matrix of single values

$\mathbf{V} = n \times n$ matrix of right singular values (orthonormal matrix)

$\mathbf{T} = z \times n$ matrix of target outputs

In this method, only the most relevant information is retained to compute the weights. The least important information is discarded because it is most likely to result from noise. The amount of noise that is removed from the solution to the system is determined relative to the largest weight expected in the network. This is achieved by altering the diagonal matrix of singular values (S). Singular values in the matrix S that are less than a cutoff value (ϵ), are changed to zero in the inverse matrix. The weights are then calculated by

$$\mathbf{W}_k = \mathbf{V}\mathbf{S}^{-1}\mathbf{U}^T\mathbf{T} \tag{8.7-5}$$

Experimental results show that keeping the magnitude of the weights within ± 10 provides excellent network generalization with no loss of important information.

Adaptation of Models to Changing Conditions

The concentration of the model information in the hidden layer weights pointed out by Lo (1996) as described above also addresses one of the most troublesome problems in using neural networks to model many real-world situations, namely, the nonstationary system with small but often continuous changing of the operating conditions over time. Adapting the neural network model to changing operating conditions without the model representation suffering deterioration can be performed by adjusting the linear output layer weights using the regression method (i.e., singular value decomposition) as described above. This updates effectively the neural network to actual plant conditions (Hines, 1996).

There is, however, risk that using this method for continuous adaptation of the neural network model may mask a continuously deteriorating condition of the system. Hence, it may be useful to have duplicate neural network models, one that is continually adjusted and one that is not adjusted. By monitoring the difference between the outputs of these two models, it is possible to assess whether the changing conditions representation represents deteriorating or normal behavior.

8.8 MODULAR NEURAL NETWORKS[4]

The modular neural network is another rather special extension of the backpropagation network. It is comprised of (a) several independently operating expert networks competing to produce the correct response to individual input vectors and (b) a gating network mediating this competition. Basically, the modular neural network consists of the input layer, a processing "superlayer" comprising the expert networks and the gating network, and the output layer. The input units are fully connected to the input units of both the expert networks and the gating network, but there are no weights in these connections. The expert network output units are fully connected to the modular neural network output units, with connections having weights whose values correspond to the values produced in the output units of the gating network. The output units perform a summation of the weighted incoming signals without applying any activation function to the result. A topology of the modular neural network with two input units, three expert networks, and a single output unit is shown in Figure 8.16.

For the function approximation task the expert networks are typically simple perceptrons (i.e., often backpropagation networks with only two layers) with their output neurons performing only a simple summation without using an activation function. However, the expert networks can also be the classical three-layer backpropagation networks) or even another modular neural network, thus creating a complex hierarchical structure), and their output nodes may use nonlinear activation functions. The structure of the expert networks has to be chosen in such a way that the task of concern cannot be solved by a single expert network, because otherwise the modular neural network would gradually degenerate during training into a network with the structure of this expert network (i.e., all the input vectors would be processed solely by one of the expert networks). In any case, all the expert networks in the modular neural network have to have the same structure with the same number of layers and identical processing units in them.

The gating network is a fully connected feedforward neural network, having typically only two layers. There is one output neuron in the gating network for each expert network in the whole modular neural network (hence, the weights in all connections between the output neurons of each expert network and the output neurons of the whole modular neural network are identical). Similarly to the backpropagation network processing neurons, the gating network output neurons sum the weighted signals received from the input units and filter the result through an activation function. This activation function, however, is the so-called "softmax" function which essen-

[4]Part of this section was extracted from a thesis "Modeling a Probabilistic Safety Assessment Using Neural Networks" by Vaclav Hojny, a graduate student at the University of Tennessee, 1993–1995.

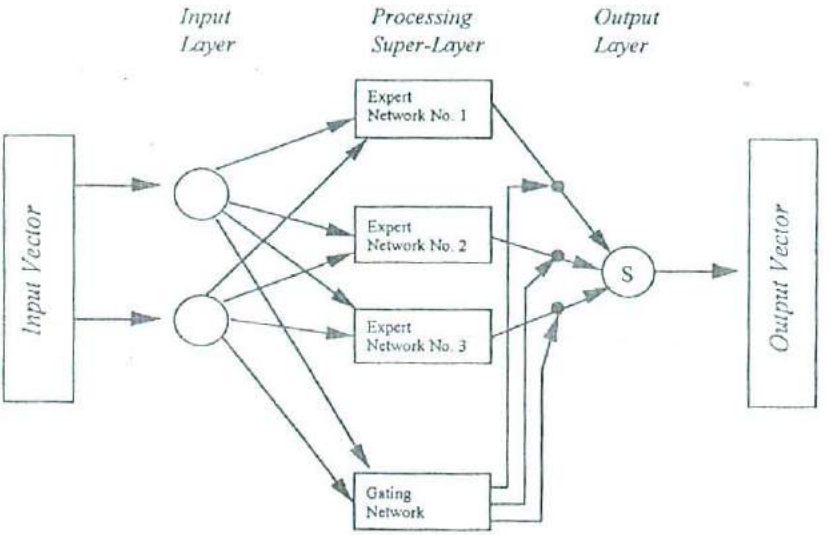


Figure 8.16 Topology of the modular neural network.

tially normalizes activations of output units and amplifies differences between them. The activation function for the softmax function as given in Figure 8.17 is

$$\Phi_N = \frac{e^{I_N}}{\sum_{j=1}^{m} e^{I_j}} \tag{8.8-1}$$

Operating of the modular neural network consists of the following: division of a complex task to be solved into several simpler subtasks, finding of separated solutions for these subtasks, and combination of these subsolutions into the desired solution of the original complex task. This approach is sometimes referred to as a principle of "divide and conquer." To achieve this, the modular neural network utilizes a special combination of supervised and unsupervised training based on maximization of the likelihood function, which represents a product of the probabilities of generating the correct output vectors for individual input–output vector pairs. These probabilities are typically modeled using a mixture model (i.e., a linear combination) of multivariate Gaussian distributions, which characterize conditional probabilities of producing the correct output vectors by the individual expert networks for a given input vector from the training set. To maximize the likelihood function, its known parameters have to be optimized—that is, properly adjusted during training. In this modular neural network, these parameters represent (a) the weights in connections of the expert network processing units and (b) the weights in connections of the expert network output units

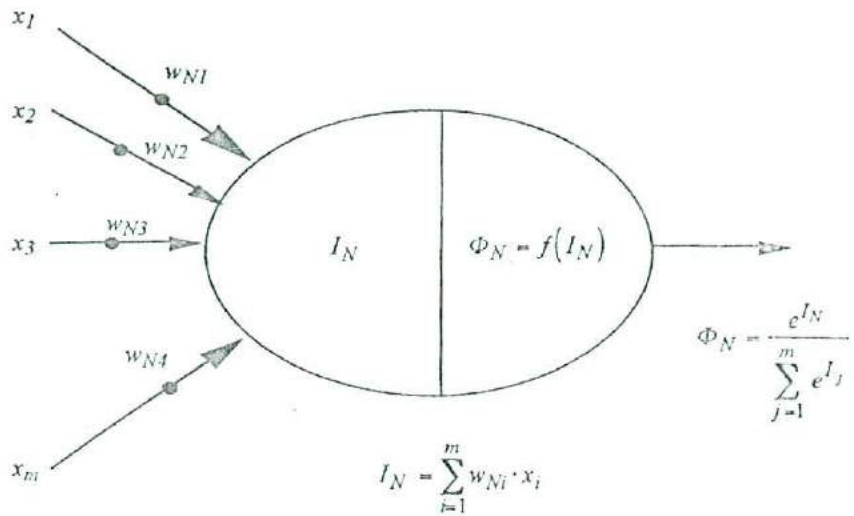


Figure 8.17 Output neuron of the modular neural network gating network.

and the output units of the whole modular neural network. All these weights are trained simultaneously starting from small arbitrary values. During the training, the modular neural network is presented with individual input vectors from the set of training samples to which individual expert networks and gating networks respond with certain output vectors.

The basic training of the expert networks is supervised and utilizes a procedure very similar to that used by the backpropagation neural network. The resulting update values of the weights for each expert network are, however, modified by a probability that the particular expert network is allowed to produce the particular modular neural network output vector. The value of this probability is determined as a product of the distance (typically Euclidean) between the output vector produced by the given expert network and the target output vector multiplied by the value produced by the gating network output corresponding to this expert network. Values of these products, determined for individual expert networks, are then processed through the softmax function with the aim of amplifying the outcome of the competition of the expert networks. The resulting values of the products effectively represent posterior probabilities that the particular expert networks are allowed to produce the particular modular neural network output vector, while the values produced by the corresponding gating network outputs represent prior probabilities of it. The training of the gating network is essentially unsupervised and aimed at minimization of the differences between these prior and posterior possibilities. The training procedure for the gating network is, in fact, the same one used in backpropagation networks. The elements of the target output vector of the gating network are deter-

mined by the posterior probabilities of the corresponding expert networks (which are, to a great extent, independent on the modular neural network target output vector). All the input–output vector pairs in the training set are repeatedly presented to the modular neural network until its error decreases to an acceptable level, or until the modual neural network reaches a steady state when the values of its weights stop changing.

8.9 RECIRCULATION NEURAL NETWORKS

Another variation of the basic backpropagation network is the recirculation neural network (RNN) introduced by Geoffrey Hinton and James McClelland (1988) as a neurally plausible alternative to the autoassociative backpropagation network. They considered backpropagation to be neurally implausible and hard to implement in hardware, because it requires that all connections be used backwards, that these connections be symmetrical, and that the units are different input–output functions for the forward and backward passes. In a backpropagation network, errors are passed backwards through the same connections that are used in the forward pass, but they are scaled by the derivative of the feed forward activation function. In a recirculation neural network, data are processed through weights in only one direction.

The recirculation neural network is a four-layer autoassociative type network as shown in Figure 8.18, in which the input and output layers are buffer layers with the same number of neurons. The other two layers are called the "visible" and the "hidden" layers. In a recirculation neural network, the visible and hidden layers are fully connected to each other in both directions by separate links with separate sets of weights. The visible-to-hidden connections involve what is called the *bottom-up weights*, and the hidden-to-visible connections contain the *top-down weights*. Each neuron in the visible and hidden layers is connected to a bias element with a trainable weight.

The learning schedule involves two complete passes between the visible and hidden layers. The learning is carried out using only local knowledge—that is, the state of the processing element and the input values of the particular connection to be adapted. The purpose of the learning rule is to construct in the hidden layer an internal representation of the data presented at the visible layer. Recirculation neural networks use unsupervised learning, in the sense that no desired vector is required to be present at the output layer. A bias term is used for all neurons in the hidden and visible layers. The learning process proceeds in the following manner. Initially, all weights, including the bias weights, are randomly set to small values. The data are first presented at the visible layer (time 0), then filtered through the bottom-up weights to the hidden layer (time 1), and then circulated back to the visible layer through the top-down weights (time 2). Finally, the data are passed for a second time (recirculated) to the hidden layer through the

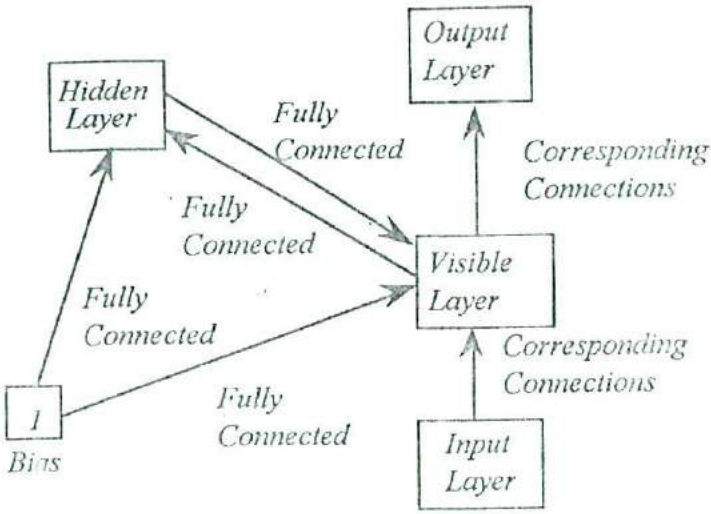


Figure 8.18 Structure of a recirculation neural network.

bottom-up weights (time 3) and back to the visible layer through the topdown weights (time 4) and on to the output buffer layer. If desired, there could be a second output buffer layer connected to the hidden layer where the compressed version of the output could be made available.

Learning occurs only after the second pass through the network. The output of the visible layer at time 2 is the reconstruction of the original input vector from the compressed vector in the hidden layer at time 1. The aim of the learning is to minimize the error between the original input and the reconstructed vector at time 2 by adjusting the top-down weights, as well as to minimize the error between the compressed vectors at times 1 and 3 by adjusting the bottom-up weights. All summations over the hidden layer (times 1 and 3) or visible layer (times 0 and 2) include the bias terms. During training the output of the hidden layer at time 1 is the compressed version of the input data. In Hinton and McClelland's simulations, cumulative learning is used—that is, changes in the weights are accumulated over an epoch—and the actual weights are changed only at the end of an epoch.

The state of the visible layer at time 2 is the top-down response of the network to the initial bottom-up stimulus. Hinton and McClelland used sigmoid functions for the activation functions for both visible and hidden layers, although their analysis assumes that the activation function for the hidden layers is linear and that the activation function for the visible layer is any smooth monotonic nonlinear function with bounded derivatives.

The recirculation neural network has full connectivity between the hidden and visible layers in both directions. Learning for the top-down weights in the connections from the jth hidden layer to the ith visible layer and bottom-up

weights in the connections from the ith visible layer to the jth hidden layer are given by

$$\Delta w_{ji} = \varepsilon_{y_j}^{(1)}\left[y_i^{(0)} - y_i^{(2)} \right] \qquad \text{(top down weight change)} \qquad (8.9\text{-}1)$$

and

$$\Delta w_{ij} = \varepsilon_{y_i}^{(2)}\left[y_j^{(1)} - y_j^{(3)} \right] \qquad \text{(bottom up weight change)} \qquad (8.9\text{-}2)$$

where $y_i^{(0)}$ is the state of the ith visible neuron at time 0, $y_i^{(2)}$ is the state of the ith visible neuron after the activity has passed around the loop once, $y_j^{(1)}$ is the state of the jth visible neuron at time 1 (first pass around the loop), $y_j^{(3)}$ is the state of the jth neuron at time 3 (second pass around the loop), and $\varepsilon_{y_j}^{(1)}$ and $\varepsilon_{y_i}^{(2)}$ are the errors after recirculation.

This learning process for the recirculation neural network approaches gradient descent under certain specific conditions. The error at each processing element in the visible layer between its state at time 0 and at time 2 is referred to as the "reconstruction error." The learning in the top-down weights seeks to reduce the reconstruction error. Hinton and McCleland have shown that under certain conditions the learning in the bottom-up weights also performs gradient descent learning in the reconstruction error. For the visible-to-hidden connections, the changes are partially related to the gradient descent. Early changes do not necessarily improve the state of the system, but as learning progresses, these changes tend to agree with the gradient descent and total agreement occurs after the hidden-to-visible weights are approximately aligned with the visible-to-hidden weights.

Example 8.3 An Application of Recirculation Neural Networks[5]. One of the applications of the recirculation neural network is to transform narrow peaks in a Fourier transform of undamped vibration data of rotating machinery into a pattern where the information in the peaks is spread over the entire frequency range. An example of the influence of small changes in the amplitude and the frequency for a single narrow peak is shown in the next four figures. In Figure 8.19 we have a single peak at a specific frequency in which the amplitude decreases for the series of six examples. When these six peaks are subject to transformation by the use of the RNN, the results are shown in Figure 8.20, where the individual values in the spectrum have been connected. It is clear that the information contained in the single peak has been spread throughout the frequency range and that the shape changes as the amplitude is decreasing. However, this transformation is even more drastic when there is a small shift in the frequency. In this case, small changes in the frequency in the original spectrum in Figure 8.21 produces drastic changes in the transformed spectrum shown in Figure 8.22. □

[5]Results presented in Figures 8.19 through 8.22 were produced by Dr. Israel E. Alguidingue, when he was a graduate student at the University of Tennessee, 1989–1993.

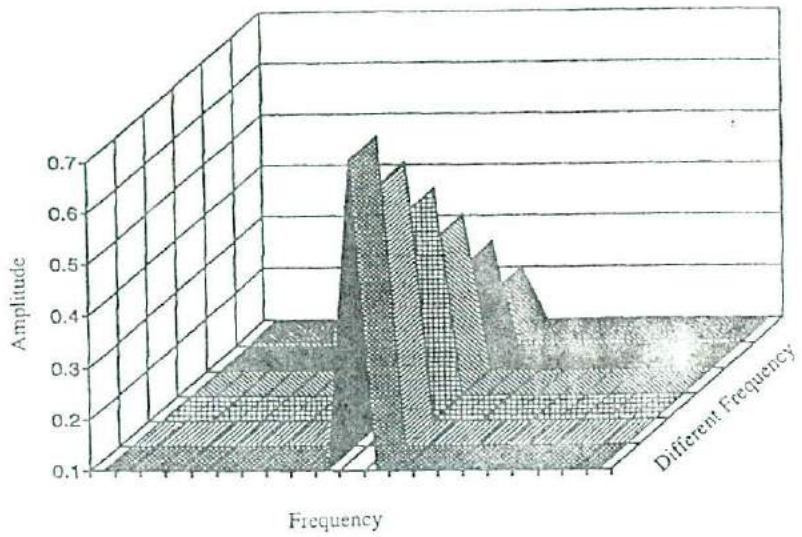


Figure 8.19 Single peaks in a power spectral density plot for several different amplitudes at a specific frequency.

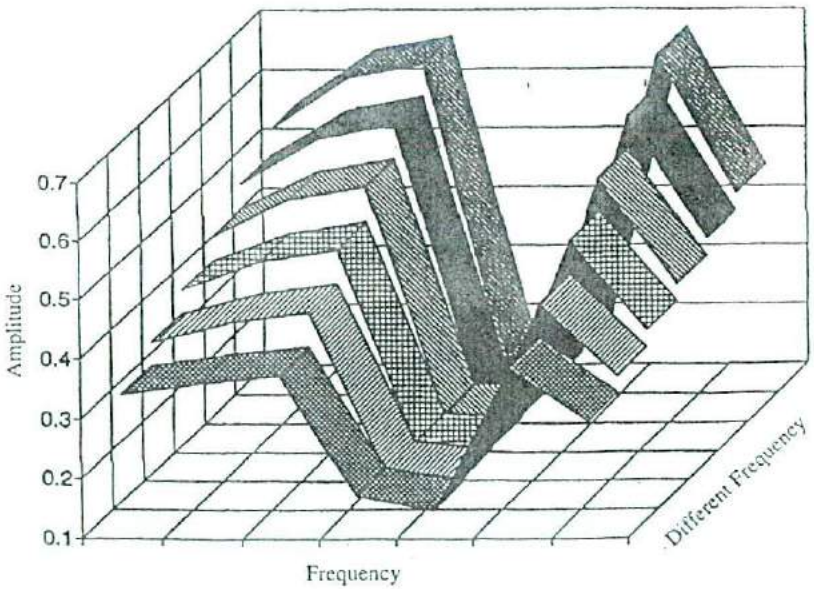


Figure 8.20 Results of processing of peaks in Figure 8.14 with a recirculation neural network.

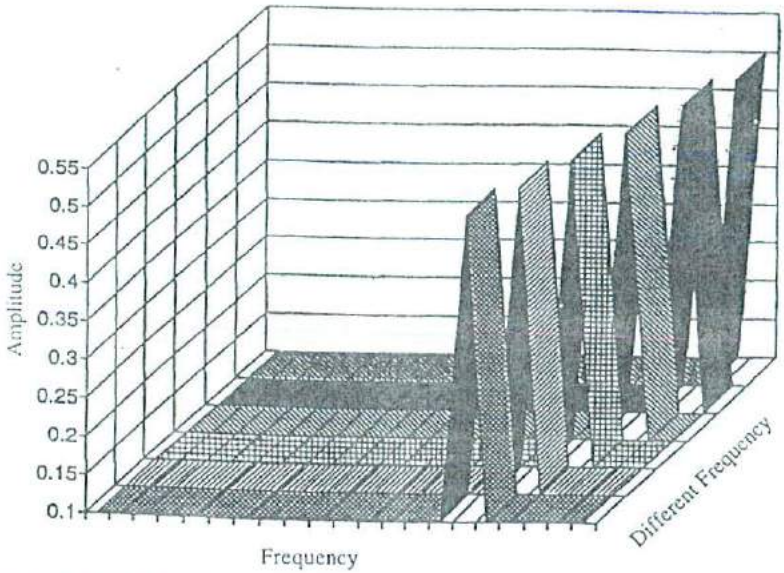


Figure 8.21 Single peaks with the same height in power spectral density plots for several frequencies.

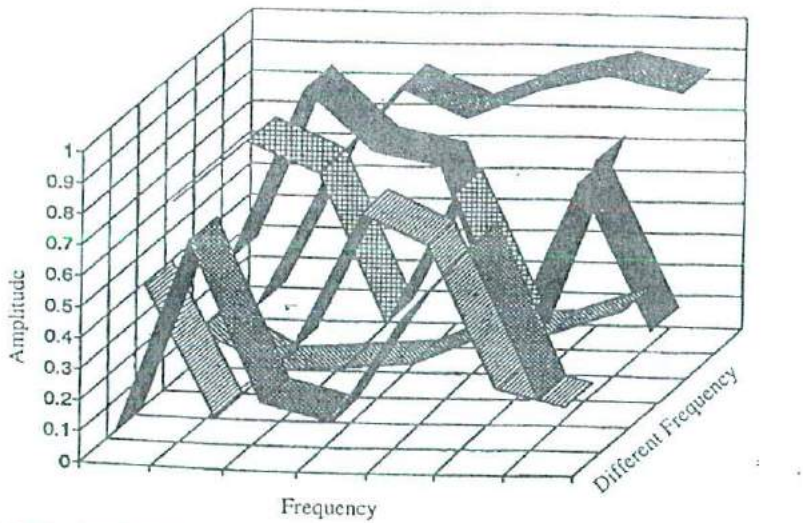


Figure 8.22 Results of processing of peaks in Figure 8.17 with a recirculation neural network.

8.10 FUNCTIONAL LINKS

In conventional backpropagation networks, weights are applied only to connecting links leading to neurons. The hidden layer in backpropagation networks provides the ability to form complex relationships between input pattern elements. However, if the data pattern presented is in a form that already has complex elements in the form of functional links, then the hidden layer may not be necessary. In functional link networks, developed by Yoh-Han Pao (1989) connections (links) provide information to the network by incorporating a representation of the relationships between the input and output patterns. This involves adding inputs that are functions of the normal inputs. While it is possible for backpropagation to learn complex relationships (e.g., $x^2, xy, \cos^2(x), \sin(x)$, etc.), functional link networks establish these relationships directly. The difficulty is knowing which functions to use. Generally, this requires an understanding of the nature of the problem involved. If the problem can be represented by a polynomial, then simple power and cross terms (e.g., x^2, xy, x^2y^3, etc.) may be appropriate as additional inputs to the network. If a problem has cyclic terms for frequencies that are important, then sine and cosine terms may be appropriate. Functional link networks are feedforward networks that use standard backpropagation training. Clearly, the output layer neurons must have nonlinear activation functions if there are only two layers in the network.

There are two kinds of functional links. The first type is the outer product (tensor) model where each component of the input is multiplied by the entire input vector $x_i (1 \leq i \leq n)$ or $x_i x_j$, where $i \leq j \leq n$ and $i \leq j$. Representation of the input space is enhanced, making it easier for the model to learn.

The second general type of functional link is functional expansion where the input variables are individually acted upon by the appropriate functions —that is, $\sin(x)$, $\cos(x)$, $\sin(2x)$, and so on. The functions selected may be a subset of the orthonormal basic functions. The overall effect is to map the input vector into a larger pattern space, enhancing the representation. Of course, it is possible to combine the tensor and functional expansion types of functional links.

Although functional links offer an attractive analytical alternative to the general problem of specifying the architecture of a network, they have their own limitations. The principal concern with functional links are the following:

1. As the number of inputs increases, the number of connections (with weights) increases.
2. There are indications that a smaller number of training examples relative to the number of connections can influence the ability of the network to generalize.

3. With fewer examples per connection, the model may learn to reproduce the training set errors and not generalize. This problem is especially difficult with noisy data.

8.11 CASCADE-CORRELATION NEURAL NETWORKS

Cascade-correlation is a supervised learning algorithm for neural networks that adjusts the network architecture as well as the weights (Fahlman and Lebiere, 1990). Cascade-correlation starts with a minimal network and adds new hidden neurons one-by-one, creating a multilayer network in the learning process. Once a new hidden neuron has been added to the network, its input-side weights are fixed, and it becomes a permanent part of the network, helping to serve the function of hidden layers (i.e., feature detection). This architecture attempts to overcome the issues which cause backpropagation to be so slow in training a neural network. Two specific issues that are addressed are (1) the step-size problem and (2) the moving target problem.

The "step problem" arises in backpropagation because only infinitesimal changes during the learning process (which implies an infinite training time) can reasonably ensure that a global minimum can be reached. Large changes, which would speed up the training process, tend to cause backpropagation to reach local minima; and various methods, such as the use of momentum of simulated annealing, must be used to reach a global minimum.

The "moving target problem" arises because each neuron in the interior of the network is trying to evolve into a feature detector that will play some useful role in the network's overall computation, but its task is complicated by the fact that it cannot communicate to other neurons to which it is connected (both directly and indirectly), which are changing all the time. One way of decreasing the moving target problem is to allow only a few of the weights in the network to change at once. In a sense, this is reducing the dimensionality of the training process.

There are two related primary features of the cascade-correlation training process: (1) the cascade architecture in which hidden neurons with fixed (nontrainable) inputs are added to the network one at a time and (2) the learning algorithm which creates and installs the new hidden neurons. All neurons have bias inputs with trainable weights and nonlinear activation functions which may be any of the sigmoidal functions. All weights are initially randomized between -1 and $+1$.

The cascade-correlation neural network starts as a two-layer, fully connected perceptron with adjustable weights on every connection which are initially randomized. The direct input–output connections are trained using the Widrow–Hoff delta training rule (or any other training algorithm for two layer networks such as Quickprop). Training is terminated when the weight values approach an asymptotic value, based on a "patience" parameter set by

the used. Then the overall error is measured, and a decision is made as to whether to continue training.

Training is continued by adding a single neuron to create a hidden layer. It is connected to all input neurons through connections with fixed weights and to all output neurons through trainable (randomized) weights. The fixed input weighs of the new neuron are set by a "pretraining" process before the outputs are connected to the output layer. In this pretraining process, a number of training sets are applied to this single neuron, and the input weights are adjusted after each pass to maximize the sum (over all outputs) of the magnitude of the correlation between the neurons output and the residual output error of the neuron.

Training proceeds the same as previously because the fixed input weights allow the network to be treated as if it were a two-layer network. When the training stops because the "patience" parameter is reached, the overall error is calculated and a decision is again made whether to proceed. If so, another single neuron is added, in the same manner as the first neuron, fully connected with fixed weights (set by the optimization pretraining process described above) to the inputs, and fully connected with trainable (randomized) weights to the output layer. However, the only connection between the two added neurons is from the output of the first neuron to the input of the second neuron through a fixed weight. This means that we have effectively added a second hidden layer with a single neuron.

The process of adding one neuron at a time continues until the user is satisfied with the overall error of the network for the training data. The multitude of single-neuron hidden layers presents a very powerful feature detector, but it also leads to a large fanout of the input connections and a very "deep" network. Fahlman and Lebiere (1990) indicate that strategies for addressing these issues are being investigated.

8.12 RECURRENT NEURAL NETWORKS

The backpropagation neural networks previously discussed are strictly "feedforward" networks in which there are no feedbacks from the output of one layer to the inputs of the same layer or earlier layers of neurons. However, such networks have no memory since the output at any instant is dependent entirely on the inputs and the weights at that instant.

There are situations (e.g., when dynamic behavior is involved) where it is advantageous to use feedback in neural networks. When the output of a neuron is fed back into a neuron in an earlier layer, the output of that neuron is a function of both the inputs from the previous layer at time t and its own output that existed at an earlier time—that is, at time $(t - \Delta t)$, where Δt is the time for one cycle of calculation. Hence, such networks exhibit characteristics similar to short-term memory, because the output of the network depends on both current and prior inputs.

Neural networks that contain such feedback are called *recurrent neural networks*. Although virtually all neural networks that contain feedback could be considered as recurrent networks, the discussion here will be limited to those that use backpropagation for training (often called "recurrent backprop networks"). Let us consider the elementary feedforward network shown in Figure 8.23*a*, where the input, middle, and output layers each have only one neuron, and where neuron h is a buffer neuron that instantaneously send the input x to neuron p. When the input $x(0)$ (x at time 0) is applied to the input, the outputs of neurons p and q at time (0), $v(0)$, and $y(0)$, respectively, are

$$v(0) = \{\Phi[w_{ij}x(0)]\} \tag{8.12-1}$$

$$y(0) = \Phi\{w_{jk}[v(0)]\} = \Phi\{w_{jk}\{\Phi[w_{ij}x(0)]\}\} \tag{8.12-2}$$

where Φ is the activation function operator (usually a sigmoidal function) and (0) indicates the value at time 0.

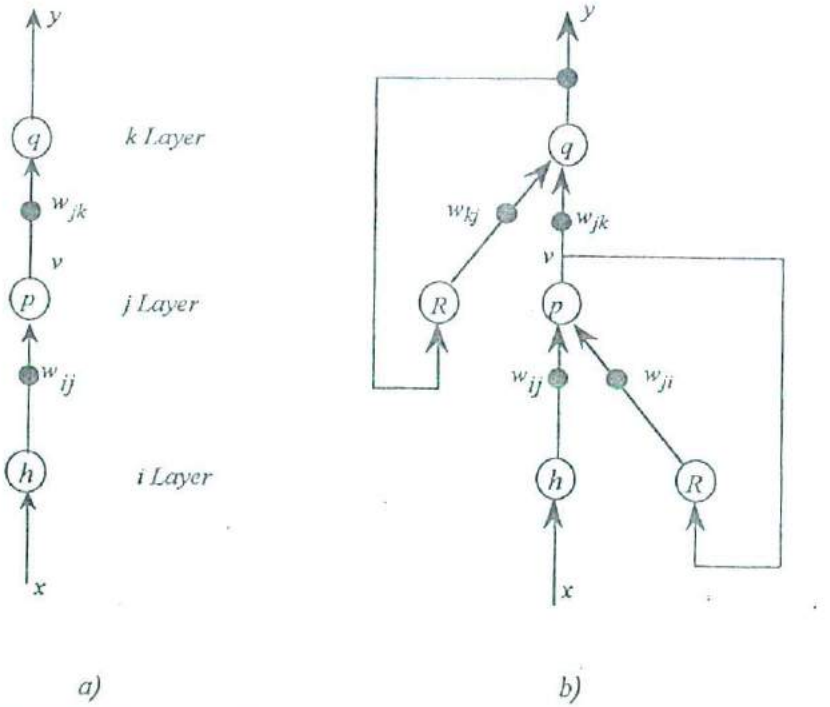


Figure 8.23 Simple neural networks without (*a*) and with (*b*) recurrent feedback.

For nonrecurrent networks (no feedback), this relationship remains valid for times $1, 2, 3, \ldots, n, \ldots, N$, and so on. Hence above equation becomes

$$v(n) = \{\Phi[w_{ij}x(n)]\} \tag{8.12-3}$$

$$y(n) = \Phi\{w_{jk}[v(n)]\} = \Phi\{w_{jk}\{\Phi[w_{ij}x(n)]\}\} \tag{8.12-4}$$

To make this network recurrent, we add feedback from the output neuron to the middle layer and from the middle neuron to the input layer through the recurrent neurons (which are buffer neurons) labeled R and the corresponding weights w_{kj} and w_{ji}, respectively, as shown in Figure 8.23*b*. The outputs to neurons p and q must exist before there can be any feedback. Hence, as the process proceeds step by step, the feedback term of neurons p and q will not come into play until time 1. Hence, the equation (8.11-2) for $y(0)$ is valid for a recurrent network for time 0, but the feedback terms must be added for all subsequent times. The output of the network for neurons p and q at time 1 are

$$\begin{aligned} v(1) &= \Phi\{w_{ij}[x(1)] + w_{ji}[v(0)]\} \\ &= \Phi\{w_{ij}x(1) + w_{ji}\{\Phi[w_{ij}x(0)]\}\} \end{aligned} \tag{8.12-5}$$

$$\begin{aligned} y(1) &= \Phi\{w_{jk}v(1) + w_{kj}y(0)\} \\ &= \Phi\{w_{jk}\{\Phi\{w_{ij}x(1) + w_{ji}\{\Phi[w_{ij}x(0)]\}\} \\ &\qquad + w_{kj}\{\Phi\{w_{jk}\Phi[w_{ij}x(0)]\}\}\}\} \end{aligned} \tag{8.12-6}$$

For time 2, the outputs of neurons p and q are

$$\begin{aligned} v(2) &= \Phi\{[w_{ij}x(2)] + [w_{ji}v(1)]\} \\ &= \Phi\{[w_{ij}x(2)] + w_{ji}\{\Phi[w_{ij}x(1)] + w_{ji}\{\Phi[w_{ij}x(0)]\}\}\} \end{aligned} \tag{8.12-7}$$

$$\begin{aligned} y(2) &= \Phi\{[w_{jk}v(2)] + [w_{kj}y(1)]\} \\ &= \Phi\{w_{jk}\{\Phi[w_{ij}x(2)] + w_{ji}\{\Phi[w_{ij}x(1)] + w_{ji}\{\Phi[w_{ij}x(0)]\}\}\} \\ &\qquad + w_{kj}\{\Phi\{w_{jk}\{\Phi[w_{ij}x(1)] + w_{ji}\{\Phi[w_{ij}x(0)]\}\}\} \\ &\qquad\qquad + w_{kj}\{\Phi\{w_{jk}\{\Phi[w_{ij}x(0)]\}\}\}\}\} \end{aligned} \tag{8.12-8}$$

For time 3, the outputs of neurons p and q are

$$\begin{aligned} v(3) &= \Phi\{[w_{ij}x(3)] + [w_{ji}v(2)]\} \\ &= \Phi\Big\{[w_{ij}x(3)] + w_{ji}\Big\{\Phi\Big\{[w_{ij}x(2)] + w_{ji}\Big\{\Phi[w_{ij}x(1)] \\ &\quad + w_{ji}\{\Phi[w_{ij}x(0)]\}\Big\}\Big\}\Big\}\Big\} \end{aligned} \tag{8.12-9}$$

$$\begin{aligned} y(3) &= \Phi\{[w_{jk}v(3)] + [w_{kj}y(2)]\} \\ &= \Phi\Big\{w_{jk}\Big\{\Phi\Big\{[w_{ij}x(3)] + w_{ji}\Big\{\Phi\Big\{[w_{ij}x(2)] + w_{ji}\Big\{\Phi[w_{ij}x(1)] \\ &\quad + w_{ji}\{\Phi[w_{ij}x(0)]\}\Big\}\Big\}\Big\}\Big\} + w_{kj}\Big\{\Phi\Big\{w_{jk}\Big\{\Phi[w_{ij}x(2)] \\ &\quad + w_{ji}\Big\{\Phi[w_{ij}x(1)] + w_{ji}\{\Phi[w_{ij}x(0)]\}\Big\}\Big\}\Big\} \\ &\quad + w_{kj}\Big\{\Phi\Big\{w_{jk}\{\Phi[w_{ij}x(1)] + w_{ji}\{\Phi[w_{ij}x(0)]\}\}\Big\} \\ &\quad + w_{kj}\Big\{\Phi\Big\{w_{jk}\{\Phi[w_{ij}x(0)]\}\Big\}\Big\}\Big\}\Big\} \end{aligned} \tag{8.12-10}$$

Magnitude of Terms

Note that the equation (8.11-10) for $y(3)$ has $x(0)$, $x(1)$, $x(2)$, and $x(3)$ as inputs. An equation for $y(4)$ would add $x(4)$ to the list of inputs; $y(5)$ would add $x(5)$, and so on. Furthermore, an examination of the terms of the equation for $y(3)$ indicates that the magnitudes of the earlier inputs decrease when later inputs are added. This is seen more readily if we assume that the activation function is linear—that is,

$$\Phi\{x\} = x \tag{8.12-11}$$

Then the equations for $y(0)$, $y(1)$, $y(2)$, and $y(3)$ become

$$y(0) = w_{jk}w_{ij}[x(0)] \tag{8.12-12}$$

$$y(1) = w_{jk}w_{ij}\{[x(1)] + [w_{ji} + w_{kj}][x(0)] \tag{8.12-13}$$

$$\begin{aligned} y(2) &= w_{jk}w_{ij}\{[x(2)] + [w_{ji} + w_{kj}][x(1)] \\ &\quad + [w_{ji}^2 + w_{ji}w_{kj} + w_{kj}^2][x(0)] \end{aligned} \tag{8.12-14}$$

$$\begin{aligned} y(3) &= w_{jk}w_{ij}\Big\{[x(3)] + [w_{ji} + w_{kj}][x(2)] \\ &\quad + [w_{ji}^2 + w_{ji}w_{kj} + w_{kj}^2][x(1)] \\ &\quad + [w_{ji}^3 + w_{ji}^2w_{kj} + w_{ji}w_{kj}^2 + w_{kj}^3][x(0)]\Big\} \end{aligned} \tag{8.12-15}$$

Since the weights are usually less than 1, the increasing power of the weights for the earlier terms [w^2 for $x(3)$, w^3 for $x(2)$, w^4 for $x(1)$, and w^5 for $x(0)$] causes the coefficients to decrease rapidly. When the sigmoidal (or any nonlinear activation function) is used, the presence of multiple nonlinear activation functions in the above equations for $y(0)$, $y(1)$, $y(2)$, and $y(3)$, each with a maximum value of 1, reduces the early terms much faster than for a linear activation function. As later inputs are introduced, the influence of the earlier terms become negligible. This reduced weighting of the earlier terms is analogous to the decrease of influence of earlier values in a convolution transformation.

The complexity introduced by feedback connections, even for the elementary system of Figure 8.23*a*, is readily apparent. For networks more complicated than the elementary system in Figure 8.23*b*, the same principles are applied, but the complexity grows even more rapidly. However, the increase in complexity is often compensated for because the feedback almost always drastically reduces the number of cycles needed to train a neural network significantly. Feedback can often be used advantageously to speed up the training of a neural network and to avoid local minima. Indeed, it is sometimes possible to train a neural network after feedback has been added, whereas it may not have been previously possible to train it to the desired level of error. However, capturing dynamic behavior in a model is the most common justification for the use of feedback, and recurrent neural networks (or neural networks with delayed inputs) are almost always used for dynamic signals.

REFERENCES

Amari, S.-I, Characteristics of Random Nets of Analog Neuron-like Elements, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. SMC-2, pp. 643–647, 1972.

Fahlman, S. E., Faster-Learning Variations of Backpropagation: An Empirical Study, *Proceedings of the 1988 Connectionist Models Summer School*, Morgan Kaufmann, San Mateo, CA, 1988.

Fahlman, S. E., and Lebiere, C., The Cascade-Correlation Learning Architecture, in *Advances in Neural Information Processing Systems* 2, D. Touretzky, ed., Morgan Kaufmann, San Mateo, CA, 1990, pp. 534–532.

Guha, A., Neural Network Based Inferential Sensing and Instrumentation, in *Proceedings of the* 1992 *Summer Workshop on "Neural Network Computing for the Electric Power Industry*, D. J. Sobajic, ed., Stanford, CA, August 17–19, 1992a.

Guo, Z., and Uhrig, R. E., Use of Artificial Neural Networks to Analyze Nuclear Power Plant Performance, *Nuclear Technology*, Vol. 99, pp. 36–42, July 1992.

Hashem, S., Sensitivity Analysis for Feedforward Artificial Neural Networks with Differentiable Activations Functions, in *Proceedings of the IJCNN*, Baltimore, MD, June 1992.

Hines, J. W., Wrest, D. J., and Uhrig, R. E., "Plant-Wide Sensor Calibration Monitoring," Proceedings of the *1996 IEEE International Symposium on Intelligent Control*, Detroit, MI, Sept. 15–18, 1996.

Hinton, G., and McClelland, J., Learning Representations by Recirculating," in *Proceedings of the 1987 IEEE Conference on Neural information Processing Systems—Natural and Synthetic*, American Institute of Physics, New York, 1988, pp. 358–366.

Hojny, V., Modeling a Probabilistic Safety Assessment Assessment Using Neural Networks, M. S. Thesis, University of Tennessee Library, Knoxville, TN, 1995.

Jacobs, R. A., Increased Rates of Convergence Through Learning Rate Adaptation, *Neural Networks*, Vol. 1, pp. 295–307, 1988.

Kramer, M. A. Nonlinear Principal Component Analysis Using Autoassociative Neural Networks, *Journal of the American Institute of Chemical Engineers*, Vol. 37, pp. 233–243. 1991.

Kramer, M. A. Autoassociative Neural Networks, *Computers in Chemical Engineering*, Vol. 16, No. 4, pp. 313–328, 1992.

Masters, T. *Practical Neural Network Recipes in C + +*, Academic Press, San Diego, CA, 1993.

Minai, A. A., and Williams, R. D., Acceleration of Backpropagation Through Learning and Momentum, in *Proceedings of the International Joint Conference on Neural Networks*, Vol. 1, 1990, pp. 676–679.

Pao, Y-H., *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley, Reading, MA 1989.

Parker, D., Learning Logic, Invention Report, S81-64, File 1, Office of Technology Licensing, Stanford University, Palo Alto, CA, 1972.

Parker, D., Optimal Algorithms for Adaptive Networks: Second Order Back Propagation, Second order Direct Propagation, and Second Order Hebbian Learning, *Proceedings of the IEEE First International Conference on Neural Networks*, Vol. II, San Diego, CA, 1987, pp 593–600.

Poggio, T., and Girosi, F., Regularization Algorithms for Learning that Are Equivalent to Multiple-Layer Networks, *Science*, Vol. 247, pp. 978–982, 1990.

Rogers, S. K., and Kabrisky, M., *An Introduction to Biological and Artificial Neural Networks for Pattern Recognition*, SPIE Optical Engineering Press, Belingham, WA, 1992.

Rumelhart, D. E., Hinton, G. R., and Williams, R. J., Learning Internal Representations by Error Propagation, in *Parallel Distributed Processing*, Vol. 1, D. E. Rumelhart, and J. L. McClelland, eds., MIT Press, Cambridge, MA, 1986.

Samad, T., Backpropagation is Significantly Faster If the Expected Value of the Source Unit Is Used for Update, *The International Network Society Conference Abstracts*, 1988.

Samad, T., Backpropagation Extensions, Honeywell SSDC Technical Report, 1000 Boone Ave. N., Golden Valley, MN 55427.

Sejnowski, T., and Rosenberg, C., Parallel Networks that Learn to Pronounce English Text, *Complex Systems*, Vol. 1, pp. 145–168, 1987.

Stornetta, W., and Huberman, B., An Improved Three-Layer Backpropagation Algorithm, *Proceedings of the IEEE First International Conference on Neural Networks*, Vol. II, San Diego, CA, 1987, pp. 637–644.

Tsoukalas, L. H., Ikonomopoulos, A., and Uhrig, R. E., Generalized Measurements in Neuro-Fuzzy Systems, in *Proceedings of the International Workshop on Neural Fuzzy Control*, Muronan, Japan, March 22–23, 1993.

Uhrig, R. E., Hines, J. W., Black, C., Wrest, D. J., and Xu, X., "Instrumentation Surveillance and Calibration Verification System, Final Report, Sandia National Laboratory contract AQ-6982 (Contractor: University of Tennessee), March 1996.

Upadhyaya, B. R., and E. Eryurek, Application of Neural Networks for Sensor Validation and Plant Monitoring, *Journal of Nuclear Technology*, Vol. 97, pp. 170–176, February 1992.

Wald, A., Sequential Tests on Statistical Hypotheses, *Annals of Mathematical Statistics*, Vol. 16, pp. 117–186, 1945.

Werbos, J. P., *The Roots of Backpropagation*, John Wiley & Sons, New York, 1994.

Werbos, P., Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences, Ph.D. Dissertation, Harvard University, Boston, MA, 1974.

Wrest, D., Hines, J. W., and Uhrig, R. E., Instrument Surveillance and Calibration Verification to Improve Nuclear Power Plant Reliability and Safety Using Autoassociative Neural Networks, *Proceedings of the International Atomic Energy Agency Specialist Meeting on Monitoring and Diagnosis Systems to Improve Nuclear Power Plant Reliability and Safety*, Barnwood, Gloucester, United Kingdom, May 14–17, 1996.

PROBLEMS

1. The backpropagation training algorithm is developed in Section 8.3, using the logistic function as the activation function where the derivative has the convenient form given in Equation 8.1-4. Derive the backpropagation training algorithm for the case where the activation function is an arctan function where the derivative is given by Equation 8.1-6.

2. Derive the backpropagation training algorithm for the case where the neurons in the hidden layer have a logistic function for the activation function and the neurons in the output layer have linear activation functions. Compare the results with those obtained in Section 8.7 where this arrangement of activation functions are used.

3. In the recurrent neural network of Section 8.12, the feedback of both loops comes into action at time 1. If the feedback from the output neuron does not come into action until time 2, how will the Equations 8.11-3 through 8.11-15 change?

4. Discuss the results shown in Figures 8.19 through 8.22 for a recirculation neural network and their implications.

5. In the operation of backpropagation, contrast how the changes of learning constant, changes of momentum coefficient, and changes in the α value in the logistic function influence the time required to train a neural network.

6. Derive the backpropagation training algorithm for the simple recurrent neural network shown in Figure 8.23b using the approach given in Section 8.3.

7. Discuss the role of the "bottleneck" layer in a five-layer autoassociative neural network with respect to the identification of principal components. (*Hint.* see references by Kramer and McAvoy at the end of this chapter.)

8. Compare the reduced representation of an input vector in the hidden layer of a recirculation neural network with the reduced representation of in input vector in the middle layer of three- and five-layer autoassociative neural networks.

9

COMPETITIVE, ASSOCIATIVE, AND OTHER SPECIAL NEURAL NETWORKS

9.1 HEBBIAN LEARNING

Donald Hebb (1949) introduced a nonmathematical statement of biological learning in 1949. The Hebbian system was the first truly self-organizing system developed. Even today, it is very prevalent throughout the neural network field because there are many paradigms based on Hebbian learning. Hebb's law can be summarized as follows:

> As A becomes more efficient at stimulating B during training, A sensitizes B to its stimulus, and the weight on the connection from A to B increases during training as B becomes sensitized to A.

One problem with Hebb's law is that it is too vague. Questions such as the following arise: How much should a weight increase? How active does B need to be for training to occur? Furthermore, there is no way for the weights to decrease. In theory, they can increase to infinity. Inhibitory synapses are not allowed, whereas it is well known that real biological systems clearly have inhibitory (negatively weighted) connections.

Corrections to Hebb's law involve normalizing the weights to force them to stay within limited bounds and forcing them to both increase and decrease to retain the normalization. There are many variations of Hebbian learning that are utilized. One of these is the Neo-Hebbian learning put forward by Steven Grossberg, who developed an explicit mathematical statement for the weight change law of the form

$$w_{AB}^{\text{new}} = w_{AB}^{\text{old}}(1 - \alpha) + \beta x_B x_A \qquad (9.1\text{-}1)$$

where w_{AB} is the weight on the synapse connecting neuron A with neuron B, α is the "forgetting" term that accounts for the fact that biological systems forget slowly with time, and β is a "learning" constant that accounts for simultaneous firing of neurons A and B. The right-hand term is called the "Hebbian learning term" because it ties the learning rate to the product of the neuron outputs. Hebbian learning is characterized by the product of two neuron activities. Hence, anytime such a product appears in an equation, Hebbian learning is involved. Generally, both α and β are in the range between 0 and 1.

If we rearrange equation (9.1-1) and put it into the form

$$\frac{w_{AB}^{\text{new}} - w_{AB}^{\text{old}}}{\Delta t} = \frac{\Delta w_{AB}}{\Delta t} = \frac{dw_{AB}}{dt} = -\alpha w_{AB}^{\text{old}} + \beta x_B x_A \qquad (9.1\text{-}2)$$

and consider only the terms involving w_{AB}, it is clear that the forgetting term involves a slow exponential decay with time constant α. Even so, neo-Hebbian learning does not permit the weights to decrease when the neuron outputs decrease.

To overcome this problem, Grossberg introduced differential Hebbian learning. It has the same mathematical form as Hebbian learning of equation (9.1-1) except that it uses the product of rates of change in the outputs for the neurons A and B, as given

$$w_{AB}^{\text{new}} = w_{AB}^{\text{old}}(1 - \alpha) + \beta \frac{dx_B}{dt} \frac{dx_A}{dt} \qquad (9.1\text{-}3)$$

9.2 COHEN–GROSSBERG LEARNING

Pavlov's Experiments

Cohen–Grossberg learning comes from an attempt to mathematically explain the observations from psychological conditioning experiments carried out by Pavlov. Let us consider the various types of conditioning. The first is "observational conditioning," which involves copying the actions of others and is sometimes described as "monkey see, monkey do." The second type of conditioning is "operational conditioning," which involves an action and a response. It is described as "push button, get food." The third type of conditioning is "classical conditioning," and it involves a stimulus and a response. The experiments of Pavlov fall into the classical conditioning category.

The psychological model used in the Cohen–Grossberg learning is Pavlovian learning in which a dog is offered food at the same time a bell rings. Eventually, the dog associates food with the bell ringing and salivates when the bell rings even when no food is presented. This behavior is illustrated in

Table 9.1 Stimuli and corresponding responses in Pavlov's experiment

Stage	*Stimulus*	*Response*
1.	Unconditional Stimulus ====> (Plate of Food)	Unconditioned Response (Dog Salivates)
2.	Unconditional Stimulus (Plate of Food) *plus* ====> Conditioned Stimulus (Bell Rings)	Conditioned Response (Dog Salivates)
3.	Conditioned Stimulus ====> (Bell Rings)	Conditioned Response (Dog Salivates)

Example 9.1 to be Grossberg outstar learning. The three stages of Pavlovian learning are shown diagrammatically in Table 9.1.

Instars and Outstars

Next, we must develop the concept of "instars' and "outstars" to explain Pavlovian conditioning or learning. Every neuron receives hundreds or thousands of inputs through its own synapses from the axon collaterals of other neurons. Schematically, this can be represented as a "star" with radially inward paths called the instar. Indeed, every artificial neuron in a neural network is an instar.

Every neuron also sends out hundreds or thousands of collaterals which branch off from the main axon and go to the synapses of other neurons. This also can be represented by a "star" with radial outward paths. This configuration is called an "outstar," and again every neuron is effectively an outstar. A geometrical interpretation of the instar and outstar configurations is shown in Figure 9.1. The instar has many inputs and a single output whereas the outstar has a single input and many outputs. The fanout of a neuron in the

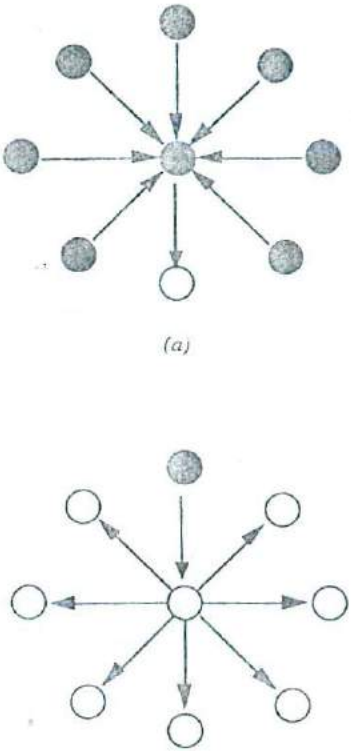


Figure 9.1 Graphical representations of an "instar" (*a*) and an "outstar" (*b*).

input layer of a neural network can be considered as an outstar, whereas a neuron in the output layer can be considered as an instar.

Development of Cohen–Grossberg Learning Equations—Instar Activity

Let us consider Pavlovian learning from the perspective of an instar. The activity of the instar processing element or neuron has a number of requirements:

1. The activity must grow when there is an external stimulus.
2. It must rapidly decrease if it is no longer stimulated from the outside.
3. It must respond to stimuli from other neurons in the network.

Let us consider the arrangement in Figure 9.2, in which an instar y_j receives

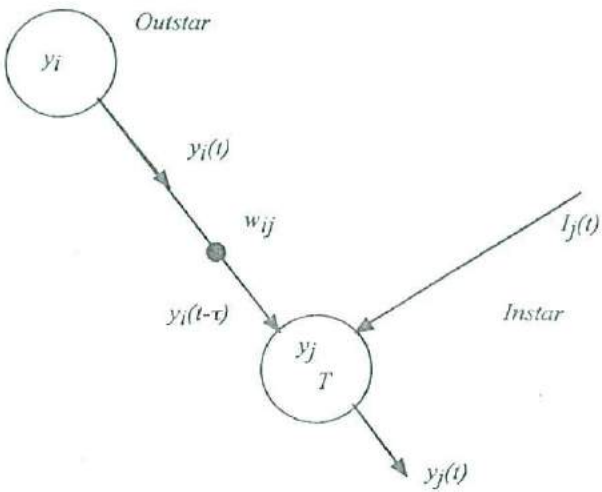


Figure 9.2 An instar y_j receives a signal from the outstar y_i through a weight w_{ij}. (T is the threshold for the incoming signal to node y_j, and τ is the time required for the signal to travel from y_i to y_j.)

signals $y_i(t)$ from outstars y_i through weights w_{ij}. The activity of the instar $y_j(t)$ can be represented by a differential equation

$$\frac{dy_j(t)}{dt} = -Ay_j(t) + I_0 + B\sum_{i=1}^{n} w_{ij}y_i(t) \qquad (9.2\text{-}1)$$

where $y_i(t)$ is the activity of the ith neuron, I_0 is the external stimulus, w_{ij} is the weight between the ith and jth neurons, and A and B are constants. The first term on the right-hand side of equation (9.2-1) allows the activity of the instar to decrease exponentially with a time constant A when it is no longer stimulated by I or inputs from other neurons. The second term I_0 corresponds to an external stimulus, and the third term represents the stimuli from the n neurons in the network. We need to allow for signals received at neuron j that were actually generated in the neuron i at some previous time τ ago and transmitted to neuron j, where τ is the "average" transmission time to from neuron i to neuron j. We also need to put a threshold (T) on the intraneuron inputs so that random noise will not interfere with the network's operation. Hence, we can modify equation (9.2-1) as follows:

$$\frac{dy_j(t)}{dt} = -Ay_j(t) + I_0(t) + B\sum_{i=1}^{n} w_{ij}\left[y_i(t-\tau) - T\right]^+ \qquad (9.2\text{-}2)$$

where the superscript + means that only positive values are used.

Instar's Learning Law

The two processes involved in instar's learning law are Hebbian learning and forgetting. We need a process that will explicitly allow forgetting to occur—that is, to allow the weight to slowly decay. Also, we need to put a threshold on the incoming activity term and to account for the transmission time between neurons i and j. Hence, we can write the instar learning law, which controls the adjustment of the weights between neurons i and j in the form given in Equation (9.1-2)

$$\frac{dw_{ij}(t)}{dt} = -Fw_{ij}(t) + Gy_j(t)[y_i(t-\tau) - T]^+ \qquad (9.2\text{-}3)$$

where F is the forgetting time constant that should be much smaller (i.e., a slower decay rate) than the activity decay constant A, and G is the gain or learning constant. The value of F is never greater than about 0.01, and the superscript + means that we should use only the positive values.

Equation (9.2-3) incorporates the "simple" Hebbian learning, and hence it typically does not allow the weights to decrease except for the very slow decay process associated with the forgetting term. A more appropriate Cohen–Grossberg learning law would be one in which the first derivative of the activities with respect to time are substituted for the activities—that is,

$$\frac{dw_{ij}(t)}{dt} = -Fw_{ij}(t) + G\frac{dy_j(t)}{dt}\left[\frac{dy_i(t-\tau)}{dt} - T\right]^+ \qquad (9.2\text{-}4)$$

This is a version of the differential Hebbian learning given in equation (9.1-3).

Grossberg Learning in Outstars

The minimum number of artificial neurons that must be activated to cause recall of a complex spatial pattern is only one, the hub of the outstar. Repeatedly applying a stimulus on the hub neuron and simultaneously putting a pattern stimuli on m neurons on the rim of the outstar (see Figure 9.1*b*), or to a grid of neurons (see Figure 9.3), each connected to the hub neuron, eventually will cause the weight pattern to reflect the input pattern on the rim due to the Hebbian learning. The mathematical mode of spatial learning in outstars describes the outcome of the standard psychological experiments conducted by Pavlov.

Grossberg outstar learning is based on Hebbian learning; that is, if a stimulus arrives at a receiving neuron at the time when it is active, then the weight associated with that link will be increased. In outstar learning, the weight increase depends on the product of the input and the output signals to the grid neuron; that is, the grid neurons corresponding to bright spots in a

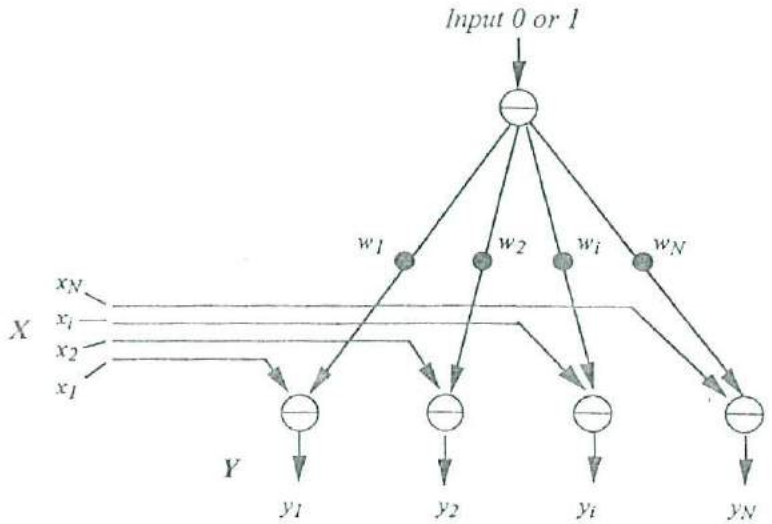


Figure 9.3 An "outstar" learning network.

pattern will have large outputs at the time the stimulus arrives. Hence, the weights will be increased. The grid neurons corresponding to medium spots will have lower outputs, and the weights will change less. After a number of cycles, bright spots will correspond to large weights while medium spots will correspond to medium weights. How about the dark spots? They are a problem. The use of neo-Hebbian learning adds a forgetting term. Hence, weights subject to neo-Hebbian learning that do not increase will slowly decrease.

Example 9.1 Grossberg Outstar Learning. This example illustrates "outstar learning." Consider the neural network shown in Figure 9.3 that has a single outstar neuron and N instar neurons in the second layer. The input to the outstar is a binary signal that switches alternatively between 0 or 1 with a period Δ. The output is a vector $\mathbf{Y}$ having N components $y_1, y_2, y_i, \ldots, y_N$. The desired output is a vector $\mathbf{X}$ having components $x_1, x_2, x_i, \ldots, x_N$. The weights $w_1, w_2, w_i, \ldots, w_N$ are to be adjusted using the Hebbian learning algorithm.

Initially, the weights are set to randomly small values. When a 1 (which is considered a "high") is applied to the outstar neuron at the same time the vector $\mathbf{X}$ (whose components may be "high," "low," or any value in between) is applied to the second layer, Hebbian learning requires that the weight on the connections between two neurons increase in proportion to the product of the magnitudes of the two weights. Hence, those weights on connections

leading to neurons in the second layer with large values of x_i increase significantly since the outstar output is unity. The weights on the connections to neurons with medium values of x_i increase somewhat less. The weights on the connections to neurons with low values of x_i increase only slightly or do not increase at all. This process occurs each time the outstar signal switches to 1. The changes in weights become smaller as the weights increase and eventually stop increasing. Now, the magnitudes of the weight vector components w_i mimic the magnitudes of the corresponding values of the input vector components x_i.

At this point, the desired output vector X can be removed. When the input is 0, all the components y_i of output vector Y are equal to 0. When the input is 1, the components y_i of the output vector Y are proportional to components w_i of the trained weight vector W; that is, the output vector Y mimics the desired output vector X even though it is no longer applied. Hence, except for a constant of proportionality, the output vector Y is the same as the desired vector X. This means that all that is necessary to produce the desired output at the second layer is to apply a 1 to the outstar.

If we return to the analogy with Pavlovian learning, the pattern X is analogous to the food, the unconditioned stimulus; the input to the outstar is analogous to ringing the bell, the conditioned stimulus; the stimulus of the grid output is analogous to dog salivating when the food is presented initially, the unconditioned response; and the output Y after the pattern X is eliminated is analogous to the dog salivating after the food is eliminated, the conditioned response. □

Driver Reinforcement Learning

Driver reinforcement learning is a variation of outstar learning that uses differential Hebbian learning in which the weight increase depends on the product of the change in the output signal of the receiving neuron and a time-weighted sum of the changes of the inputs to that neuron over a period of time. The weight used in weighing the sum of the changes is the weight of the neuron at that time.

During training, we artificially cause grid neurons to display the image we want the network to reproduce. Hence, grid neurons corresponding to bright spots on the pattern have large outputs when the outstar stimulus is received, and weights are increased with each repetition. Eventually, the outstar's stimulus alone is sufficient to cause the neurons to produce the pattern without the input. This process is the essence of driver reinforcement learning.

9.3 ASSOCIATIVE MEMORIES

An associative memory is any memory system that stores information by associating each data item with one or more other stored data items. The

characteristics of associative memories are such that they usually store information in a distributed form. They are content addressable memories; that is, data are accessed by its content, not by its address. They are robust and can usually handle garbled or incomplete data inputs and can usually operate with some failed elements. In many ways, they are very similar to human memory.

Data are stored as patterns of activity in an associative memory. As a result, associative memories are insensitive to minor differences in details. This provides the robustness which allows garbled inputs to be still understood, and minor errors or damage to the network do not cause loss of functionality.

It is useful to distinguish between heteroassociative and autoassociative memories. In a heteroassociative memory network, the input X and the output Y are different patterns; that is, the input and output are not the same. In the case of the autoassociative memory networks, the input and the output patterns are the same. At first glance, this may seem like a trivial system, but it is very useful. When the input to an autoassociative memory is somewhat garbled (e.g., it is a distorted version of X), the network will produce a "correct" version of X as the output.

While associative memories do not have to involve neural networks, there are several types of neural networks that constitute associative memories. The most common types are the crossbar associative memory and the adaptive filter (e.g., the Adaline neural network can be used as an associative memory). There are other architectures in neural networks that can be used as associative memories, but they are not common. Generally, associative networks are important because they provide robust and efficient storage of pattern data and are generally considered to be essential to any "intelligent" system.

Another classification of associative memories is that they are accretive or interpolative. This indicates how they interpret data. Suppose we have an associative memory that associates the color red with the numerical value 1, the color blue with 2, and the color green with 3; that is, if we put the color magenta into the associative network, an accretive associative network will return the value of either 1 or 2, depending upon whether the shade of magenta is closer to red or blue. If the magenta corresponded to a value of 1.6, then the accretive network would give an output of 2. On the other hand, an interpolative associative memory would return the actual numerical value (1.6 in this case).

Crossbar Structure

Crossbar networks have the structure of an early twentieth-century telephone exchange system from which they take their name. They typically have one or two layers of artificial neurons, and each neuron or layer is fully intercon-

nected with the other neurons or layers. Crossbar representations that are commonly used are matrix representation, energy surface representation, and a feedback competition representation. The matrix representation is perhaps the most common because the weights are stored as elements of a matrix. The matrix weight representation discussed here is substantially the same as those given in Figures 7.6 and 7.7. This representation is popular because matrix mathematics and operations are well understood by most researchers. They are also mathematically tractable, allowing simple explanations of characteristics. If we consider the fully connected network shown in Figure 9.4, the input is a column vector X with components x_1, x_2, x_3, and x_4, and the output is a column vector Y with components y_1, y_2, and y_3; then we can say that

$$\mathbf{Y} = \mathbf{W} \cdot \mathbf{X} \tag{9.3-1}$$

where W is the weight matrix. If we expand the terms in equation (9.3-1), it becomes

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{21} & w_{31} & w_{41} \\ w_{12} & w_{22} & w_{32} & w_{42} \\ w_{13} & w_{23} & w_{33} & w_{43} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{9.3-2}$$

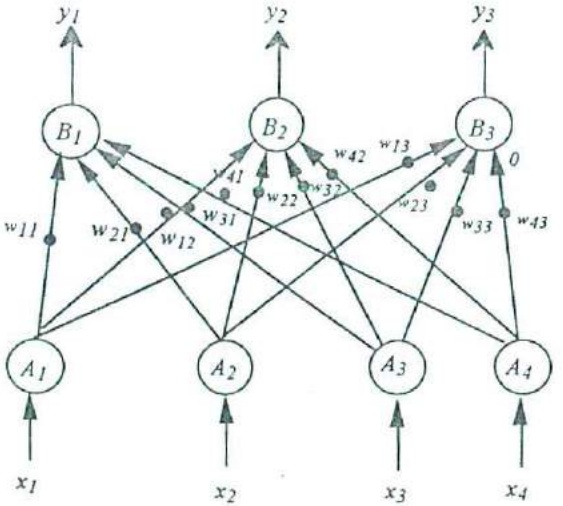


Figure 9.4 Matrix representation of a neural network.

Bidirectional Associative Memory

Mathematically, a bidirectional associative memory (BAM) developed by Kosko (1988) is also a matrix; technically it is a crossbar network with symmetric weights. Each neuron in each layer has one input from the outside and inputs from each of the neurons in the other field of the neurons.

Suppose we construct a BAM to store three pattern pairs: $[X_1, Y_1]$, $[X_2, Y_2]$, and $[X_3, Y_3]$. Since a BAM is bidirectional we can enter any X and retrieve the corresponding Y, or we can enter any Y and retrieve the corresponding X.

The process in a BAM is fundamentally different than the operation of other types of neural networks. For instance, in the backpropagation network discussed previously, the weights are trained to provide the desired input–output mapping. In the case of a BAM, the weight matrix is not trained; it is constructed using the input–output pairs. The process involves constructing a matrix for each input–output pair and then combining them into a master matrix. X_i and Y_i are treated as column vectors, and then the matrix is produced by taking the product of the X_i vector and transpose of the Y_i vector, Y_i^T. Let us consider the three pairs shown below:

$$X_1: \quad (+1 \ -1 \ -1 \ -1 \ -1 \ +1) \Leftrightarrow (-1 \ +1 \ -1) \quad :Y_1 \qquad (9.3\text{-}3)$$

$$X_2: \quad (-1 \ +1 \ -1 \ -1 \ +1 \ -1) \Leftrightarrow (+1 \ -1 \ -1) \quad :Y_2 \qquad (9.3\text{-}4)$$

$$X_3: \quad (-1 \ -1 \ +1 \ -1 \ -1 \ +1) \Leftrightarrow (-1 \ -1 \ +1) \quad :Y_3 \qquad (9.3\text{-}5)$$

Weight Matrix Representation Since X_1 has 6 elements and Y_1 has 3 elements, the matrix for each set of inputs result in a 6×3 matrix. It is important to note that each of the patterns is made up of $+1$ and -1 values, which means that the components are bipolar. If the patterns values are binary (i.e., made up of 1 and 0 values), they should be converted to bipolar form by substituting -1 for each 0 before they are used in a BAM. The correlation matrices M_i for equations (9.3-3), (9.3-4), and (9.3-5) are obtained by cross product of X_i and Y_i—that is,

$$M_i = X_i \times Y_i^T \qquad (9.3\text{-}6)$$

The three correlation matrices are

$$M_1 = X_1 \times Y_1^T = \begin{bmatrix} +1 \\ -1 \\ -1 \\ -1 \\ -1 \\ +1 \end{bmatrix} \times \begin{bmatrix} -1 & +1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & +1 & -1 \\ +1 & -1 & +1 \\ +1 & -1 & +1 \\ +1 & -1 & +1 \\ +1 & -1 & +1 \\ -1 & +1 & -1 \end{bmatrix} \qquad (9.3\text{-}7)$$

$$\mathbf{M}_2 = \mathbf{X}_2 \times \mathbf{Y}_2^T = \begin{bmatrix} -1 \\ +1 \\ -1 \\ -1 \\ +1 \\ -1 \end{bmatrix} \times \begin{bmatrix} +1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -1 & +1 & +1 \\ +1 & -1 & -1 \\ -1 & +1 & +1 \\ -1 & +1 & +1 \\ +1 & -1 & -1 \\ -1 & +1 & +1 \end{bmatrix} \tag{9.3-8}$$

$$\mathbf{M}_3 = \mathbf{X}_3 \times \mathbf{Y}_3^T = \begin{bmatrix} -1 \\ -1 \\ +1 \\ -1 \\ -1 \\ +1 \end{bmatrix} \times \begin{bmatrix} -1 & -1 & +1 \end{bmatrix} = \begin{bmatrix} +1 & +1 & -1 \\ +1 & +1 & -1 \\ -1 & -1 & +1 \\ +1 & +1 & -1 \\ +1 & +1 & -1 \\ -1 & -1 & +1 \end{bmatrix} \tag{9.3-9}$$

Note that each value in the above matrices is a product of two quantities, one component of $\mathbf{X}$ and one component of $\mathbf{Y}$. This product $x_i \cdot y_j$ is a classical indication that Hebbian learning is involved.

In order to obtain an associative weight memory (called the master weight matrix) capable of storing the three pairs in equations (9.3-3), (9.3-4), and (9.3-5), we simply add the three correlation matrix equations (9.3-7), (9.3-8), and (9.3-9). The result is

$$\mathbf{M} = \mathbf{M}_1 + \mathbf{M}_2 + \mathbf{M}_3 \tag{9.3-10}$$

$$\mathbf{M} = \begin{bmatrix} -1 & +3 & -1 \\ +3 & -1 & -1 \\ -1 & -1 & +3 \\ +1 & +1 & +1 \\ +3 & -1 & -1 \\ -3 & +1 & +1 \end{bmatrix} \tag{9.3-11}$$

Matrices can be added only if they are the same size. Hence, this means that all of the $\mathbf{X}_i$ vector patterns must have the same number of components, and all of the $\mathbf{Y}_i$ vector patterns must have the same number of components. However, the number of components in the $\mathbf{X}_i$ pattern can be different from the number of components in the $\mathbf{Y}_i$ patterns (as is the case in this example).

In order to put in any $\mathbf{X}_i$ and get back any $\mathbf{Y}_i$ (or put in any $\mathbf{Y}_i$ and get back any $\mathbf{X}_i$), we have to take the product of the input vector and the matrix. This is equivalent to taking the dot product of the vectors and the master matrix. The result is

$$\mathbf{X}_i = \mathbf{M} \cdot \mathbf{Y}_i \tag{9.3-12}$$

and

$$\mathbf{Y}_i = \mathbf{M}^T \cdot \mathbf{X}_i \tag{9.3-13}$$

where the M's are 6×3 weight matrices, $\mathbf{X}_i$ is a 6×1 column vector, and $\mathbf{Y}_i$ is a 3×1 column vector. Note that we must use the transpose of the master matrix to get $\mathbf{Y}_i$; that is,

$$\mathbf{M}^T = \begin{bmatrix} -1 & +3 & -1 & +1 & +3 & -3 \\ +3 & -1 & -1 & +1 & -1 & +1 \\ -1 & -1 & +3 & +1 & -1 & +1 \end{bmatrix} \tag{9.3-14}$$

Example 9.2 Using a Bidirectional Associative Memory. Let us use equation (9.3-12) to obtain $\mathbf{X}_2$ from $\mathbf{Y}_2$:

$$\mathbf{X}_2 = \begin{bmatrix} -1 & +3 & -1 \\ +3 & -1 & -1 \\ -1 & -1 & +3 \\ +1 & +1 & +1 \\ +3 & -1 & -1 \\ -3 & +1 & +1 \end{bmatrix} \cdot \begin{bmatrix} +1 & -1 & -1 \end{bmatrix} = \begin{bmatrix} -3 \\ +5 \\ -3 \\ -1 \\ +5 \\ -5 \end{bmatrix} \Rightarrow \begin{bmatrix} -1 \\ +1 \\ -1 \\ -1 \\ +1 \\ -1 \end{bmatrix} \tag{9.3-15}$$

which is the correct pattern. The last step in equation (9.3-15) is accomplished through the use of the threshold rule; that is, the component is -1 if the original value is < 0, and it is $+1$ if the original value is ≥ 0.

Operation of a BAM. The master matrix now has three pairs stored in it. If any of the Xs or Ys are introduced to this matrix in the proper way, the corresponding response is given immediately. The problem comes when the input is a distorted version of X (or Y) which we will call $\mathbf{X}^*$ (or $\mathbf{Y}^*$) is introduced, especially if $\mathbf{X}^*$ has some similarity to more than one of the Xs. The initial response obtained as the dot product of $\mathbf{X}^*$ and M may not be any of the Ys stored in the matrix, but may be some combination of two or more of the Bs which we will call $\mathbf{Y}'$. In turn, $\mathbf{Y}'$ is sent back through the BAM to give $\mathbf{X}'$ as the dot product of $\mathbf{Y}'$ and $\mathbf{M}^T$. $\mathbf{X}'$ moves back across the BAM to give $\mathbf{Y}''$ as the dot product of $\mathbf{X}'$ and M. $\mathbf{Y}''$ then moves back across the BAM to give $\mathbf{X}''$ as the product of $\mathbf{Y}''$ and $\mathbf{M}^T$. This process continues until an equilibrium condition is attained when successive values of $\mathbf{X}^i$ and $\mathbf{Y}^i$ do not change.

The sequence of events are as follows:

1. An X input pattern is presented to the BAM.
2. The neurons in field X generate an activity pattern that is passed to field Y through the weight matrix M
3. Field Y accepts input from field X and then generates a response back to field X through the transpose weight matrix $\mathbf{M}^T$

4. Field X accepts the return response from Y, and then it generates a response back to field Y through the weight matrix M.
5. The activity bounces back and forth until a "resonance" is achieved, which means that no further changes in the patterns occur (i.e., successive values of Xs and Ys are the same). At this point, the output Y is one of the Y values stored in the master matrix, and it is the correct response for the distorted X input.

In summary, we constructed a master matrix with three pairs of input patterns $[\mathbf{X}_1, \mathbf{Y}_1]$, $[\mathbf{X}_2, \mathbf{Y}_2]$, and $[\mathbf{X}_3, \mathbf{Y}_3]$. We transposed this matrix (if necessary, depending upon which half of the pair we used as input) and then applied it to the input patterns. The result following thresholding was the other half of the pattern pair. This apparently arbitrary methodology always generates a memory matrix from which we can recall the input patterns used to produce it. Where are the patterns actually stored in the BAM matrix? They are not stored in any individual element of the weight matrix, because if we were to change one of the three patterns and reconstruct the matrix, we would get an entirely different weight matrix with virtually every element changed. Changing an individual pattern doesn't just change one row or one column of the matrix; it changes every element. We must therefore conclude that the information is stored not in an individual elements but in the matrix as a whole, and each pattern is distributed over the entire matrix.

Adding and Deleting Pattern Pairs to the Master Matrix. We can add another pattern pair $[\mathbf{X}_4, \mathbf{Y}_4]$ to our matrix by adding its matrix $\mathbf{M}_4$ to get to the memory matrix M:

$$\text{New } \mathbf{M} = \mathbf{M}_1 + \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_4 \tag{9.3-16}$$

Alternately, we can "forget" or erase a pattern pair by subtracting the matrix for that pattern pair from the memory matrix. For instance, if we wanted to remove the pair $[\mathbf{X}_2, \mathbf{Y}_2]$ from the memory, we could do it by subtracting the matrix $\mathbf{M}_2$ from the memory matrix:

$$\text{New } \mathbf{M} = \mathbf{M} - \mathbf{M}_2 \tag{9.3-17}$$

This system has all the requisite features of a memory system. It can store into memory, it can recall from memory, it can write new information, and it can erase old information. □

Capacity and Efficiency of a Crossbar Network

The capacity of a crossbar network of size $N \times N$ neurons is theoretically limited to approximately N patterns. In reality, the actual capacity of the crossbar networks is more on the order of 10–15% of N. In the matrix example cited earlier, 288 bits were needed (18 elements of 16 bits per

element) to store three patterns of 9 (total) bits each, or 27 bits total. The storage and recall operation required several major matrix operations (multiplication, transposition, and addition). With current technology this is much less efficient than simply storing the data conventionally.

Disadvantages of Crossbars

There are many disadvantages of crossbars that need to be considered. These are as follows:

1. *The Number of Connections.* A 100-node network has 100×100, or 10,000, total connections.
2. *Binary-Only Input.* To implement an analog problem, some suitable transformation must be used to convert analog quantities to binary signals.
3. *Capacity.* The theoretical storage is low for the number of connections, and the real storage capacity is even much lower.
4. *Orthogonality.* For best results, the stored data patterns should be as orthogonal as possible to minimize the overlap.
5. *Spurious Results.* In energy surface representation, spurious minima or "energy wells" that have nothing to do with the problem are sometimes produced. These are the so-called "localized minima."

There are, however, some mitigating circumstances. Near-orthogonality is usually adequate because the capacity is so low. There are few spurious minima because low capacity implies sparse coding (lots of zeros) in the data. Finally, the efficiency could be dramatically improved when practical optical systems become a reality.

Hopfield Networks

Dr. John Hopfield is the person perhaps most responsible for the rejuvenation of the neural network field after publication of *Perceptrons*. His contributions include work conceptualizing neural networks in terms of an energy model (based on spin glass physics). He showed that an energy function exists for the network and that processing elements with bistable outputs are guaranteed to converge to a stable local energy minimum. His presentation at the National Academy of Science meeting in 1982 triggered the subsequent large-scale interest in neural networks. A crossbar associative network is called the Hopfield network in his honor (Hecht-Nielsen 1990).

A typical Hopfield network is shown in Figure 9.5. It has only one computing layer, called the Hopfield layer, and the other two layers are the input and output buffers. In contrast to the backpropagation network discussed earlier, the Hopfield network has feedback from each neuron to each

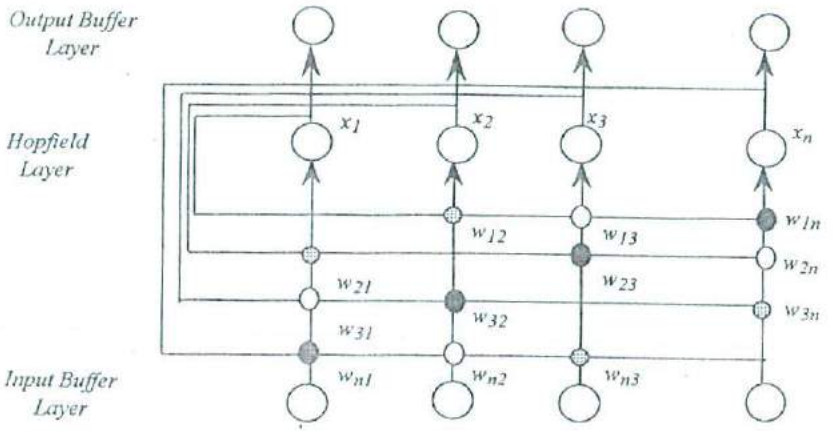


Figure 9.5 Hopfield network architecture.

of the other neurons, but not to itself. The Hopfield layer of neurons computes the weighted sum of the inputs, and it quantizes the output to 0 or 1. (This restriction was later relaxed.) The activation function used was a sigmoid with a reactive (resistor–capacitor) delay. An examination of this network shows that the weights are symmetrical; that is,

$$w_{ij} = w_{ji} \tag{9.3-18}$$

The basic Hopfield learning rule is

$$\Delta w_{ij} = (2x_i - 1)(2x_j - 1) \tag{9.3-19}$$

where x_i and x_j have values of 0 and 1, x_j is the current neuron, x_i is the input to the neuron, and w_{ji} is the connection between the jth neuron and the ith neuron. Furthermore, symmetry dictates that weight changes are symmetrical; that is,

$$\Delta w_{ji} = \Delta w_{ij} \tag{9.3-20}$$

Examination of equation (9.3-19) shows that the learning is Hebbian; that is, the change in weight is the product of two activities, and the change in weight is proportional to this product. Since x_i and x_j can only have values of 0 and 1, then the $(2x - 1)$ terms effectively convert the binary inputs and outputs into bipolar inputs and outputs; that is, the 1 remains a 1 and the 0 becomes a -1. The connections are strengthened (i.e., made more excitatory) when the output of a neuron is the same as the input (i.e., both are 0 or both are 1). Connections are weakened (i.e., made more inhibitory) when the

input differs from the output of the neuron (when one input or output is a 0 and the other is a 1). Hopfield has shown that for a large number of neurons the upper bound for the memory capacity is $0.5 \times (N/\log N)$, where N is the number of processing elements.

Energy Surface Representation

The key to the popularity of crossbar networks is that the "state" of the network can be represented by an "energy surface" in which data storage corresponds to "sculpting" energy minima in the energy surface. This view is mathematically equivalent to well-understood physical systems known as "spin glasses." Each energy well in the energy surface has a corresponding area within which all states will move to the bottom of that well or "basin of attraction."

A crossbar associative (Hopfield) memory operates by attracting the network state to an energy minimum. If we consider the energy function of the crossbar associative memory as a foam sheet with dents of different depths, the bottoms of the dents are the energy minima. They correspond to the data stored in the crossbar network. It is sometimes said that the network state is falling down into the nearest energy well, which may or may not be the global minimum.

Simulated Annealing

Simulated annealing is a process used in neural networks to reach a global minimization of an error function. It is analogous to the annealing process in metallurgy in which a metal is heated beyond a transition temperature, allowing the preexisting structure to change physically (relieving residual stresses, changing the metallographic structure, eliminating dislocations and disruptions in the crystal lattice, etc.) due to thermal agitation. Then the temperature is lowered slowly to room temperature, allowing the metal structure to slowly go through a transformation and grow structures by which it attempts to attain a global "minimum energy" configuration. In practice, the annealing process does not take place suddenly, but instead the transition starts almost simultaneously at many locations, creating many homogeneous regions that are usually separated by dislocations. Hence, there is no guarantee that the final energy level will be lower, but it usually is lower.

When a minimization process is trapped in a spurious local minimum, one of the few ways to get out of this trap is to add noise to the function until it literally escapes the minimum. This is equivalent to raising the temperature in the annealing process. When the high noise level has driven the function away from the spurious local minimum, the noise level (temperature) can be gradually lowered, allowing the function to gradually approach a global minimum. The success of this process is dependent upon the temperature used and the programmed cooling rate. If the global minimum is not reached,

the process can be repeated as many times as necessarity using other temperatures and cooling rate curves. The details of this process are discussed in detail in several references (Hecht-Nielsen, 1990; Korn, 1992; Aarts and Korst, 1989).

It is common to combine simulated annealing with other minimization or training processes. An example of this was discussed in Section 8.4 of Chapter 8, in which simulated annealing was used after the backpropagation training process had become stuck in a local minimum. After simulated annealing has moved the process away from the local minimum, backpropagation was resumed to complete the training of the neural network.

Stochastic Neural Networks

Stochastic neural networks use noise processes in their operation in an effort to reach a global minimum of an error function. The process involved in virtually all statistical networks is simulated annealing. Examples of statistical neural networks are the Boltzmann machine and the Cauchy machine. The Boltzmann machine is a discrete-time Hopfield net in which the processing element transfer function is modified to use the annealing process. The Cauchy machine is similar to the Boltzmann machine, in which different temperatures, cooling rate patterns, and procedures are used. Both allow the error to increase under some conditions in order to move out of a local minimum.

9.4 COMPETITIVE LEARNING: KOHONEN SELF-ORGANIZING SYSTEMS

"Self-organization" refers to the ability of some networks to learn without being given the corresponding output for an input pattern. Self-organizing networks modify their connection strengths based only on the characteristics of the input patterns. The Kohonen feature map, perhaps the simplest self-organization system, consists of a single layer of neurons (called the Kohonen layer) which are highly interconnected (lateral connections) within the Kohonen layer as well as to the outside world through an input buffer layer that is fully connected to the neurons in the Kohonen layer through adjustable weights.

Lateral Inhibition

Kohonen networks utilize lateral inhibition (i.e., connections between neurons within a layer) to provide (a) positive or excitatory connections to neurons in the immediate vicinity and (b) negative or inhibitory connections to neurons that are further away. The strengths of the connections vary inversely with distance between the neurons, that is, the strengths are stronger when neurons are close, but they are weaker when the neurons are

distant. The inputs from lower levels and outputs to higher levels (if any) are the same as for other networks. Generally, there is no feedback from higher to lower layers.

Lateral connections moderate competition between neurons in the Kohonen layer. When an input pattern is presented to the Kohonen layer, each neuron receives a complete copy of the input pattern modified by the connecting weights, and the varying responses establish a competition that flows over the intralayer connections. The purpose of the competition is to determine which neuron has the strongest response to the input. Each neuron in the layer tries to enhance its output and the output of its immediate neighbors and inhibit the output of the remaining neurons that are further away. Lateral connections can cause oscillations in networks, but the output eventually stabilizes with the output of the neuron, with the strongest response being declared the winner and being transmitted to the next layer if there is one. The activity of all other neurons is squashed, as the network determines for itself which neuron has the greatest response to the input pattern. The relative impact of a neuron's interlayer inhibition is also permitted to decrease with training. Initially, it starts fairly large and is slowly reduced to include only the winner and possibly its immediate neighbors. It has been shown that similar systems exist in the brain with regard to vision.

The complexity of intralayer connections makes lateral inhibition and excitation hard to implement. An alternative which is much easier to implement is to use a "max" function to determine the neuron with the greatest response to the input and then assign this neuron a +1 value to the output while assigning a zero to all other neurons in that layer. The winning neuron represents the category to which the input pattern belongs. This is not a true implementation of lateral inhibition, but it generally gives the same result as a true implementation, and it is far more efficient when implemented using serial computers. An even simpler alternative is to merely compute the dot product of each of the weight vectors with the input and then select the winner from this list.

In training, the Kohonen network classifies the input vector components into groups that are similar. This is accomplished by adjusting the Kohonen layer weights, so that similar inputs activate the same Kohonen neurons. Preprocessing the input vectors is very helpful. This involves normalizing all inputs before applying them to the network—that is, divide each component of the input vector by the vector's length:

$$x_i' = \frac{x_i}{\left[x_1^2 + x_2^2 + x_3^2 + \cdots + x_N^2\right]^{1/2}} \tag{9.4-1}$$

When building a Kohonen layer, two new things are required:

1. Weight vectors must be properly initialized. Generally, this means that the weight vectors point in random directions.

2. Weight vectors and input vectors must be normalized to a constant fixed length, usually "unity." Such normalization can cause loss of information in some situations, and there are methods of dealing with it if it occurs.

Let us assume that the weight vectors are randomly distributed and then determine how close each neuron's weight vector is to the input vector. The neurons then compete for the privilege of learning. In essence, the neuron with the largest dot product of the input vector and a weight component is declared the winner. This neuron is the only neuron that will be allowed to generate an output signal; all other neuron outputs will be set to zero. Furthermore, this neuron and its immediate neighbors are the only ones permitted to learn in this presentation. Only the winner is permitted to have an output (i.e., winner takes all).

Kohonen Learning Rule

Determining the winner is the key to training a Kohonen network. Only the winner and its immediate neighbors modify the weights on their connections. The remaining neurons experience no training. The training law used is

$$\Delta w_i = \eta\left[x_i - w_i^{\text{old}}\right] \tag{9.4-2}$$

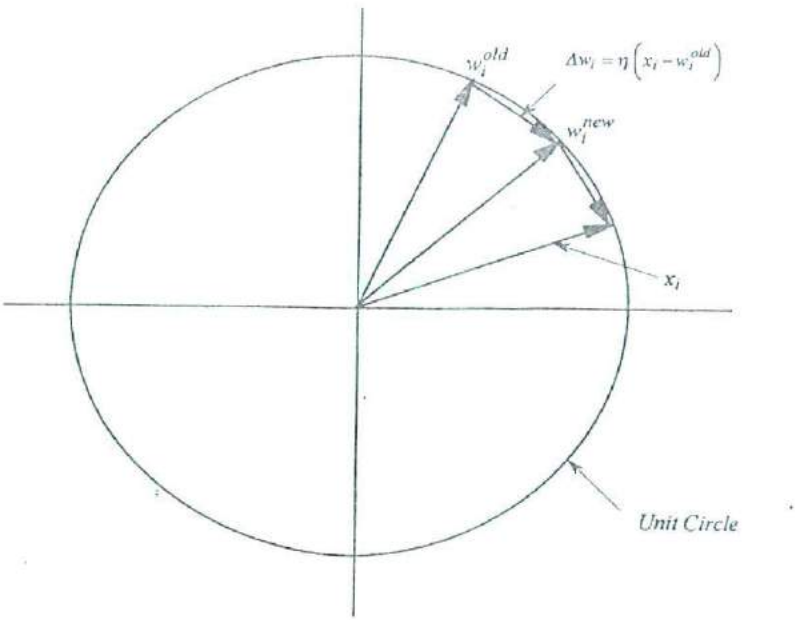


Figure 9.6 Learning in a Kohonen neural network.

where η is the learning constant whose value may vary between 0 and 1, with a typical value of about 0.2, and x_i is the input along the ith connection. It can be shown that this learning rule is a variation of the Woodrow–Hoff learning rule. This is shown graphically in Figure 9.6 for the two-dimensional case. Learning is illustrated for the case of an input x_i and a weight w_i^{old}. The difference between these two unit vectors is a vector from the tip of w_i^{old} to the tip of x_i. In Figure 9.6, this vector is broken into two parts (cords of the unit circle) so that the w_i^{new} will have a unit length. The vector from the tip of w_i^{old} to w_i^{new} represents the change in the weight vector due to learning and is equal to $\eta(x_i - w_i^{\text{old}})$.

If we consider the collection of weights for a given neuron as the components of an n-dimensional weight vector W, and we consider the corresponding inputs as the components of an m-dimensional input vector I, then Kohonen learning merely moves the weight vector so that it is more nearly aligned with the input vector. Since both input vectors and weight vectors are generally normalized to a unit magnitude, each vector points to a position on the unit circle. The winning neuron is the one with the weight vector closest to the input vector. Each training pass nudges the weight vector closer to the input vector. The winner's neighbors also adjust weights using the same learning equation, and their weight vectors move closer to the input vector. Training a Kohonen layer begins with a fairly large neighborhood size that is slowly decreased as training proceeds. The learning constant also starts with a large value and decreases as training progresses.

Let us consider the unsupervised training process for three input vectors, each with eight components and hence eight weights (a small training set solely for illustrative purposes) at three different stages of training: initial random weight distribution, partially trained weights, and fully trained weights. These three conditions are shown in two dimensions in Figure 9.7. Initially, the weight vectors are randomly scattered around the unit circle. As

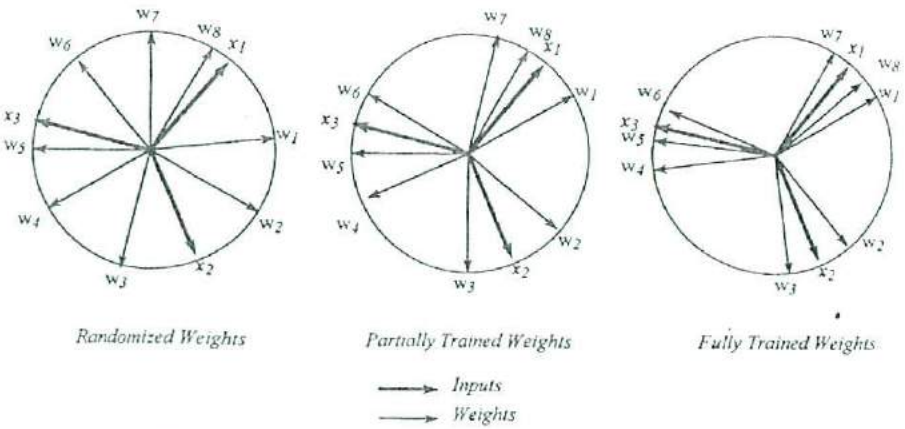


Figure 9.7 Training of weights in a Kohonen neural network.

training proceeds, the weight vectors move toward the nearest input. When the system is fully trained, the weights cluster around the three inputs so that the centroids of the three local weight clusters are on the three input vectors.

Training of a Kohonen Neural Network.

Let us consider a Kohonen neural network [sometimes called a self-organizing map (SOM)] with an input buffer layer (typically a linear array) and a Kohonen layer (typically a rectangular array or grid) that are fully connected. An input vector is applied to the buffer layer, and its component vectors are transmitted to each neuron in the Kohonen layer through randomized connecting weights. The neuron in the Kohonen layer with the strongest response (let's call it neuron q) is declared the winner and its value is set equal to 1. Then the weights connecting all component vectors from the buffer layer to the winning neuron undergo training in accordance with the process shown graphically in Figure 9.6. Neurons immediately adjacent to the winner are also allowed to undergo training. Then a second input vector is applied to the buffer layer, another neuron in the Kohonen layer is declared the winner, its value is set equal to unity, and it and its neighbors are allowed to undergo training. This process continues until all the inputs in the epoch of data have been applied to the buffer input layer. In the training process, the weights tend to cluster around the input vectors as indicated in Figure 9.7. Training stops when a criterion relating the nearness of the weights in the clusters to the relevant input vector is satisfied. Kohonen neural networks train relatively rapidly compared to backpropagation neural networks. Often a single cycle through an epoch of data, especially if the data set is large, constitutes adequate training.

It is important to note that just because the input vector selected neuron q as the most active in the first cycle of training does not mean that it will select neuron q in the second or subsequent cycles of training, because the weights on connections to neuron q change during training and perhaps training caused by adjacent neurons being winners in the first cycle. Furthermore, it is common for a particular Kohonen layer neuron to be the winner for many inputs. During the training process, input vectors that have similar characteristics move into a cluster of neurons in a particular area of the Kohonen layer, often to a single neuron. Other input vectors that have similar characteristics, which are different than those of the first cluster, move toward another area of the Kohonen layer. There will be as many clusters as there are types of inputs if an appropriate-sized rectangular array size is chosen for the Kohonen layer. More clusters require larger rectangular arrays of neurons and, the larger the number of neurons in the Kohonen layer, the longer the training process.

A Kohonen network models the probability distribution function of the input vectors used during training. Many weight vectors cluster in portions of the hypersphere that have relatively many inputs, and few weight vectors

cluster in portions of the hypersphere that have relatively few inputs. Kohonen networks perform this statistical modeling, even in cases where no closed-form analytical expression can describe the distribution. The Kohonen network can achieve this modeling spontaneously, with no outside tutor.

Kohonen networks work best when the networks are very large. The smaller the network, the less accurate the statistical model will be. Kohonen neural networks are very fast, even while training. Activation of the network is a single-pass, feedforward flow. Thus, Kohonen networks have the potential for real-time application learning. Kohonen neural networks can literally learn continuously. Hence, if the statistical distribution of the input data changes over time, it can automatically adapt to those changes and continually model the current distribution of the input pattern. The statistical modeling capabilities of the Kohonen network are unmatched by any other neural network.

The learning rate coefficient η is always less than 1; it usually starts at about 0.7 and is gradually reduced during training. If only one input vector were to be associated with each Kohonen neuron, the Kohonen layer could be trained in one calculation per weight. The weights of a winning neuron would be adjusted to the components of the training vector (with $\eta < 1$). Usually, a training set has many input vectors that are similar, and the network should be trained to activate the same Kohonen neuron for each of them.

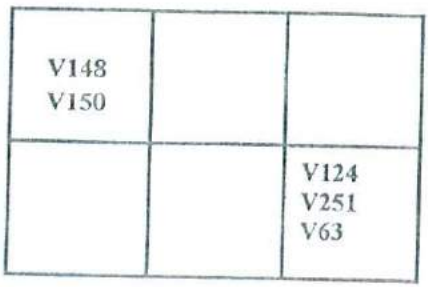
The weight vector must be set before training begins. It is common practice to randomize these weights to small values. For the Kohonen network training, the randomized weights must be normalized. After training, the weights must end up equal to the normalized input vectors. Prenormalization to unit vectors will start weight vectors closer to their final states, thereby shortening the training process.

The most desirable arrangement is to distribute the weight vectors according to the density of the input vectors that must be separated. This places more weight vectors in the vicinity of the input vectors. Although this is impractical to implement directly, there are several techniques that approximate this ideal arrangement (Masters, 1993).

Example 9.3 Valve Status Classification Using Kohonen Neural Networks. Uhrig et al. (1994) have reported a method of utilizing an accelerometer spectrum to determine the status of check valves under full flow conditions. The same data have also been applied to a Kohonen SOM to illustrate its ability to classify check valves by type and condition. The procedure involved acquiring an analog time record from an accelerator mounted on a check valve, digitizing the time record, performing a fast Fourier transform of the data to produce many spectra, and then introducing these spectra as input vectors to a Kohonen SOM. Typically, four seconds of data, sampled at 25,000 samples per second and filtered through a band pass filter with 50- and 10,000-Hz cutoff frequencies, were fast Fourier transformed to produce

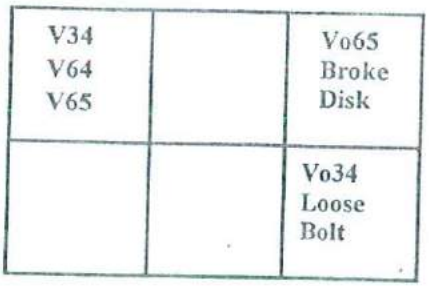
390 spectra, each with 128 values. Hence the input buffer layer had 128 neurons. The size of the Kohonen rectangular array was varied, depending upon the number, type, and condition of the valves that were being investigated. In several cases, spectral measurements were taken on the same valve when it was faulty or broken and again after it had been repaired. The results of several runs are shown in Figure 9.8 through 9.10.

Figure 9.8 shows a plot of a Kohonen SOM for five 18-inch-diameter valves, three of them (V63, V124, and V251) being identical single-disk swing check valves and two of them (V148 and V150) being identical "duo-check" valves that have two moving vanes, each covering half of the valve opening. Clearly, the spectra for the swing check valves clustered in the lower



V148 and V150 are Identical 18-Inch "Duo-Check" Valves
V63, V124, and V251 are Identical 18-Inch Swing Check Valves

Figure 9.8 SOM plot for 18-inch check values.



V34, V64, and V65 are Identical 14-Inch Swing Check Valves
Vo34 is V34 Operating with a Loose Bolt in the Disk Structure
Vo65 is V65 Operating with a Broken Disk

Figure 9.9 SOM plot for 14-inch swing check values.

3-Inch 30 % SP Degrad.		3-Inch 1st Good Valve	3-Inch 2nd Good Valve	3-Inch 30 % HP Degrad.	
					3-Inch Stuck Disk
		Both 4-Inch Good Valves			4-Inch 30 % SP Degrad.
4-Inch 30 % HP Degrad.					4-Inch Stuck Disk

Figure 9.10 SOM plot for similar 3-inch check valves operating under normal and degraded conditions.

right-hand neuron, whereas the spectra for the duo-check valves clustered in the upper left-hand neuron. Identical results were obtained for spectra from accelerometer measurements on the upstream side near the hinge pin and on the downstream side near the backstop.

Figure 9.9 shows a plot of a Kohonen SOM for three identical 14-inch swing check valves, two of them having measurements taken both in a degraded or broken condition and after they were repaired. All the spectra for the good valves clustered in the upper left-hand neuron, whereas the spectra for the degraded and broken valves clustered in the two right-hand neurons. Again, identical results were obtained from accelerometer measurements taken on the upstream and downstream sides of the valve.

Figure 9.10 shows a plot of a Kohonen SOM for similar 3-inch and 4-inch swing check valves that were tested in a test flow loop facility for both normal and degraded conditions. Both valves were tested under normal conditions twice and deliberately subjected to 30% degradation of the hinge pin (a less serious problem), 30% degradation of the stud pin (a more serious problem), and a stuck disk condition (a very serious problem). Spectra for the 3-inch valve tended to cluster in the top rows of neurons, whereas spectra for the

4-inch valve tended to cluster in the bottom rows of neurons. Spectra for normal conditions (with one exception) clustered in the third column of neurons, whereas spectra for degraded conditions (with one exception) clustered in the leftmost or rightmost columns of neurons. In all cases, the spectra for the hinge pin (HP) degraded condition (a lesser problem) clustered closer to the neurons for normal (good) conditions than the spectra for the stud pin (SP) degraded conditions and the stuck disk conditions (both more serious problems).

The choice in the size and arrangement of the Kohonen layers were arrived at by trial and error, although similar results usually were found over a range of sizes. One of the interesting aspects of this work is that there was no way of controlling where the clustering occurred within the Kohonen network. This is to be expected since self-organizing is involved here, which means that the results are dependent only on the data that is introduced into the Kohonen SOM. □

Learning Vector Quantization

A variation of the learning scheme and the addition of an output layer of neurons can make the Kohonen network into a classification network called learning vector quantizer (LVQ). The modification involves changing the training scheme from an unsupervised system to a supervised procedure. This requires a collection of training examples, each assigned to one of a set of known categories or classes. The number of neurons in the output layer is equal to the number of classes in the training data set. The Kohonen layer is trained first using a modified training procedure. LVQ training proceeds in a manner similar to that of Kohonen feature map training. An input pattern is presented to the network, and the winning node is determined by selecting the neuron with weight vector closest to the input vector. This neuron responds with its assigned category and is allowed to update its weights. The Kohonen training law is modified as shown here:

$$\Delta w_i = \eta\left(x_i - w_i^{\text{old}}\right) \qquad \text{(if answer is correct)} \qquad (9.4\text{-}3)$$

$$\Delta w_i = -\eta\left(x_i - w_i^{\text{old}}\right) \qquad \text{(if answer is not correct)} \qquad (9.4\text{-}4)$$

That is, if the winning weight vector (the one closest to the input vector) is the correct category for the input pattern, the weight vector is nudged closer to that input pattern. If, however, the winning vector is the wrong one, the weight change repels the weight vector from the input pattern vector. This should allow another weight vector to win the next time that input pattern, or one similar to it, is presented to the network. LVQ systems, including some more elaborate variations on the basic idea presented here, can achieve performance that is nearly as good as an optimal Bayesian decision system. The system is mathematically simple to implement and does not require

knowledge of the probabilities involved (Haykin, 1994). After the training is complete, the activated neurons in the Kohonen layer for input vectors associated with each class are connected directly to the output neuron for that class. Kohonen layer neurons not activated by any input vectors are not connected to the output layer.

9.5 COUNTERPROPAGATION NETWORKS

The counterpropagation network was developed by Robert Hecht-Nielsen in 1987 (Hecht-Nielsen, 1990) as an alternative to the back propagation network. It can reduce training time by a factor of 100, but it is not as general in its application. The counterpropagation network is a combination of two networks, a self-organizing Kohonen network and a Grossberg outstar network. This combination yields properties not available in either alone. In many respects this network can function as a "look-up" table that is capable of generalization. It has a supervised learning process, because the training associates input vectors with the corresponding output vectors (which may be binary or continuous). Once the network is trained, applying an input produces the desired output, even with partially incomplete input. It is useful for pattern recognition, pattern completion, and signal enhancement. The counterpropagation network combines the categorization capability of the Kohonen self-organizing network with the conditioning capabilities of the outstar network.

Robert Hecht-Nielsen, the inventor of counterpropagation, realized its limitations, indicating that counterpropagation is obviously inferior to backpropagation for most applications. Its advantages are its simplicity, the fact that it forms a good statistical model of the input vector environment, its ability to train rapidly, and its ability to save large amounts of computing time. It can be useful for rapid prototyping of systems where great accuracy is not required or a quick approximation is adequate. Furthermore, the ability to generate a function and its inverse is often useful.

Unidirectional Counterpropagation Network

Figure 9.11 shows the connection scheme of a unidirectional counterpropagation network. For clarity, only a few of the input neurons' connections to the middle layer are shown as well as only a few of the middle layers connections to the output layer. At first glance, this appears to be very similar to a fully connected backpropagation network, with connections between the input and output layers that bypass the Kohonen middle layer, but it is very different.

Consider a mapping of pattern A of size n elements to pattern B of size m elements. The objective is to introduce the A pattern to the network and get back to corresponding B pattern. The input layer receives both the A and B patterns, and thus it must be of size $m + n$. The output layer must be able to

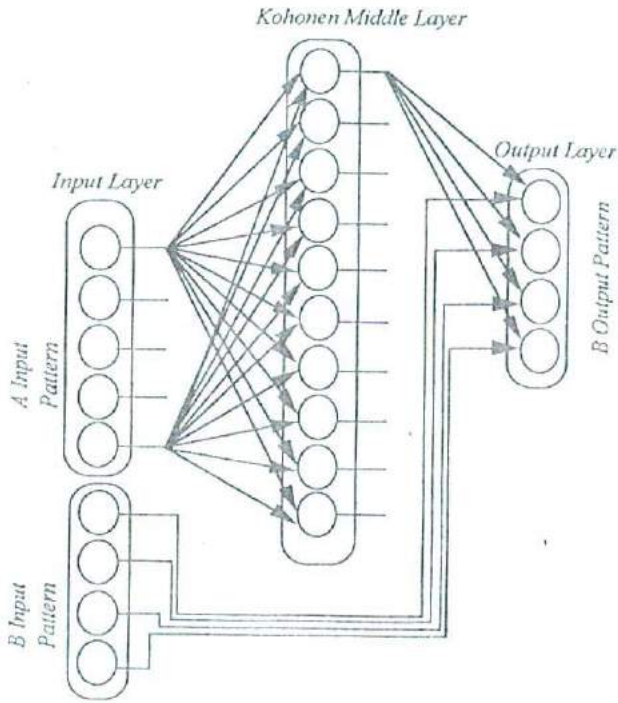


Figure 9.11 Unidirectional counterpropagation neural network.

reproduce only the B patterns so it must be a size m. There is also a direct connection between the B input and the output neurons. These connections are shown in Figure 9.11. The size and geometry of the middle layer also has to be determined.

The input layer is split into two subsections, one which receives the incoming A pattern and the other that receives the incoming B pattern. The middle layer is a competitive layer in which only a single neuron generates an output signal for each input. This output is normally set to +1 as it is with the Kohonen network. As a result, each neuron in the output layer merely receives a single signal, $a + 1$ representing the input patterns category, from the middle layer and an output from the B section of the input layer. The connections between the middle layer and the output layer obey the neo-Hebbian (outstar) learning law as demonstrated in Example 9.1, thereby producing the B output pattern when the A pattern is applied to the input layer.

The output layer's main function is to associate the correct output pattern for each category generated by the middle Kohonen layer. Because the outstar uses a supervised learning procedure corresponding to classical

conditioning, the direct connections from the input layer's B subsection to the output layer are used to provide each neuron with an "external" input or unconditioned stimulus that defines the correct output for each of middle layer's categories. The single +1 signal that arrives from the middle (Kohonen) layer neurons acts as the condition stimulus during the training of the output layer. Hence, the weight on the connection between the Kohonen layer and an output neuron is trained using neo-Hebbian learning.

The operation of a trained unidirectional counterpropagation network can be summarized as follows: An input pattern is presented to the A subsection of the input layer and is categorized by the middle (Kohonen) layer. The output layer treats the category generated by the middle layer as an outstar stimulus, because the output layer itself corresponds to the grid of an outstar network. After training is completed, an input of a particular category presented to the A input section of the network causes the output layer to generate the correct output pattern for that category without any input from the B input section.

Although the operation is simple, training the counterpropagation network is not simple, because this network involves two very different learning methods, Kohonen and outstar. Kohonen uses unsupervised training, whereas outstar requires supervised training. Training such a hybrid network normally involves a two-step procedure. In the counterpropagation network the middle Kohonen layer and the output outstar layer are separately trained. First the Kohonen layer is trained on input patterns and develops a valid feature map for the input data. Generally, the Kohonen layer is trained until it adequately recognizes the input patterns and categorizes them into the correct number of categories. During this training period the output layers and learning constants are set to very low values (or even zero) because the output of the network does not matter at this time. Once the middle layer is adequately trained, the weights between the input and middle layer are frozen. The learning constants for the middle layer are set to zero to ensure that no further changes occur, because the middle layer has learned the correct category for each input. Now it is up to the outstar to reproduce the correct output for each category. The learning constants for the outstar layer are increased so that learning occurs and continues until the output layer is appropriately trained.

Bidirectional Counterpropagational Network

The unidirectional counterpropagation network really has little advantage over other networks and systems that perform a mapping function between the input pattern and output pattern. The only advantages over backpropagation is that the middle layer does give a probability distribution mapping of the input data and the training may be faster.

The bidirectional counterpropagation network is shown in Figure 9.12 with only a few of the connections. Both the A and B input layer subsections

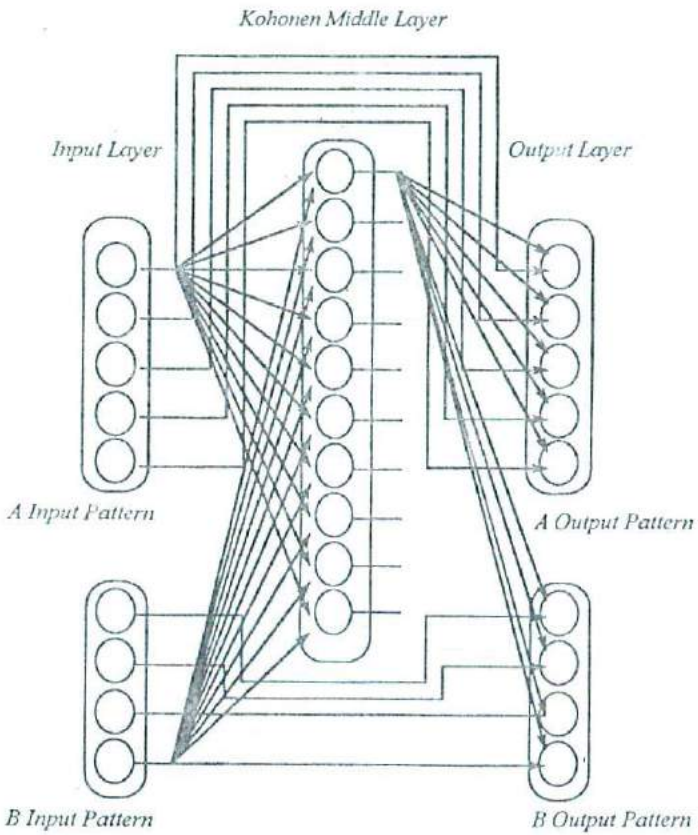


Figure 9.12 Bidirectional counterpropagation neural network.

connect to the Kohonen middle layer, and each also has a one-to-one connection to the corresponding subsection of the output layers. This effectively constitutes two counterpropagation networks, one to map A patterns to B patterns and one to map B patterns to A patterns, operating with a single Kohonen layer. A bidirectional network can accept either kind of pattern as its input and respond with a corresponding pattern with the opposite type. An A input yields a B pattern output, and a B input pattern yields an A output pattern.

In bidirectional counterpropagation networks, the input and output layers are the same size, both have A and B input sections with full connections to the middle Kohonen layer, and the middle layer is fully connected to both sections of the output layer. Each input neuron in each section has direct connections to the corresponding output neuron. The middle layer receives

input from all elements of the input layer and transmits its output to the entire output layer. Training is the same two-step process as for the unidirectional network. First the middle layer is trained using Kohonen learning to categorize the inputs correctly. Then the output layer is trained to produce the correct output for each category of input using outstar learning. The major difference is that bidirectional network can only learn one-to-one mappings. If several A patterns generate the same B pattern, then when that B pattern becomes an input, it cannot determine which A pattern to produce. Hence, one-to-many or many-to-one mappings are not possible with a bidirectional counterpropagation neural network.

Characteristics of Counterpropagation Neural Networks

Counterpropagation networks have the same disadvantages as do both the Kohonen and the outstar networks. The problem encountered most frequency is getting a variety of winners in the Kohonen layer so that the input patterns are categorized correctly. It is not unusual to find that the Kohonen layer has only a few distinct clusters during the early part of the training section, particularly if the weight vectors are randomly distributed through n-dimensional space.

Counterpropagation networks tend to be larger than corresponding backpropagation networks. If a certain number of mapping categories are to be learned, the middle layer must have at least that number of neurons. Training is usually faster if the number of neurons in the middle layer is substantially larger than the number of mappings. However, the counterpropagation network can do inverse mappings. It can provide "ungarbled" versions of A and B when supplied with garbled versions. Very few networks have a bidirectional ability; most require two networks to achieve the same result. Furthermore, the self-organization of the features of the Kohonen layer is lost, and the outputs must be supplied for supervised training.

9.6 PROBABILISTIC NEURAL NETWORKS

The probabilistic neural network (PNN) developed by Donald Specht provides a general technique for solving pattern classification problems. In mathematical terms, an input vector (called a feature vector) is used to determine a category (e.g., the spectral energy values from a sensor system can be represented as a feature vector), and the network classifiers are trained by being shown data of known classifications. The PNN uses the training data to develop distribution functions that are in turn used to estimate the likelihood of a feature vector being within several given categories. Ideally, this can be combined with *a priori* probability (relative frequency) of each category to determine the most likely category for a given feature vector.

Bayesian Probability

The PNN is a neural network implementation of Bayesian classifiers. Therefore, let us look at how Bayesian probability works. Bayes inversion formula gives

$$P(X|Y) = \frac{P(Y|X) \cdot P(X)}{P(Y)} \tag{9.6-1}$$

This equation indicates that for an event X with a certain known probability $P(X)$, the probability of event X given event Y has occurred $[P(X|Y)]$ can be computed from the probability of Y occurring given that X has occurred $[P(Y|X)]$ and the overall probability that Y will occur at all $[P(Y)]$. The relationship $P(Y|X)$ is called the *a posteriori* (the posterior) probability indicating that the probability is known only before after the event X itself has occurred.

The Bayesian formula also provides a method for categorizing patterns. In this formulation, Y is interpreted as a possible category into which a pattern might be placed and X is interpreted as the pattern itself. The decision function can be associated with each possible category (all values of Y). Bayesian decision theory tries to place a pattern in the category that has the greatest value of its decision function. However, in real-world problems, we rarely have known probabilities and must estimate or approximate such Bayesian probabilities. A probabilistic neural network has this capability. Bayesian classifiers require probability density functions that can be constructed using Parzen estimators which are used to obtain the probability density function over the feature space for each category. This allows the computation of the chance a given vector lies within a given category. Then, combining this information with the relative frequency of each category, the PNN selects the most likely category for a given feature vector. The PNN is a simple network that categorizes by estimating the probability distribution function. Like the Kohonen feature map, input data to the PNN is often normalized to a standard value, usually one.

Structure of Probabilistic Neural Networks

The probabilistic neural network consists of four layers as shown in Figure 9.13. The first layer is the input layer, which is a "fanout" or buffer layer. The second or pattern layer is fully connected to the input layer, with one neuron for each pattern in the training set. Each of the neurons in the pattern layer performs a weighted sum of its incoming signals from the input layer and then applies a nonlinear activation function to give that neuron's output.

The third layer is the summation layer to which each pattern layer neuron transmits its output to a single summation layer neuron. The weights on the

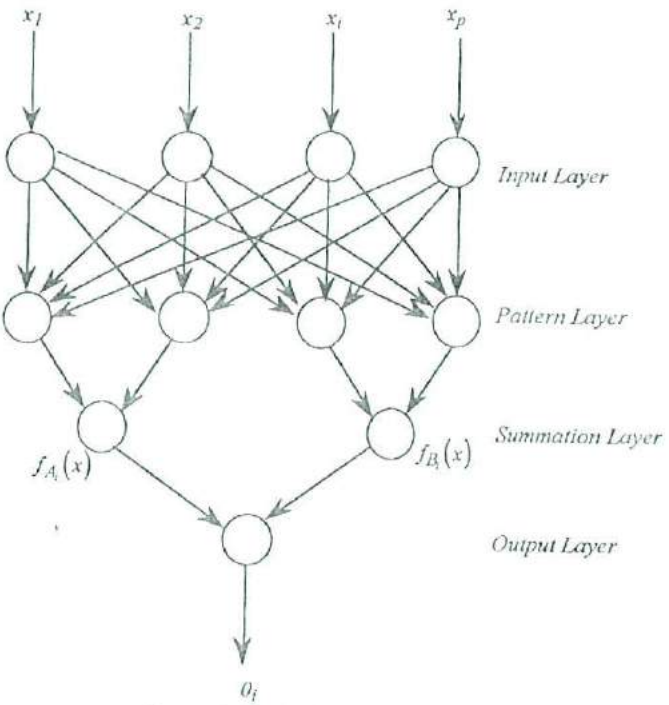


Figure 9.13 Architecture of a PNN.

connections to the summation layer are fixed at 1.0 so that the summation layer merely adds the outputs from the pattern layer neurons, which generates the networks category choice. There is one summation layer neuron per category.

The nonlinear activation function used by pattern layer neutrons is not a sigmoidal function but instead is an exponential function as shown in Figure 9.14. This activation function is given by

$$\Phi(I_i) = \exp\left[(I_i - 1)/\sigma^2\right] \qquad (9.6\text{-}2)$$

where I is the weighted input to the neuron and the σ is the smoothing parameter that determines how smooth the surface separating categories will be. A reasonable range of values for σ is 0.1 to 10. The reason for the exponential activation function is that it is a simplification of the Parzan estimator of a Bayesian surface. Using a Bayesian estimating function in the pattern layer neurons allows the PNN to approximate Bayesian probabilities in categorizing patterns.

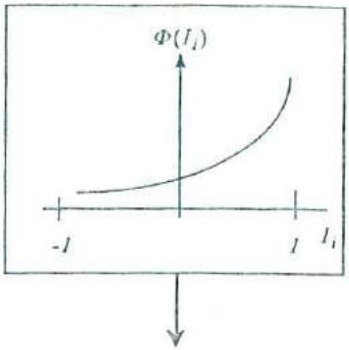


Figure 9.14 Exponential activation function for pattern layer neurons in a probabilistic neural network.

$$\Phi(I_i) = \exp\left[\frac{I_i - 1}{\sigma^2}\right]$$

The pattern layer has one neuron for each pattern in the training set. if there are 20 patterns in the training set, 12 in category A, and 8 in category B, then there are 20 neurons in the pattern layer. Each of these neurons has a set of weighted connections between it and the input layer. Each pattern layer neuron is assigned to one of the 20 training patterns, which connects to the summation layer neuron that represents its patterns category. Since in this case the summation layer has two neurons, the category A neuron receives inputs only from the 12-pattern layer neurons that represent category A, and the category B neuron receives inputs only from the eight-pattern layer neurons that represent categories B patterns. The weights on the connections from the pattern layer to the summation layer are fixed at unity.

Each neuron in the output layer received only two inputs, one from each of two summation units. One weight is fixed with a strength of unity; the other weight has a variable strength equal to

$$w' = -[h_B/h_A][l_B/l_A][n_A/n_B] \tag{9.6-3}$$

where h refers to *a priori* probability of patterns being in category A or B, l is the loss associated with identifying a pattern as being in one category when it is in reality in the other category, and n is the number of A or B patterns in the training set. The values for h_A, h_B, n_A, and n_B are determined by the data pattern themselves, but the losses must be based on knowledge of the application. In many real-world cases there is no difference in loss if the categorization is wrong in one direction or the other. If so and if the training samples are present at approximately the ratio of their overall likelihood of occurrence, this weight reduces to unity.

The PNN is trained by setting the weights of one neuron in the pattern layer to the magnitude of each training pattern's elements. That neuron is then connected to the summation unit corresponding to that pattern's category. With a single pass through the training set the network is trained.

The Smoothing Parameter

A smoothing parameter which affects the generality of decision boundaries can be modified without retraining. The PNN usually needs a reasonable number of training samples for good generalization, but it can give good results with a small number of training samples. Since each training sample is represented by a neuron in the pattern layer, this serial implementation of a PNN will typically take longer in the recall mode than a backpropagation model. Inputs need not be normalized, which in some cases may distort the inputs space in an undesirable way. However, some implementations of PNN do normalize inputs for convenience.

The smoothing parameter σ varies between zero and infinity, but neither limit provides an optimal separation. A degree of averaging of nearest neighbors provides better generalization where the degree of averaging is dictated by the density of the training samples. Figure 9.15 shows the smoothing parameter σ as it varies between 0.1 and 1.0. For the lowest value the estimated probability density function has five distinct neurons, whereas for the largest variable there is a very severe flattening of the probability density function between -3 and $+3$.

Advantages and Disadvantages of the PNN

The advantage of probabilistic neural networks is that the shape of the decision surface can be made as complex as necessary using the smoothing parameter. The decision surface can approach Bayes optimal solutions, and the neural network tolerates erroneous samples and works reasonably well with sparse data. For time-varying statistics, old patterns can be overwritten with new patterns.

The PNN operates in parallel without feedback, and training is almost instantaneous. As soon as one pattern per category has been observed, the network can begin to generalize. As new patterns are included, the decision boundary becomes more complex and better defined, and the entire training set must be stored. Testing (recalling) requires that the entire data set be used. The amount of computation required for PNN to classify any unknown pattern is proportional to the size of the training set. Unfortunately, the PNN is not as general as other neural network algorithms.

One of the serious drawbacks of the PNN is that it cannot deal with extremely large training sets. Since there must be one neuron in the pattern layer for each example in the training set, the network memory requirements can increase very rapidly with the size of training sets. In effect, the entire

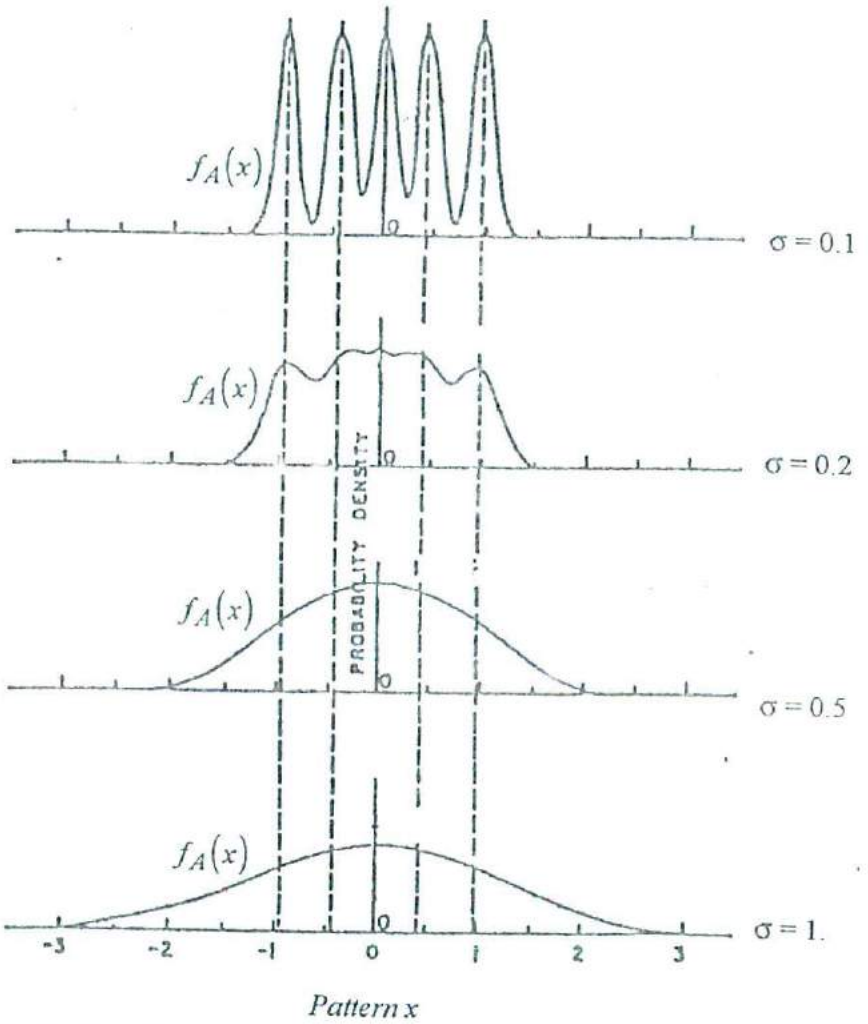


Figure 9.15 Influence of smoothing parameter σ on probability density function $f_A(x)$, the output of the summation neurons in the PNN.

training set is stored continually and retained during the classification of all feature patterns. On the other hand, the PNN signal training technique provides an extremely fast training time, particularly in comparison with an iterative network such as the backpropagation network. Furthermore, the network can deal with problems that have only a few samples of some of the categories.

9.7 RADIAL BASIS FUNCTION NETWORK

The radial basis function network (RBFN) (NeuralWare, 1993; Hush and Horne, 1993; Moody and Darken, 1989; Wasserman, 1993) always consists of three layers: the input layer, the pattern (or hidden) layer, and the output layer (i.e., the topology of the RBFN is thus identical to the backpropagation neural network). It is a fully connected and feedforward network with all connections between its processing units provided with weights. The individual pattern units compute their activation using a radial basis function; typically the Gaussian kernel function as shown in Figure 9.16 is used where σ is the width of the radial function. The activations of pattern units essentially characterize the distances of centers of radial basis functions of the pattern units from a given input vector. The radial basis functions thus produce localized, bounded, and radially symmetric activations—that is, activations rapidly decreasing with the distance from the function's centers (in contrast, the backpropagation network sigmoidal activation functions produce global and unbounded activations). Use of the radial basis activation functions requires a careful choice of the number of the pattern units to be used for a specific application, especially when a good generalization is needed; the areas of significant activation have to cover all the input space while overlapping in just the right way. For function approximation applications, this means that the samples included in the training set have to evenly represent all possible input vectors. The output units of the RBFN simply sum the weighted activations of individual pattern units without using any activation function. To speed up the training, the pattern layer neurons are augmented with bias units which have their activation values fixed to one.

The training of the RBFN differs substantially from the training used for the backpropagation network. It consists of two separate stages. During the

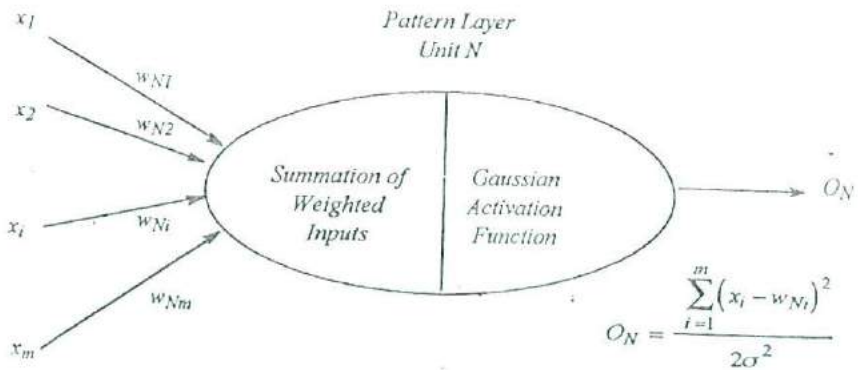


Figure 9.16 Pattern unit of the radial basis function network.

first stage, parameters of the radial basis functions (i.e., their centers and widths) for individual pattern units are set using the unsupervised training. The centers of radial basis functions of the individual pattern units define input vectors causing maximal activation of these units. Location of these centers is the first step of the training and is conducted with the help of some clustering algorithm (typically, the k-means algorithm is used). The clustering algorithms usually operate iteratively, and the clustering process is finished when locations of the centers for individual pattern units stabilize. The resulting values of individual elements of the center vectors are then directly used as values of the weights in connections between the input units and the corresponding pattern units. The widths of radial basis functions of the individual pattern units (denoted σ in Figure 9.16) determine the radii of the areas of the input space around the centers where activations of these units are significant. Their determination is the next step of the training and is performed using the "nearest-neighbor" heuristic.

In the second training stage, the weights in connections between the pattern units and the output units are determined using the supervised training based (as when training the backpropagation network) on minimization of a sum of squared errors of RBFN output values over the set of training input–output vector pairs. Before the training starts, these weights are randomized to small arbitrary values. At that stage, the weights in connections between the input units and the pattern units and the parameters of the radial basis functions of the pattern units are already set as determined in the first training stage and are not subject to any further changes. During this training, the RBFN is presented with individual input vectors from the set of training samples and responds with certain output vectors. These output vectors are compared with the target output vectors also given in the training set, and the individual weights are updated in a way ensuring a decrease of the difference between the actual and target output vectors (typically, the steepest descent optimization algorithm is used). The individual input–output training pairs are presented to the RBFN repeatedly until the error decreases to an acceptable level.

9.8 GENERALIZED REGRESSION NEURAL NETWORK

The generalized regression neural network (GRNN) (NeuralWare, 1993; Wasserman, 1993; (Specht, 1991; Caudill, 1993) is a special extension of the RBFN. It is a feedforward neural network based on nonlinear regression theory consisting of four layers: the input layer, the pattern layer, the summation layer, and the output layer (see Figure 9.17). It can approximate any arbitrary mapping between input and output vectors. While the neurons in the first three layers are fully connected, each output neuron is connected only to some processing units in the summation layer. The function of the input and pattern layers of the GRNN is exactly the same as it is in the

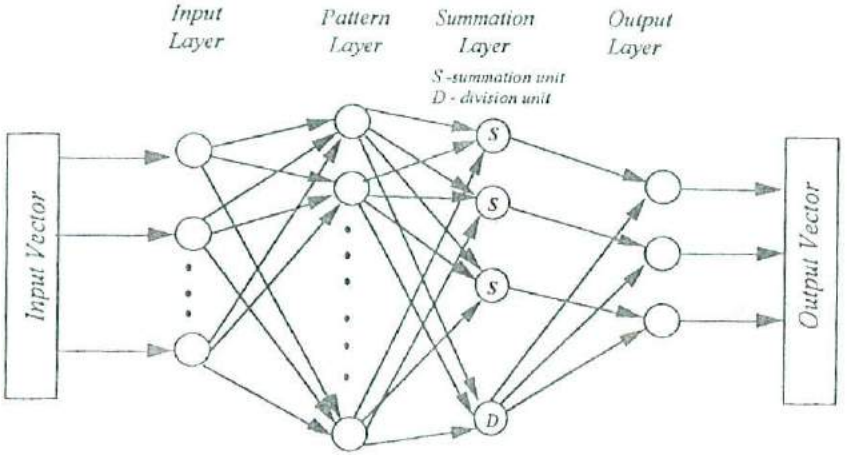


Figure 9.17 Topology of the generalized regression neural network.

RBFN. The summation layer has two different types of processing units: the summation units and a single division unit. The number of the summation units is always the same as the number of the GRNN output units; their function is essentially the same as the function of the output units in the RBFN. The division unit only sums the weighted activations of the pattern units without using any activation function. Each of the GRNN output units is connected only to its corresponding summation unit and to the division unit; there are no weights in these connections. The function of the output units consists in a simple division of the signal coming from the summation unit by the signal coming from the division unit. The summation and output layers together basically perform a normalization of the output vector, thus making the GRNN much less sensitive to the proper choice of the number of pattern units than the RBFN. The overlapping of radial basis functions of individual pattern units is not a problem for the GRNN; in fact, it turns out to be an important parameter allowing the user to influence generalization capabilities of the GRNN. In general, larger values of the width of radial basis functions of the pattern units results in a smoother interpolation of the output vectors values among the values corresponding to the centers of radial basis functions of the individual pattern units.

The training of the GRNN is quite different from the training used for the RBFN. It is completed after presentation of each input–output vector pair from the training set to the GRNN input layer only once; that is, both the centers of the radial basis functions of the pattern units and the weights in connections of the pattern units and the processing units in the summation layer are assigned simultaneously. The training of the pattern units is unsupervised, as in the case of the RBFN, but employs a special clustering

algorithm which makes it unnecessary to define the number of pattern units in advance. Instead, it is the radius of the clusters that needs to be specified before the training starts. The first input vector in the training set becomes the center of the radial basis function of the first pattern unit. The next input vector is then compared with this center of the first pattern unit and is assigned to the same pattern unit (cluster) if its distance from this center is less than the prespecified radius; otherwise it becomes the center of the radial basis function of the next pattern unit. In the same manner, all the other input vectors are compared one-by-one with all the pattern units already set, and the whole pattern layer is thus gradually built. During this training, the determined values of individual elements of the center vectors are directly assigned to the weights in connections between the input units and the corresponding pattern units. Owing to the much lower sensitivity of the GRNN to the overlapping of the radial basis functions of the pattern units, the widths of radial basis functions of the individual pattern units need not be set according to the resulting structure of the pattern layer. Instead, their setting typically becomes the subject of experimentation as their values determine generalization properties of the GRNN.

Simultaneously with building the pattern layer, the values of the weights in connections between the neurons in the pattern layer and the summation layer are also set using the supervised training. The weights in connection between each pattern unit and the individual summation units are directly assigned with values identical to the elements of the output vector corresponding in the training set to the input vector which formed the center of the radial basis function of that particular pattern unit. In case that some additional input vectors in the training set are assigned to the same pattern unit, values of the elements of their corresponding output vectors are simply added to the previous values of these weights. At the same time, the weight in the connection of each pattern unit and the division unit, which was originally set to zero, is increased by one for each input vector from the training set which is assigned to this pattern unit.

9.9 ADAPTIVE RESONANCE THEORY (ART-1) NEURAL NETWORKS

Adaptive resonance neural networks are among the more complex neural networks in use today. They are based on adaptive resonance theory (ART) developed by Carpenter and Grossberg (1986). Three general types of ART networks are used: (a) ART-1, which can handle only binary inputs and was developed in 1986; (b) ART-2, which can handle gray-scale inputs and was developed in 1987; and (c) ART-3, which can handle analog inputs better, is more complex, and was developed in 1989 to overcome some limitations of ART-2. We will discuss ART neural networks as a general system, because the principles and characteristics of adaptive resonance in all three versions

are the same. However, the implementation becomes increasingly complicated for gray-scale and analog inputs.

General Operations of an ART Neural Network

ART neural networks are two-layer neural networks fully connected with inputs going to the bottom layer, from which they are transmitted through adjustable weights to the top or storage layer. This is a bottom-up or "trial" pattern that is presented to stored patterns of the upper storage layer. The input pattern is modified during its transmission through the "bottom-up" weights to the upper layer, where it tries to stimulate a response pattern in the storage layer that contains several possible responses. Training takes place after every pass of the pattern, up or down. Since the training rule does not matter, it is common to use Hebbian learning for convenience. If this "bottom-up" pattern is selected, then "resonance" occurs, and the input is put into the matching pattern category. If it is not selected, the resulting activity in the "top-down" layer (called the "expectation" pattern or "first guess" pattern) is usually different from the bottom-up pattern because the top-down pattern is presented through the top-down weights to the bottom layer. Then the weights are adjusted, and the process is repeated. After a number of trials, the process is stopped, and a new category of pattern is created in the storage layer. This ability of ART to create new categories is its most important characteristic.

When a pattern fails to produce a match, a new pattern of nodes (from the storage layer) is now free to attempt to reach resonance with the input layer's pattern. In effect, when the trial patterns do not match, a reset subsystem signals the storage layer that a particular guess was wrong. Then that guess is "turned off," allowing another pattern from storage to take its place. This cycle repeats as many times as necessary. When resonance is reached and the guess is deemed acceptable, the search automatically terminates. This is not the only way a search can terminate; the system can terminate its search by learning the unfamiliar pattern being presented. As each trial of the search occurs, small weight changes occur in the weights of both the bottom-up and top-down pathways. These weight changes mean that the next time the trial pattern is passed up to the storage layer, a slightly different activity pattern is received, providing a mechanism for the storage layer to change its guess. If the system cannot find a match and if the input pattern persists long enough, the weights eventually are modified enough that an uncommitted node in the storage layer learns to respond to the new pattern. These changes in weights also explain why the storage layer's second or third guess may prove to be a better choice than the original one. The small weight changes ensure that the activity generated by the bottom-up pattern in the second pass is somewhat different from the activity generated in the first pass. If the input is a slightly noisy version of a stored pattern, it may require a few weight changes before the truly best guess can be matched.

Alternate View of Adaptive Resonance Operation.

The basic mode of operation in adaptive resonance is hypothesis testing. An input pattern is passed to the upper storage layer which attempts to recognize it by making a guess about the category to which the input layer belongs. It is then sent in the form of a top-down pattern to the lower layer. The result is then compared to the original pattern. If the guess is correct (or close enough), the two patterns reinforce each other and all is well. If the guess is incorrect, the upper layer tries again. Eventually, either the pattern is placed into an existing category or it is learned as the first example of a new category. Thus the upper layer forms a hypothesis of the correct category for each input pattern, which is tested by sending it back down to the lower layer. If a good match is made, the hypothesis is validated. However, a bad match results in a new hypothesis. If the pattern excited in the input layer nodes by the top down input is a close match to the pattern excited in the input layer by the external input, then the system is said to be in *adaptive resonance*, because each layer's activity mutually reinforces and strengthens the other layer's activity. It is adaptive because both sets of weights on the interconnections between the layers are continually modified to strengthen the recognition of the input pattern while the patterns resonate. Complexities must be added to carry out all the comparisons and decisions. The implementation of the acceptance/rejection process and storage of patterns are straightforward but complex and based on logical operations.

Vigilance

ART-1 also has the property of *vigilance* by which the accuracy with which the network guesses the correct match can be varied. By setting a new value for vigilance, the user can control whether the network deals with small differences or concerns itself only with global features. A low reset threshold implies high vigilance and close attention to detail. A high threshold implies low vigilance and a more global view of the pattern in the matching process. By controlling the vigilance, the user can differentiate "insignificant noise" and a "significant new pattern." Hence, the coarseness of the categories into which the system sorts patterns can be chosen. High vigilance forces the system to separate patterns into a large number of fine categories, while low vigilance causes the same set of patterns to be lumped into a small number of coarse categories.

Properties of ART-1

ART-1 possesses several of the characteristics needed in a system capable of autonomous learning. The more important characteristics are listed below:

1. It learns constantly but learns only significant information and does not have to be told what information is significant.

2. New knowledge does not destroy information already learned.
3. It rapidly recalls an input pattern it has already learned.
4. It functions as an autonomous associate memory.
5. It can (with a change in the vigilance parameter) learn more detail if that becomes necessary.
6. It recognizes its associative categories as needed.
7. Theoretically, it can even be made to have an unrestricted storage capacity by moving away from single-node patterns in the storage layer.
8. However, it can handle only binary patterns.
9. Its ability to create new categories is its most important attribute.

REFERENCES

Aarts E., and Korst J., *Simulated Annealing and Boltzmann Machines*, John Wiley & Sons, New York, 1989.

Carpenter, G. A., and Grossberg, S., Associative Learning, Adaptive Pattern Recognition, and Cooperative-Competitive Decision Making By Neural Networks, in *Optical and Hybrid Computing, SPIE Proceedings*, Vol. 634, H. Szu, ed., Bellingham, WA, 1986, pp. 218–247.

Caudill, M., GRNN and Bear It, *AI Expert*, Vol. 8, No. 5, pp. 28–33, 1993.

Haykin, S., *Neural Networks—A Comprehensive Foundation*, Macmillan, New York, 1994.

Hebb, D., *Organization of Behavior*, John Wiley & Sons, New York, NY, 1949.

Hecht-Nielsen, R., *Neurocomputing*, Addison-Wesley, Reading, MA, 1990.

Hush, D. R., and Horne B. G., Progress in Supervised Neural Networks, *IEEE Signal Processing Magazine*, Vol. 10, No. 1, pp. 8–39, 1993.

Korn, G. A., *Neural Network Experiments on Personal Computers and Workstations*, MIT Press, Cambridge, MA, 1992.

Kosko, B., Bidirectional Associate Memories, *IEEE Trans. Systems, Man and Cybernetics*, Vol. 18, (1), pp. 49–60, 1988.

Masters, T., *Practical Neural Network Recipes in C++*, Academic Press, San Diego, CA, 1993.

Moody, J., and Darken, C. J., Fast Learning in Networks of Locally Tuned Processing Units, *Neural Computation*, Vol. 1, pp. 281–294, 1989.

Neural Ware, *Neural Computing—A Technology Handbook for Professional II/Plus and NeuralWorks Explorer*, NeuralWare, Pittsburgh, PA, 1993.

Specht, D. F., A General Regression Neural Network, *IEEE Transactions on Neural Networks*, Vol. 2, pp. 568–576, 1991.

Uhrig, R. E., Tsoukalas, L. H., Ikonomopoulos, A., Essawy, M., Yancey, S., Travis, M., and Black, C., Applications of Neural Networks, EPRI Report TR-103443-P1-P2, Project 8010-12, Final Report, January 1994.

Wasserman, P. D., *Advanced Methods in Neural Computing*, Van Nostrand Reinhold, New York, 1993.

PROBLEMS

1. Construct a bidirectional associative memory (BAM) that will map the gray code (see Table 17.1) for the ten digits (1, 2, 3, 4, 5, 6, 7, 8, 9, and 0) into the corresponding 4-bit binary code. Introduce a distorted (one bit wrong) gray scale representation for 7 and see if you get the correct binary code (0111). If so, why; if not, why not?

2. A Kohonen network has inputs at 45° and 170° on the unit circle as shown in Figures 9.6 and 9.7. Randomized weights are located at 270° and 90°. Use a learning constant of 0.5. Calculate the new positions of the weight vectors after one upgrade cycle.

3. Discuss the relative benefits and modes of operation for probabilistic neural network, radial basis function network, and the generalized regression neural network. Give examples where each can be used advantageously.

4. The Kohonen network part of the counter propagation neural network uses Hebbian learning. Derive the equation for the training algorithm for this network if backpropagation is used.

10

DYNAMIC SYSTEMS AND NEURAL CONTROL

10.1 INTRODUCTION

The ability of an artificial neural network to model a system or phenomenon allows it to be used in a variety of ways. Even elementary linear neural systems with a single neuron such as the Adaline (adaptive linear neuron) network introduced by Widrow have proven to be extremely useful. Indeed, much of what is called "adaptive linear systems theory" is directly applicable to artificial neural networks. The ability of neural networks to develop nonlinear models of a system offers an additional advantage that can be useful in many cases. In this chapter, we will explore the use of both simple and complex artificial neural networks to accomplish a variety of dynamic tasks, including control of complex systems.

10.2 LINEAR SYSTEMS THEORY

Linear systems theory is a well-developed field that is extremely important to the processing of data and the application of technologies such as neural networks to practical problems. When combined with random noise theory, the resultant technology becomes a powerful tool for investigating complex systems. It is assumed that the reader is generally familiar with the concepts of both linear systems theory and random noise theory. For those who want a review, there are a number of textbooks, including one written by one of the authors (Uhrig, 1970), that can provide the necessary background. Only the concepts needed for a general understanding of the applications will be presented here.

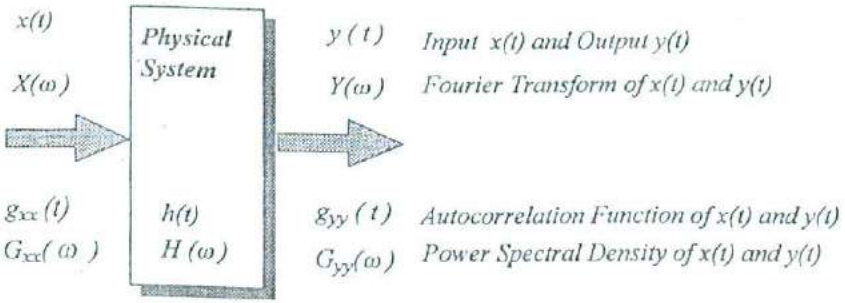


Figure 10.1 Simple physical system with input and output.

Autocorrelation and Power Spectral Density Relationships

Let us consider a physical system with an input $x(t)$, an output $y(t)$, an impulse response function $h(t)$, and a system response function $H(\omega)$, as shown in Figure 10.1. $H(\omega)$ is a complex quantity with both amplitude and phase (or real and imaginary components).[1] The system response function is the Fourier transformation of the impulse response function; that is,

$$H(\omega) = \int_{-\infty}^{\infty} h(\tau) e^{-j\omega\tau} d\tau \tag{10.2-1}$$

where τ is the variable of integration. The output $y(t)$ of the physical system in the time domain is the convolution of the input $x(t)$ and the impulse response function $h(t)$:

$$y(t) = \int_{-\infty}^{\infty} h(\lambda) x(t - \lambda)\, d\lambda \tag{10.2-2}$$

where λ is the dummy variable of integration. The corresponding relationship in the frequency domain is

$$Y(\omega) = H(\omega) X(\omega) \tag{10.2-3}$$

[1]The term *system response function* as used here is the classical meaning of the term; it is the Fourier transformation of the impulse response function of the physical system. It is sometimes erroneously called *transfer function*, which is the Laplace transformation of the impulse response function. The use of the term "transfer function" in neural networks to mean the activation function or a nonlinear filtering element on the output of a neuron is an unfortunate situation that sometimes occurs when two fields are merged.

where $Y(\omega)$ and $X(\omega)$ are Fourier transformations of the output $y(t)$ and input $x(t)$, respectively; that is,

$$X(\omega) = \int_{-\infty}^{\infty} x(\tau) e^{-j\omega\tau} d\tau \tag{10.2-4}$$

$$Y(\omega) = \int_{-\infty}^{\infty} y(\tau) e^{-j\omega\tau} d\tau \tag{10.2-5}$$

For a time stationary process (i.e., a process whose characteristics remain constant with time), the autocorrelation function $g_{xx}(\tau)$ of the fluctuating variable input $x(t)$ is defined by

$$g_{xx}(\tau) = \lim_{T\to\infty} [1/2T] \int_{-T}^{T} x(t)x(t+\tau)\, dt = E[x(t)x(t+\tau)] \tag{10.2-6}$$

and the corresponding power spectral density $G_{xx}(w)$ is given by the Fourier transformation of $g_{xx}(\tau)$ to be

$$G_{xx}(\omega) = \int_{-\infty}^{\infty} g_{xx}(\tau) e^{-j\omega\tau} d\tau \tag{10.2-7}$$

Relationships identical to equations (10.2-6) and (10.2-7) also apply to the output variable $y(t)$.

The relationship between the autocorrelation functions and the power spectral densities of the input and output variables and the physical system characteristics has been shown to be (Uhrig, 1970)

$$g_{yy}(\tau) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(\lambda)h(\xi) g_{xx}(\tau - \xi + \lambda)\, d\xi\, d\lambda \tag{10.2-8}$$

$$G_{yy}(\omega) = H^*(\omega)H(\omega)G_{xx}(\omega) = |H(\omega)|^2 G_{xx}(\omega) \tag{10.2-9}$$

where $H^*(\omega)$ is the conjugate value of $H(\omega)$ (i.e., the sign of the imaginary part of $H(\omega)$ is reversed). Note that $G_{yy}(\omega)$ and $G_{xx}(\omega)$ are real quantities, and hence only the amplitude or modulus of the system response function $|H(\omega)|$ is involved in this relationship. The phase angle of the system response function is not involved.

For the special case where the input $x(t)$ is a white noise—that is, the power spectral density $G(\omega)$ is a constant K over all frequencies, and the autocorrelation function is a Dirac delta function, $2\pi K\delta(t)$—equations (10.2-8) and (10.2-9) become

$$g_{yy}(\tau) = 2\pi K \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(\lambda)h(\xi)\, \delta(\tau - \xi + \lambda)\, d\xi\, d\lambda \tag{10.2-10}$$

$$G_{yy}(\omega) = K|H(\omega)|^2 \tag{10.2-11}$$

While equation (10.2-11) is simple and easily implemented, equation (10.2-10) is complex. However, if we have a simple first-order "lag" system with an exponential impulse response function of the form

$$h_{xx}(\tau) = Ae^{-\alpha\tau} \tag{10.2-12}$$

a situation that often occurs in practical situations, then equation (10.2-10) can be reduced to

$$g_{yy}(\tau) = \left[\pi A^2 K/\alpha\right] e^{-\alpha\tau} = K' e^{-\alpha\tau} \tag{10.2-13}$$

where K' is a constant of proportionality and α is a decay constant. Similar simplifications are possible for other impulse response functions that are more complex than equation (10.2-12).

The autocorrelation relationship of equation (10.2-8) has been developed into a more useful form (Lee, 1960) by introducing the concept of the autocorrelation function of the impulse response function $g_{hh}(\tau)$ which is defined as in equation (10.2-6) to be

$$g_{hh}(\tau) = [1/2T] \lim_{T\to\infty} \int_{-T}^{T} h(t)h(t+\tau)\,dt = E[h(t)h(t+\tau)] \tag{10.2-14}$$

Equation (10.2-8) then becomes

$$g_{yy}(\tau) = \int_{-\infty}^{\infty} g_{hh}(t) g_{xx}(\tau - t)\,dt \tag{10.2-15}$$

Cross-Correlation and Cross-Spectral Density Relationships

In a manner similar to equation (10.2-6), the cross-correlation function $g_{xy}(\tau)$ between two variables $x(t)$ and $y(t)$ is defined by

$$g_{xy}(\tau) = \frac{1}{2T} \lim_{T\to\infty} \int_{-T}^{T} x(t)y(t+\tau)\,dt = E[x(t)y(t+\tau)] \tag{10.2-16}$$

and the corresponding cross-spectral density $G_{xy}(\omega)$ is given by the Fourier transformation of $g_{xy}(\tau)$

$$G_{xy}(\omega) = \int_{-\infty}^{\infty} g_{xy}(\tau) e^{-j\omega\tau}\,d\tau \tag{10.2-17}$$

Note that the cross-spectral density $G_{xy}(\omega)$ is a complex quantity with amplitude and phase, whereas the power spectral density $G_{xx}(\omega)$ is a real quantity with magnitude only.

The time- and frequency-domain input–output relationships for the cross-correlation functions and cross-spectral densities, when applied to the physical system of Figure 10.1, are

$$g_{xy}(\tau) = \int_{-\infty}^{\infty} h(\lambda) g_{xx}(t - \lambda)\, d\lambda \tag{10.2-18}$$

$$G_{xy}(\omega) = H_{xy}(\omega) G_{xx}(\omega) \tag{10.2-19}$$

For the special case where the input variable $x(t)$ is white noise, these relationships become

$$g_{xy}(\tau) = 2\pi K h(\tau) \tag{10.2-20}$$

$$G_{xy}(\omega) = K H_{xy}(\omega) \tag{10.2-21}$$

where K is a constant of proportionality equal to the power spectral density of the "white noise."

Influence of Noise on Measurements

The relationships of equations (10.2-8), (10.2-9), (10.2-18), and (10.2-19) are valid as long as the signals are uncorrupted with noise or other extraneous signals. When noise is present, these equations can be modified to include the influence of the noise (which is dependent upon where the noise is introduced), but it is necessary to know the characteristics of the noise to obtain meaningful measurements. Sometimes, the noise is an approximation of "white" noise (i.e., the power spectral density is constant over the frequency range of interest), and this fact may allow the input–output relationships derived here, modified for the noise inputs, to be used advantageously.

For instance, let us consider the arrangement shown in Figure 10.2, which consists of a physical system with an input $x(t)$ and an instrumentation system with an input that is the sum of the output of the physical system and external or detector noise as its input. Application of equation (10.2-9) gives

$$\begin{aligned} G_{zz}(\omega) &= |H_1(\omega)|^2 \left[G_{yy}(\omega) + G_{nn}(\omega) \right] \\ &= |H_1(\omega)|^2 \left[|H(\omega)|^2 G_{xx}(\omega) + G_{nn}(\omega) \right] \end{aligned} \tag{10.2-22}$$

If the instrumentation system has a flat response beyond the range of interest and the detector noise G_{nn} is "white" (i.e., the power spectral density of the noise is constant), then equation (10.2-22) becomes

$$G_{zz}(\omega) = K_1 \left[|H(\omega)|^2 G_{xx}(\omega) + K_2 \right] \tag{10.2-23}$$

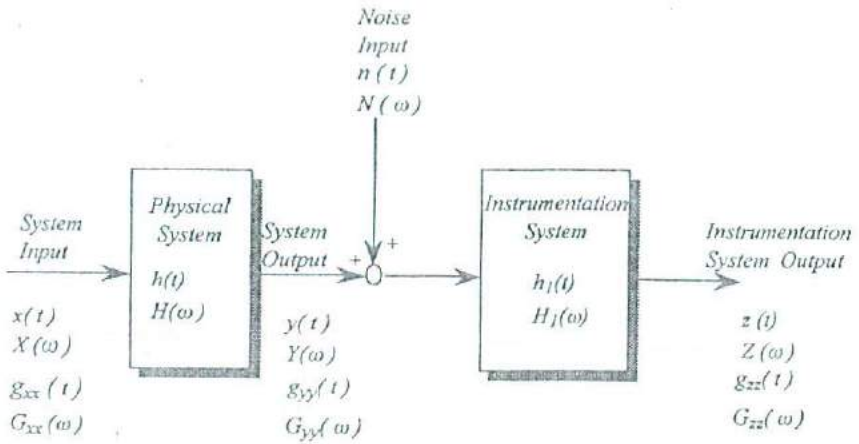


Figure 10.2 Simple physical system having a detection system with a noise input.

where K_1 and K_2 are constants. For the special case where the input $x(t)$ is "white" and $G_{xx}(\omega)$ is constant, equation (10.2-23) can be rearranged to give

$$|H(\omega)|^2 = K_3 G_{zz}(\omega) + K_4 \qquad (10.2\text{-}24)$$

The presence of K_2 and K_4 in equations (10.2-23) and (10.2-24) is due to the detection noise. If K_2 and K_4 are large compared to K_1 and K_3, respectively, then the presence of the noise may seriously deteriorate the quality of the measurement and the ability to evaluate the system response function and the parameters of the system being studied.

Cross-spectral density measurements offer a means of overcoming some of these problems under certain circumstances. Consider the case of the system in Figure 10.2 with two inputs: $x(t)$ that goes through both the physical system and the instrumentation system, and $n(t)$ that goes only through the instrumentation has been analyzed (Uhrig, 1970). The result is

$$G_{xz}(\omega) = H(\omega)G_{xx}(\omega) + H_1(\omega)G_{xn}(\omega) \qquad (10.2\text{-}25)$$

If the noise $n(t)$ is completely uncorrelated with the input $x(t)$, which is almost always the case, then $G_{xn}(\omega)$ is zero, and equation (10.2-25) reduces to

$$G_{xz}(\omega) = H(\omega)G_{xx}(\omega) \qquad (10.2\text{-}26)$$

which is identical to equation (10.2-9) for the noiseless case. This demonstrates the ability of cross-spectral density measurements to eliminate the influence of noise in many practical cases. Equation (10.2-26) can be further

reduced for a white noise input to

$$G_{xx}(\omega) = KH(\omega) \tag{10.2-27}$$

That is, the amplitude of the transfer function $|H(\omega)|$ is proportional to the amplitude of the cross-spectral density between the input and the output, and the phase angle of the system response function $\Theta(\omega)$ is equal to the phase angle of the cross-spectral density.

Coherence Function as an Index of the Quality of Measurement

As indicated above, the presence of the constants K_2 and K_4 in equations (10.2-23) and (10.2-24) can seriously degrade the measurement. We can get an index of the influence of the presence of noise on measurements by defining the coherence function $\gamma^2(\omega)$ to be the ratio of $|H(\omega)|^2$ as determined by the power spectral density method equation (10.2-9) to $|H(\omega)|^2$ as determined by the cross-spectral density method equation (10.2-19). Since equation (10.2-19) gives a result that is independent of the influence of noise whereas the validity of equation (10.2-9) deteriorates as the noise increases, this ratio is a valid reflection of the adverse influence of noise. Hence

$$\gamma^2(\omega) = \frac{|G_{xy}(\omega)|^2/[G_{xx}(\omega)]^2}{G_{yy}(\omega)/G_{xx}(\omega)} = \frac{|G_{xy}(\omega)|^2}{G_{yy}(\omega)G_{xx}(\omega)} \tag{10.2-28}$$

For the case of no input noise, the coherence function is unity. As the noise increases, the quality of the measurement using equation (10.2-9) deteriorates, and the coherence as given by equation (10.2-28) decreases. It is intuitive and readily demonstrated that the coherence function is always less than or equal to unity; that is,

$$\gamma^2(\omega) \leq 1 \tag{10.2-29}$$

Correlation and Spectral Measurements Using Pseudorandom Binary Variables

In modeling a dynamic system with an artificial neural network, one signal from the system is the input and another is the desired output. Generally, the fluctuations of the input is adequate to ensure training of the network over the desired dynamic range. However, it is sometimes necessary to introduce a small perturbation of the input to make sure that the input signal contains the desired frequency content. Generally this perturbation is either a multiple-frequency signal (sum of sinusoidal signals with frequencies spread evenly

on a logarithmic or a linear scale) or a random signal that approximates a "white" noise over the frequency range of interest.

One signal that is particularly useful and easy to implement is the "pseudorandom binary maximum-length shift register sequence" signal (Uhrig, 1970). It is a binary signal that instantaneously shifts between 0 and 1 (or between -1 and $+1$), with the shifts occurring at integral numbers of the time interval Δ. It is easily produced with software or a hardware shift register. It is a periodic signal that has (a) a narrow triangular spike as an autocorrelation function and (b) a power spectral density whose discrete values have an envelope of the general form of $\sin(x)/x$. The period of the signal ($N\Delta$) can be controlled easily by (a) the number of shifts N in one period of a shift register generating the signal and (b) the time interval Δ. By increasing N and decreasing Δ while keeping the period $N\Delta$ constant, the autocorrelation function of the signal becomes narrower, more nearly approximating a Dirac δ function, and the frequency range over which the power spectral density envelope $\sin(x)/x$ remains almost constant increases. Under these conditions, we approach the characteristics of a "white noise," and equations (10.2-20) and (10.2-21) apply. Then, the cross-correlation function between the pseudorandom input signal and the output gives a quantity proportional to the impulse function, and the cross-spectral density between the pseudorandom signal and the output gives a quantity proportional to the system response function. Since all other noise sources are uncorrelated with the pseudorandom input, they have no influence on these measurements. Indeed, two pseudorandom signals of different lengths (i.e., generated with shift registers having different time intervals and number of shifts per cycle) are independent of each other, and the cross-correlation functions with system signals are also independent of each other. Hence pseudorandom signals can be introduced at different locations in the system in order to model system response characteristics of individual components of the system. The power of cross-correlation and cross-spectral density measurements becomes apparent when it is realized that multiple sources of such "white" noise (that are, in fact, deterministic periodic variables) are independent of each other and can be injected into a system without unduly influencing the behavior of the system or the other measurements taking place.

The fact that the pseudorandom signal is periodic usually simplifies the processing of the data to obtain the cross-correlation and cross-spectral density. Furthermore, the shift register configuration for generating the pseudorandom signal can be implemented in software, and its output can be introduced directly into the input of an artificial neural network or mixed with other input signals in any desired manner. There are several variations of this pseudorandom signal and the associated shift register systems (Uhrig, 1970). A three-level pseudorandom signal that deals separately with the linear and nonlinear portions of the system response function has also been demonstrated (Gyftopoulos and Hooker, 1964).

10.3 ADAPTIVE SIGNAL PROCESSING

Much of the early work in the artificial neural network field was carried out by Bernard Widrow and his associates at Stanford University within the framework of "adaptive linear systems and adaptive signal processing" (Widrow and Hoff, 1960). Virtually all of the applications developed in this work can advantageously utilize an artificial neural network with its nonlinear capability in the place of the "adaptive linear combiner." Indeed, Widrow's Adaline is an adaptive linear combiner that utilizes a bipolar output element as its nonlinear output device in order to accomplish its tasks.[2]

Adaptive Linear Combiner

The adaptive linear combiner, a nonrecursive adaptive filter, is fundamental to adaptive signal processing, and it is an integral part of an artificial neural network. It is used, in one form or another, in most adaptive filters and control systems. It is the most important element in "learning" systems. It is essentially a time-varying, nonrecursive digital filter that is implemented in many forms, and its behavior and its means of adaptation are well understood and readily analyzed.

Multiple-Input Adaptive Linear Combiner

The general form of an adaptive linear combiner is shown in Figure 10.3. The kth input is a vector $\mathbf{X}_k$ with components $x_1, x_2, \ldots, x_n, \ldots, x_N$; that is,

$$\mathbf{X}_k = [x_{0k} x_{1k} \cdots x_{nk} \cdots x_{Nk}]^T \tag{10.3-1}$$

where the superscript T indicates the transposed vector (i.e., it is a column vector).

A weight vector $\mathbf{W}_k$ with a constituent set of adjustable weights $w_{0k}, w_{1k}, w_{2k}, \ldots, w_{Nk}$, and a summing unit produces a single output I_k. Most systems also include a bias with an amplitude x_0 equal to unity and an adjustable weight w_0. The configuration shown in Figure 10.3, known as a "multiple input adaptive linear combiner," accepts all of the components of the vector $\mathbf{X}_k$ simultaneously and produces a single output. Then the weights are adjusted, the next input vector is applied, another output is produced, the weights are adjusted again, and so on. This is the form of the classical implementation of almost all neural networks used today.

[2]There is nothing inherent in the definition of neural networks that requires that the output of the neurons be nonlinear. However, as pointed out in Chapters 7 and 8, the use of linear activation functions in the hidden layers significantly limits the usefulness and capabilities of a neural network.

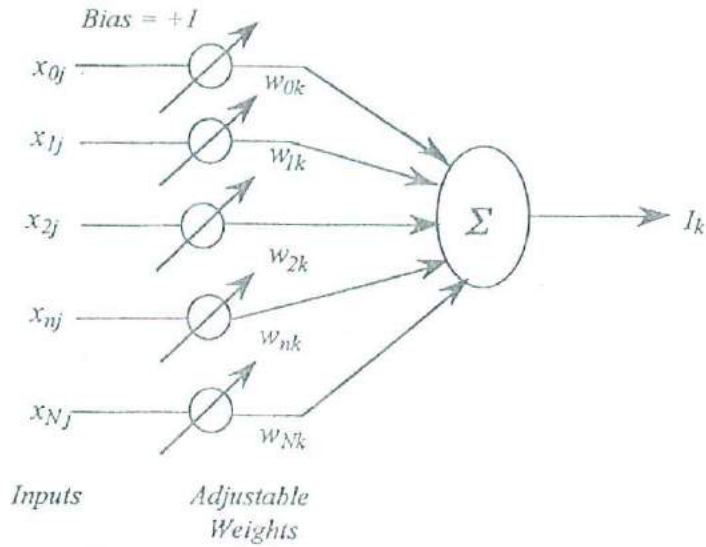


Figure 10.3 Multiple-input adaptive linear combiner.

Single-Input Adaptive Transverse Filter

An alternate form of the adaptive linear combiner (which is equally applicable to artificial neural networks in general) is shown in Figure 10.4, in which the input is applied sequentially to the input layer through a series of time delays, moving down the input layer until it reaches the end. The lines between each pair of delay units are tapped, and the signals (in addition to being sent to the next delay unit) are sent through adjustable weights to a

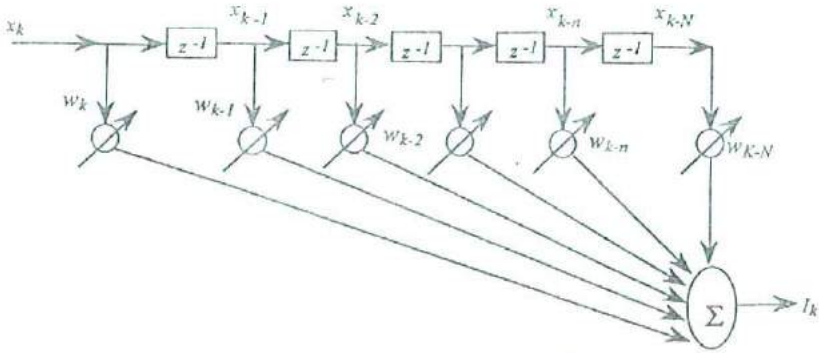


Figure 10.4 Single-input adaptive transverse filter.

summing node. Such a system has historically been called a "tapped delay line," which is quite descriptive of the process when implemented in analog hardware. In a digital computer, the process is implemented by manipulating sequential values of a sampled time series from a file. If a bias is used, it is applied directly to the summing junction through an adjustable weight. However, it is not normally required for single-input systems if the mean value has been removed from the time-varying input signal. The length of the time delay (designated z^{-1} from the "z" transform as used in digital control theory) is normally equal to the sampling interval, or some multiple of it, used in digitizing the input analog variable. The length of the input signal presented to the summing junction is the product of the number of delays in a cycle and the length of the time delay. For instance, a network having 100 input neurons (and 99 time delays) and a sampling rate of 1000 samples per second ($z^{-1} = 0.001$ sec) has an input signal that spans (100×0.001) or 0.1 sec. These parameters are related to the frequency content of the signal as well as the characteristic time constant of the system modeled in the artificial neural network. Such a system is known as a "single-input adaptive transversal filter." Its input vector is given by

$$\mathbf{X}_k = [x_k x_{k-1} \cdots x_{x-N}]^T \tag{10.3-2}$$

Input–Output Relationships

The output for the multiple-input adaptive linear combiner of Figure 10.3 is

$$I_k = \sum_{n=0}^{N} w_{nk} x_{nk} \tag{10.3-3}$$

while the output for the single-input adaptive transversal filter of Figure 10.4 is

$$I_k = \sum_{n=0}^{N} w_{nk} x_{(k-n)} \tag{10.3-4}$$

For both types of systems, we have a weight vector

$$\mathbf{W}_k = [w_{0k} w_{1k} \cdots w_{nk}]^T \tag{10.3-5}$$

With the definitions of equations (10.3-1), (10.3-2), and (10.3-5), we can express the outputs of both types of systems with a single relationship:

$$I_k = \mathbf{X}_k^T \mathbf{W}_k = \mathbf{W}_k^T \mathbf{X}_k \tag{10.3-6}$$

Desired Response and Error

Although the adaptive linear combiner can be used in both open- and closed-loop adaptive systems, the primary interest here is in closed-loop operation. Hence, the weight vector adjustment depends primarily on the output and its deviation from the desired output. The weight vector $\mathbf{W}_k$ is adjusted or optimized so as to minimize the difference between the actual output I_k and the desired (or target) output T_k—that is, to minimize the square error defined by

$$\varepsilon_k^2 = [T_k - I_k]^2 \tag{10.3-7}$$

The arrangement for dealing with this error and target output for the single-input adaptive transversal filter is shown in Figure 10.5. The minimization of square error follows the procedure described in Section 8.2 and the Widrow–Hoff Delta learning rule (1985).

Linear Control Theory

Linear control theory is well documented in the literature and is taught in most undergraduate engineering curricula. The proportional-integral-differential (PID) type of control, the most common linear control system, is discussed briefly in Section 6.1 and Example 6.1 in Chapter 6 ("Fuzzy Control") and in Section 10.5 ("Neural Network Control"). We will presume that the reader has a basic understanding of the PID control system and such concepts as stability, feedback, gain, and so on. Adaptive control and model-reference adaptive control will be explained briefly in the section of this chapter where it is introduced.

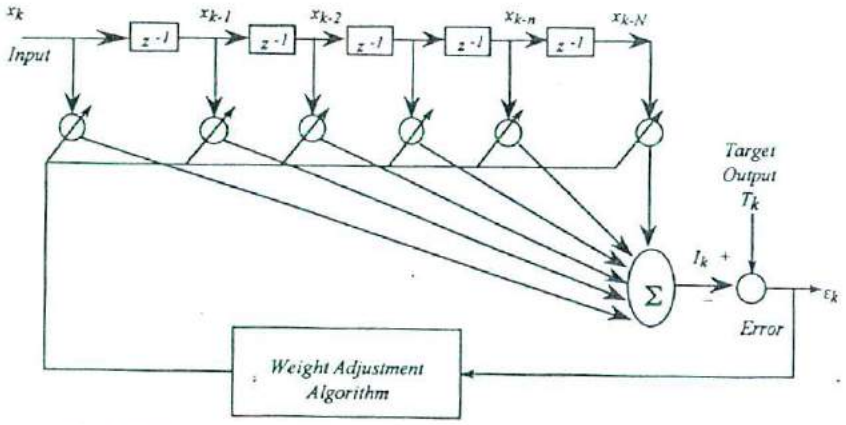


Figure 10.5 Single-input adaptive transversal filter with a target output.

The use of a single-input adaptive transverse filter (shown in Figure 10.5) to model a physical system whose characteristics are not known gives the impulse response function. After the weights have been adjusted to minimize the least squares error between the desired output and the actual output, the values of the weights sequentially from left to right give the values of the impulse response function at the corresponding time. Of course, since the number of weights is finite and an impulse response function $h(t)$ as defined in Section 10.1 extends to infinity, these weights are only an approximation of $h(t)$. As a result, the single-input adaptive transfer filter is often called a "finite impulse response" (FIR) system.

10.4 ADAPTIVE PROCESSORS AND NEURAL NETWORKS

The term "adaptive neural networks" refers to a neural network that adapts its weights to accomplish a mapping of the input to the desired output. Most neural networks that employ supervised learning belong to this category. Least squares adaptation algorithms are the basic learning systems for both adaptive signal processing systems and adaptive neural networks. Least squares minimization was discussed in Chapter 8 in conjunction with the Widrow–Hoff delta learning rule used in backpropagation training.

Linear Versus Nonlinear Systems

Although certain types of adaptive systems called "linear adaptive systems" can become linear when their adjustments are held constant after adaptation, most adaptive systems, by their very nature, have time-varying parameters and are nonlinear. Their characteristics depend on the input and the structure of the adaptive process. Adaptive systems are adjustable and depend on finite-time average signal characteristics rather than on instantaneous values of signals or instantaneous values of the internal system states. The adjustment of adaptive systems are made with the goal of optimizing specific performance measures. Hence, we can take advantage of the neural network's ability to utilize the nonlinear activation function to deal with nonlinearities in the modeling process.

Applications of Adaptive Neural Networks

Widrow and Stearns (1985) have discussed a wide variety of applications of the adaptive signal processing systems. We shall discuss many of these applications in the context of a neural network used to perform the same task. Although some adaptive systems operate in the open-loop mode, we shall deal primarily with closed-loop operation, a mode that provides "performance monitoring." Closed-loop operation, as we normally encounter it, involves measurement of the input and output signals of a system, utili-

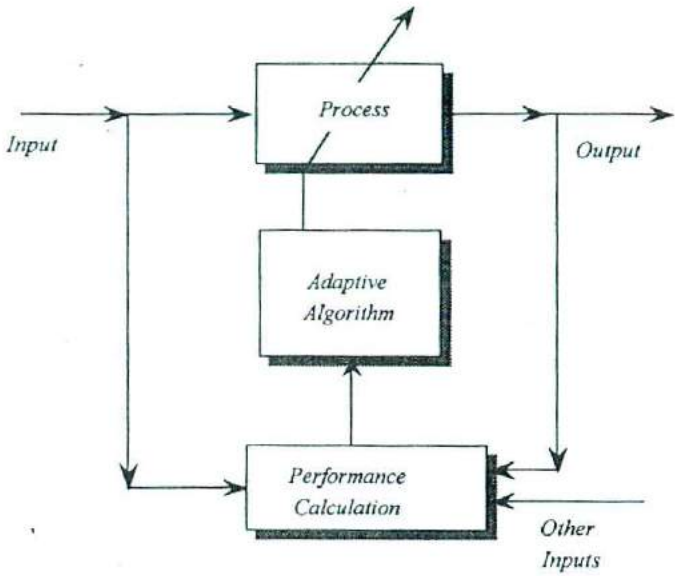


Figure 10.6 Adaptive processor and its components.

zation of a performance index (which can be as simple as a comparison of two values), and automatic adjustment of one or more input parameters of the system. These steps can be sequential discrete operations, but more commonly, all steps take place concurrently (at least in analog systems). Figure 10.6 shows the basic elements of a closed-loop adaptation system that in subsequent discussions in this section will be called an "adaptive processor." The performance calculator can be simple [i.e., the calculation of error or square error as in equation (10.3-7)] or sophisticated (i.e., the calculation of an energy or cost function using many quantities). The adaptation algorithm typically is minimization of least squares error, but it can utilize any optimization process desired. Clearly, a backpropagation neural network can be considered to be an adaptive processor.

In this chapter, the adaptive processor will be implemented as a neural network. This could be viewed as somewhat of a restriction because the type of neural network often determines the type of performance monitoring. For instance, use of a backpropagation neural network inherently involves minimization of least squares error and gradient descent optimization. However, training of neural networks can involve any type of optimization that can be used anywhere. Closed-loop operation can be used in situations where characteristic parameters of the physical system being operated are poorly known or are changing with time. Closed-loop adaptive operation can also compensate for some degree of deterioration of components in the physical system. Hence, performance monitoring can result in a more robust and/or a

more reliable system. However, the closed-loop adaptation process is not without difficulties. In some cases, the performance indices to be optimized have more than one minimum that, under some circumstances, can result in the adaptation reaching a "false optimum." Feedback can also lead to instabilities that could degrade the performance or endanger the operation of the physical systems. Even "limit cycle" instabilities (where the effect of instabilities is self-limiting due to saturation) can cause serious difficulties. Feedback systems with adaptation based on performance monitoring are subject to the same stability criteria that apply to other feedback control systems. Nevertheless, performance monitoring with adaptation of the physical system is widely used in complex systems that are difficult to analyze or model analytically.

Configurations of Adaptive Neural Network Systems

Four basic configurations for adaptive systems (Widrow and Stearns, 1985) are as follows:

1. System identification or modeling (used in adaptive control)
2. Inverse modeling (used in equalization and deconvolution)
3. Adaptive interference canceling
4. Adaptive prediction

Each of these systems uses a single adaptive processor or neural network in different configurations to carry out the adapting task in order to accomplish its desired function listed above. Each of these simple systems is subject to the limitations and problems of adaptive systems discussed above. Nevertheless, each system has been extremely successful and is capable of performing its specific task with a minimum of difficulty.

System Identification or Modeling

One of the most common processes for which adaptive neural networks are used is system identification or modeling. This involves placing the neural network in parallel with the physical system, applying the system input to the input of the network, using the system output as the desired output for the neural network, and training the neural network until the error between the system output and the network output reaches an acceptable level. This configuration is shown in Figure 10.7*a*. The single-input adaptive transversal filter discussed in Section 10.3 is usually implemented by introducing a sampled time-varying signal (a time series) into the input of a neural network where the sampled input values advance laterally along the input layer. (Note that this is not a form of lateral feedback between neurons in the input layer, but instead it is simply a means of introducing the appropriate input values to the network to represent a time series or sampled variable.) At the same

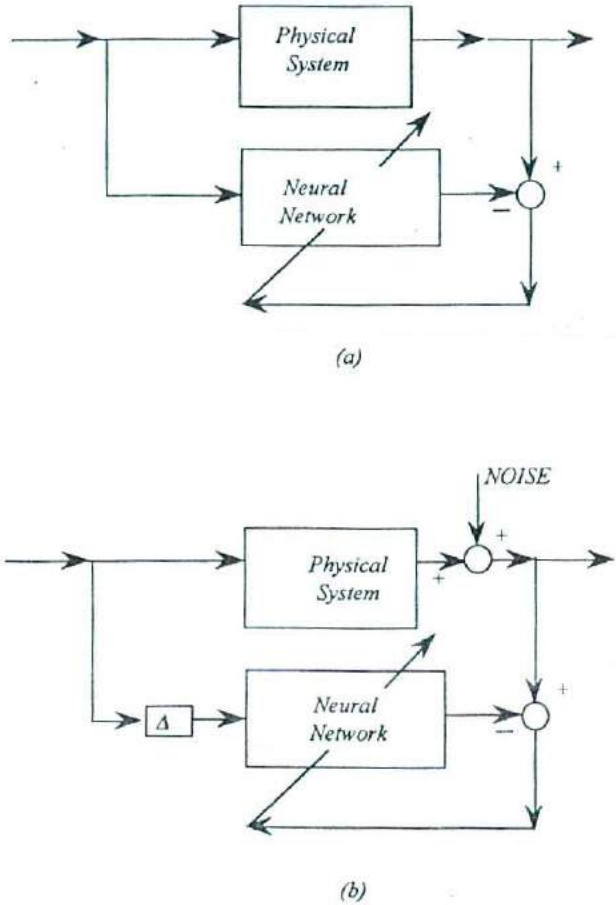


Figure 10.7 Configurations of adaptive modeling systems: (*a*) Simple modeling system. (*b*) Modeling system with delay Δ and input noise.

time, the corresponding sampled output value advances as the desired output for the single output neuron at the same rate. For multiple input–output systems, individual inputs and desired outputs may be assigned to certain parts of the input and output layers, respectively. Generally, it is necessary to specially design an artificial neural network for multiple-input, multiple-output modeling. One common arrangement is to use individual neural networks for each input and then use another neural network to combine the individual outputs.

After the network is trained, it is expected that the relationship of its input and output is the same as for the input and output of the physical system

being modeled. This will be true if the variables have been trained over the appropriate dynamic range for the particular application. For instance, if a neural network is expected to respond over a specific dynamic range (e.g., from 0.1 to 10 Hz), the input signals to the neural network during the training should cover this dynamic range. Therefore, a neural network trained over the above range should not be expected to give proper results for input signals below 0.1 Hz or above 10 Hz. Furthermore, if a periodic signal (other than a single-frequency harmonic) is used in the training, the length of the cycle should be longer than the settling time of the physical system (the time required for the impulse response to approach zero) to ensure proper modeling.

Sometimes it is necessary to introduce a time delay Δ (which has no relation to the sampling time interval z^{-1}) into the configuration in order to model the finite time that is required for a signal to move through a physical process. Indeed, the length of this delay can be a parameter that is adjusted to minimize the residual error in the neural network model. It is often necessary to introduce a noise source into the configuration if such a noise source is inherent in the processes itself. For instance, noise sometimes arises from a random process internal to the physical system (e.g., the measurement of the intensity of a radiation source involves individual events of absorptions or collisions, which are unrelated to the source emissions) and should be included in the model. Even the detection process itself may be an independent source of noise that must be included in any realistic modeling of the process and its measuring system. Figure 10.7*b* shows a modeling configuration with both a time delay and a noise source.

Inverse Modeling

In a sense, inverse modeling is the same modeling discussed above with the input and desired output signals reversed. (From the standpoint of a neural network, it really does not make any difference where the input and desired output signals originate.) Inverse models are very important in control systems where they are often put in series with the controller or the process. Figure 10.8*a* gives the configuration normally used for inverse modeling. As in direct modeling, the time for a signal to propagate through a physical system should be included in the modeling process in order to reduce the mean square error to a minimum. Noise sources, if they exist in the actual system, should also be included in the neural network model. Figure 10.8*b* shows the inverse modeling configuration with a time delay Δ and a noise source.

An inverse model will have frequency response characteristics that are opposite to those of the original system: If the amplitude of the frequency response function at a particular frequency is decreasing in the system, it will be increasing in the inverse model. As a result, the product of the amplitudes of a model and its inverse model is equal to unity when the system and inverse model are connected in series.

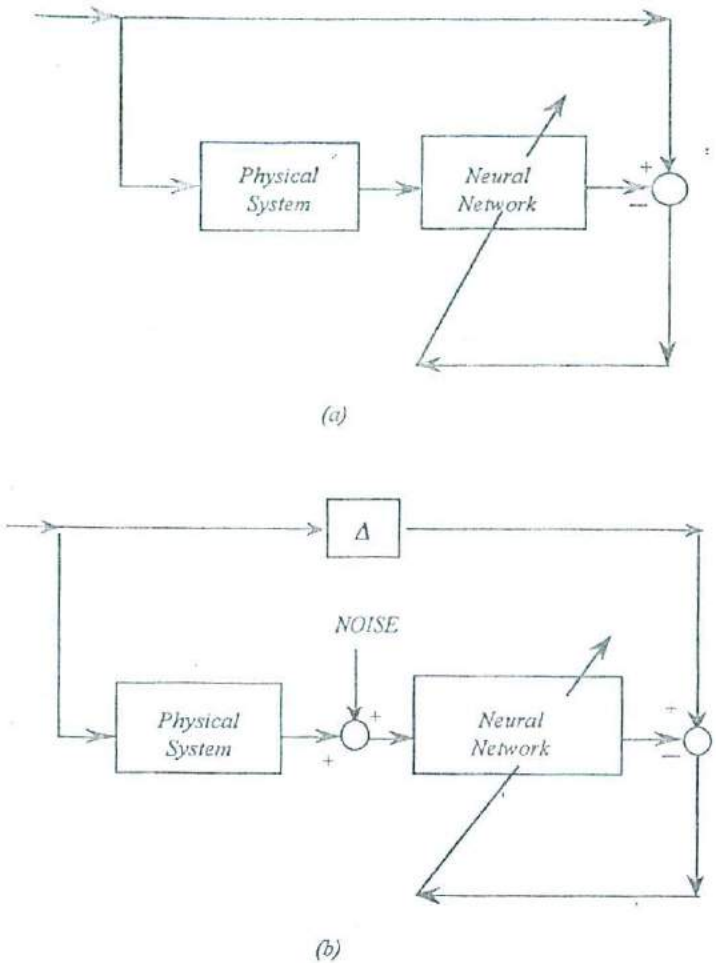


Figure 10.8 Configurations of adaptive inverse modeling systems: (*a*) Inverse modeling system. (*b*) Inverse modeling system with delay Δ and input noise.

This feature is often used in practical applications to negate the degrading influence of an instrumentation system. For instance, if the frequency response function $H_1(\omega)$ of the instrumentation system in Figure 10.2 falls off and significantly influences the measurement (we had previously assumed that its frequency response was constant over the frequency range of interest), then we can introduce an inverse model of this instrumentation system in series with the instrumentation to negate this adverse influence.

Equalization and Deconvolution

In communications, telephone and radio channels are dispersive; that is, high-frequency signals travel faster and are attenuated more rapidly than low-frequency signals. In dispersive channels, an inverse adaptive filter can be placed at the receiving end to "equalize" the channel—that is, provide a frequency and phase response in the receiver that is the inverse or reciprocal of that of the channel itself. In effect, it "deconvolves" the dissipative influence of the channel characteristic and restores the original signal characteristics. It avoids destructive interference that would virtually destroy the ability of communications channels to transmit information. High-speed transmission of digital signals, which is particularly susceptible to this problem, would not be possible without equalization and deconvolution.

The physical arrangement used for equalization and deconvolution is the same as the one used for inverse modeling in Figure 10.8*b*. The neural network attempts to recover a delayed version of the signal which may have been altered by the slowing varying system characteristics and which contains noise. The delay is to allow for the propagation time through the system and the neural network. In effect, this system attempts to deconvolve (undo) the effects of the communications channel. It also has applications in control systems.

Interference Canceling

Another very useful application of the adaptive processor is interference canceling. One of the most obvious applications is to cancel out 60-Hz interference from ordinary a.c. electrical power supplies. In this case, the interference frequency is known (60 Hz) and constant, and cancellation is relatively easy. The challenge is to cancel noise when the nature of the interference is unknown and changing. The advantage of noise cancellation as compared to noise filtering is that noise cancellation does not attenuate the signal and hence gives a much higher signal-to-noise ratio under virtually all conditions.

One of the most successful applications of this technology has been the cancellation of background noise from voice communication in small aircraft. The system used is shown in Figure 10.9, where the desired signal S (e.g., voice) is corrupted with background noise N (e.g., aircraft engine noise) when the microphone picks up both signals. The secret to success is to find a source of background noise N' that does not contain the desired signal S but is reasonably correlated with the interference noise N (i.e., a second microphone located away from the speaker's mouth). In this case, the adaptive processor is configured to produce an output Y that closely resembles N which is then subtracted from the speaker's microphone output so that the overall output S' closely resembles the original input signal S.

Another application of interference cancellation is the separation of the weak heartbeat of a fetus from the strong heartbeat of the mother. A

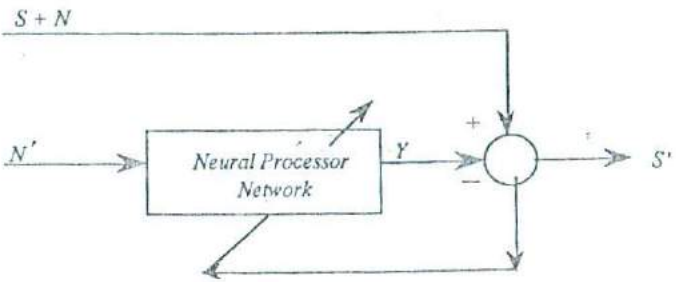


Figure 10.9 Noise cancellation system.

microphonic sensor is attached near the fetus and picks up the heartbeat of both the mother and the fetus, thus providing the $S + N$ signal. Another microphonic sensor is attached to the mother far from the fetus, where only her heartbeat is picked up, thus providing the N' signal. The system of Figure 10.9 then adapts to produce the fetal heartbeat signal S'.

Prediction

Another configuration of the adaptive processor is to predict the future of a signal from its behavior in the past. Figure 10.10 shows the system configuration used. The input signal $S(t)$ is delayed by an amount Δ before it is presented to the neural network. Then the neural network is adjusted so that

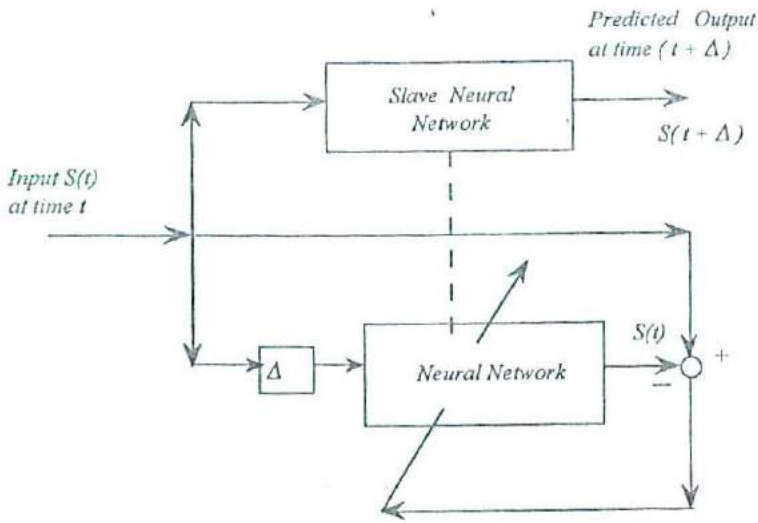


Figure 10.10 Configuration of prediction system.

the error between the delayed input and the undelayed signal is minimized. In effect, the neural network has been trained to produce an output signal $S(t)$ from an earlier input signal $S(t - \Delta)$. What is needed for prediction is a neural network that uses an input $S(t)$ and produces a future value $S(t + \Delta)$. This can be accomplished by using a "slave" neural network which has the same structure as the original neural network and whose weights are updated in real time to be identical to the weights in the original neural network. Such an arrangement is shown in Figure 10.10, where the "slave" neural network has an input signal $S(t)$ and an output $S(t + \Delta)$.

10.5 NEURAL NETWORK CONTROL

The field of control theory and systems is treated exhaustively in literally thousands of books and publications. The purpose of this treatment is to introduce only those concepts that are important and useful in dealing with neural network control systems. Control, by definition, is action taken to achieve a desired result or goal. For instance, the temperature in modern homes is controlled by a simple, but effective, on–off action, activated by a thermostat that turns the furnace on when the temperature falls below a specified temperature and turns it off when room reaches a slightly higher (typically 2–3° higher) temperature. The room temperature is compared with the desired temperature (the setpoint), and the difference constitutes an error signal that activates a simple control system to turn the furnace on and off.

Werbos (1992) has divided neural control into five categories: (1) supervised control, (2) inverse control, (3) adaptive control, (4) backpropagation through time, and (5) adaptive critic methods. Each of these methods will be discussed briefly here, and some of them will be covered in more depth in later sections.

There are many neural-network-based approaches to control problems, but most of them consist in using neural networks for two basic processes that are duplicated and combined as appropriate, along with time lags, to achieve the specific objective of the control system. These two basic processes are system or process modeling (system identification) and some means of control (often based on an inverse model of the system or process). A simple open-loop control system with a single input and output is shown in Figure 10.11*a*. Neural control systems are sometimes used to complement PID control systems by adaptively tuning the parameters of the controller to match the changing system characteristics. "Black box" identification and model-based controller optimization are commonly used in current neural control systems. For simplicity, single input–output systems are discussed here. However, multiple input–output systems are treated in the same manner; indeed, most neural control systems are multivariable systems.

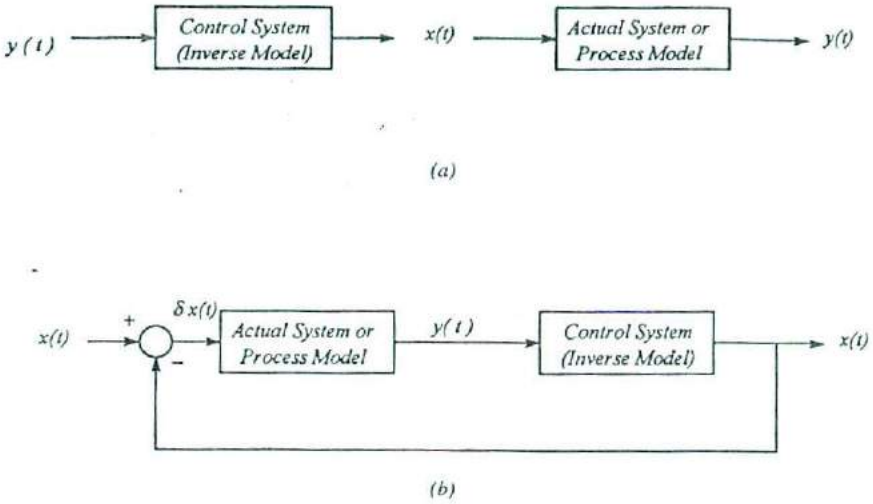


Figure 10.11 Simple open-loop (*a*) and closed-loop (*b*) control systems. $x(t)$ and $y(t)$ are system input and output, respectively, and $\delta x(t)$ is the change in $x(t)$.

For systems that are linear or can be linearized over the required range of operation, conventional linear control theory is adequate for most applications. However, most complex systems are nonlinear and require either a very sophisticated mathematical treatment or a simulation. Since the needed parameters for mathematical modeling and simulation models of most complex nonlinear systems are usually not available, an experimental determination of the system characteristics often becomes necessary.

More challenging is the mathematical derivation of an inverse model from a system response function (i.e., taking the inverse Fourier transformation if indeed it actually exists). Even for relatively simple linear systems, such an inverse operation can be quite difficult and sometimes impossible. For nonlinear systems, it is almost always impossible, thereby requiring approximations and simplifications or, more commonly, modeling of the system or process and its inverse configuration based on experimental data. This is where neural networks become very useful. Data from tests carried out on the system can be used to train a neural network to emulate the system behavior, thereby providing a neural network model, of the system or process. Because of the unique characteristics of neural networks, it is usually possible to reverse the input and output data provided to another neural network and train an inverse model of the system or process. When connected as shown in Figure 10.11*a*, these two neural networks become an open-loop control system.

Regulatory Control

Regulatory or closed-loop control is the most common control method; this is the essence of the room temperature control discussed earlier, where the control action is based on an error signal representing the difference between the desired temperature and the actual room temperature. (See Figure 10.12.) Indeed, error feedback is the essence of the training process in most supervised neural networks. For closed-loop control systems, the configuration of Figure 10.11*a* is often reversed so that the feedback signal is the input to the actual system or process model, which is often compared to a desired input $x(t)$. In this case, $x(t)$ is the input for the comparator and $\delta x(t)$ to be the input to the actual system or process model as shown in Figure 10.11*b*.

More common is the PID controller, a second-order control system where the feedback is a weighted sum of three quantities: The deviation of the output variable from its desired value (i.e., the error), the current derivative of the error, and the integral of the error over some time period are fed back in an effort to control the output in the face of fluctuating parameters and/or conditions. This arrangement is shown in Figure 10.13. In recent years, multiple input–output and multiple process systems have become

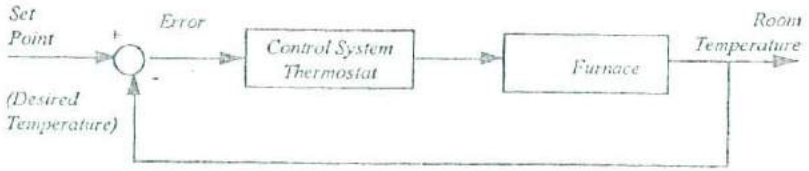
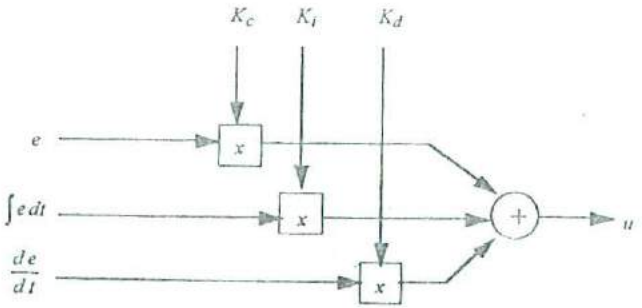


Figure 10.12 Closed-loop control system for a home heating system.



$$u = K_c e + K_i \int e\,dt + K_d \frac{de}{dt}$$

Figure 10.13 The PID controller structure. The symbol e is the instantaneous error.

ncreasingly prevalent, but virtually all of them are second-order linear s, ems.

Contro'ers also use control closed-loop operation through adjustment of paramete e.g., the "tuning" of the gains of the three feedback components of a PID trol system). Autotuning (on-line adjusting the parameters of loop controllers, or adjusting their "setpoints" for optimal performance) is a form of adaptive control. The use of a feedforward controller to provide a steady-state process input signal provides faster response and enables a feedback controller to reject noise and improves disturbance handling. Here again, these control systems fall within the linear systems domain.

The assumption of linearity is common, but it does not represent many real-world systems and processes. Most nonlinear systems are linearized over a limited range of operation. In many cases, the assumption of linearity is reasonable and produces good results, especially when a small range of adjustment is involved. Indeed, neural control is often used to complement linear control systems. Nevertheless, neural networks with their inherent ability to model nonlinear behavior show advantages that are important in many cases, particularly with complex systems.

Both the system identification and control portions of neural control can be viewed as nonlinear optimization processes; they seek to find neural network parameters (i.e., the weight matrix) for which some cost function is minimized. The type of cost function differentiates the different types of neural control concepts discussed below.

xample 10.1 Multi I/O System. An example of a multiple-input/ nultiple-output, open-loop inverse control system is a neural network to ontrol the gas–tungsten arc welding process developed by Mid-South Engi- ering and Vanderbilt University for the National Aeronautics and Space Administration (Andersen et al., 1991). This approach is necessary because the comp lexity of the physics of the arc, the molten pool, and the surrounding heat affected zone were virtually impossible to model using first principles. The lack of reliable, general, and yet computationally fast physical models of such a multivariable system makes the design of a real-time conventional controller a difficult task. Relationships between the various process inputs and outputs are nonlinear and not well-defined, and the process variables are coupled (i.e., a change in any given input parameter affects more than one output parameter). The arc welding process is controlled by a number of parameters, and the objectives of the welding process are specified in terms of the parameters shown in Figure 10.14.

An inverse open-loop system is used here because of the difficulty of providing feedback for closed-loop control purposes since the outputs of the welding process are very difficult to measure in real time. The weld must cool before physical measurements can be made. However, such output parameters as welding bead width and bead penetration can be measured off-line for purposes of modeling a neural network model from which they can be

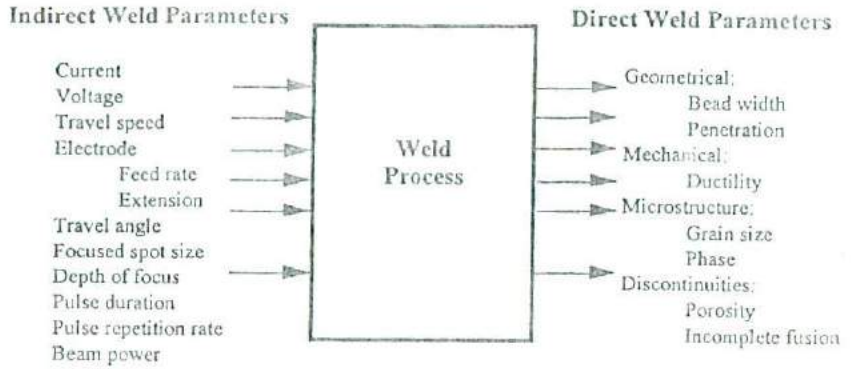


Figure 10.14 Welding illustrated as a multivariable process.

inferred on-line. The input parameters that are controlled (e.g., current, voltage, travel speed, etc.) can be measured on-line. Table 10.1 gives data obtained from test welds conducted over the expected dynamic range of operation for NASA (Andersen et al., 1991). The boldface data were used for testing the validity of the neural network given in Figure 10.15 after the rest of the data was used for training with the backpropagation algorithm. When

Table 10.1 Data used to train and test backpropagation networks for weld modeling and equipment parameter selection

Weld #	Workpiece Thickness [in]	Travel Speed [in/min]	Arc Current [A]	Arc Length[in]	Bead Width [in]	Bead Penetr. [in.]
1	0.125	6.0	80	0.100	0.118	0.024
2	0.125	6.0	100	0.100	0.165	0.051
3	**0.125**	**6.0**	**120**	**0.100**	**0.213**	**0.087**
4	0.125	6.0	140	0.100	0.256	0.126
5	0.125	4.0	100	0.100	0.216	0.071
6	**0.125**	**5.0**	**100**	**0.100**	**0.181**	**0.063**
7	0.125	7.0	100	0.100	0.153	0.043
8	0.125	6.0	100	0.090	0.161	0.063
9	0.125	6.0	100	0.080	0.161	0.075
10	**0.125**	6.0	**100**	**0.070**	**0.157**	**0.083**
11	0.250	6.0	80	0.100	0.110	0.028
12	**0.250**	**6.0**	**100**	**0.100**	**0.142**	**0.047**
13	0.250	6.0	120	0.100	0.181	0.063
14	0.250	6.0	140	0.100	0.205	0.075
15	**0.250**	**4.0**	**100**	**0.100**	**0.157**	**0.071**
16	0.250	5.0	100	0.100	0.150	0.059
17	0.250	7.0	100	0.100	0.134	0.043
18	0.250	6.0	100	0.090	0.142	0.051
19	**0.250**	**6.0**	**100**	**0.080**	**0.142**	**0.055**
20	0.250	6.0	100	0.070	0.142	0.059

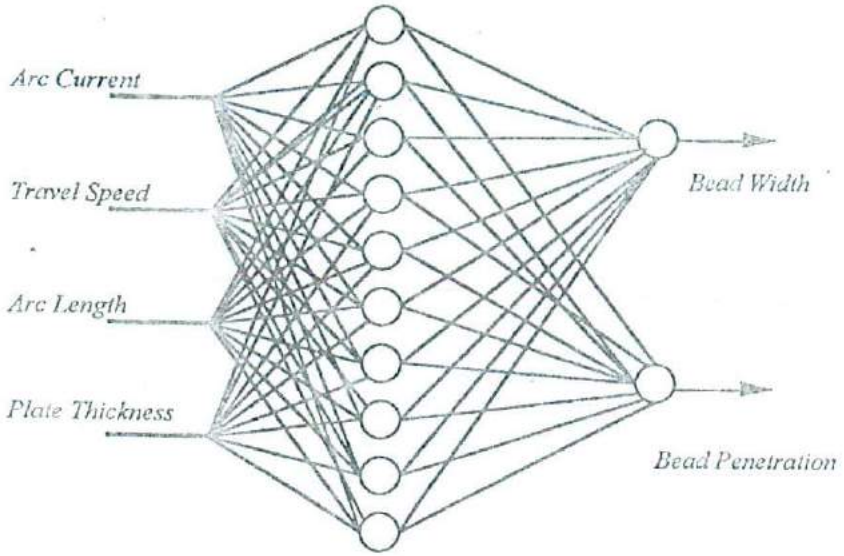


Figure 10.15 The neural network used for weld modeling.

the test data (not used in training) were introduced into the neural network, the predicted bead width and depth of penetration agreed with the actual measurements to within about $\pm 5\%$. This trained neural network now constitutes a (direct) model of the welding process.

The data in Table 10.1 were also used to train an inverse model of the welding process to provide a control neural network for parameter selection for the welder. This inverse model has the desired weld parameters as inputs and the control parameters for the welder as outputs. It can provide the welder controls with the information needed to produce a proper weld with the desired dimensions.

To simulate the performances of such a welding process, the direct and inverse neural network models were coupled together as shown in Figure 10.16 to form a cascade model of the control system (parameter selector) with the welding process. This arrangement is substantially the same as the open-loop control configuration shown in Figure 10.11*a*. A comparison of the outputs of the two neural network models with the data in Table 10.1 showed the errors of the inverse model output (first neural network) and the process model output (second neural network) to be about 10% and 2%, respectively. The reason for the low error in the process model output is that the errors in neural networks (which effectively are the inverse of each other) when trained on the same data tend to cancel each other out. □

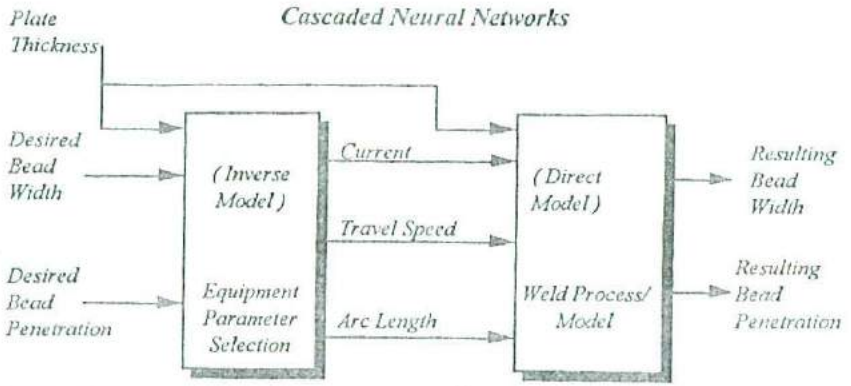


Figure 10.16 A cascade of an equipment parameter selection and welding model networks.

Neural Adaptive Control

Linear adaptive control has been a standard topic in control theory for at least three decades, and there appears to be little advantage of using neural networks in this domain except as a basis for departure into nonlinear neural adaptive control. Narendra et al. utilized neural networks to carry out linear adaptive control as a basis for later work in nonlinear control (Narendra and Parthasarathy, 1990). The goal of adaptive control is to maintain optimal performance as measured by some index of performance (e.g., efficiency, minimum emissions, etc.) under changing plant parameters or operating conditions. Often, the index of performance and optimization algorithms are included in a reference model. The difference (error) between the actual output of the system and the desired output as provided by the optimal reference model serves as the basis for adjusting parameters to improve performance. Because of the important role adaptive control plays in real-world situations, it is discussed extensively in the next section.

Supervised Control

Supervised control involves using a neural network to mimic the behavior of a conventional (i.e., PID) controller or even the behavior of a human being controlling a process or system. The neural network receives the same input and (desired) output as the PID or human controller, and training (typically backpropagation) proceeds in the conventional manner. When training is completed over the appropriate range of variables, the trained neural network replaces the PID or human controller. The major concern here is that the performance of the neural network control system can be no better than

the PID or human controller. Even so, PID and human control have often proven to be remarkably effective. It is a well-known fact that a human being's ability to recognize almost imperceptible trends and behavior has resulted in adequate performance for many systems and that the PID controller has been the "workhorse" of the control industry for over half a century. More importantly, supervised control provides a starting point from which more sophisticated control systems can be used to improve performance of a system.

Inverse Control

Inverse modeling as discussed in the previous section can readily be adapted to neural control. Inverse modeling involves training a neural network arranged in accordance with the configuration shown in Figure 10.8*b* over the appropriate range of variables. Such systems are typically used in an open-loop mode as shown in Figure 10.11*a*. The operator simply provides an input equal to the output that is desired.

The major concern is that the inverse configuration actually exists and is physically realizable. For instance, if several different inputs produce the same output, then the inverse function does not exist. Another example is a system model where the gain goes to zero under some conditions. Then the inverse model would need infinite gain.

Backpropagation Through Time

In backpropagation through time (BTT), the user specifies a model of the external environment as well as an index of performance (utility function) to be maximized. Backpropagation is used to predict the derivative of this index of performance, summed over all future times with respect to current actions. These derivatives are then used to adapt the artificial neural network that provides the output actions. This approach is used because the designer can select any index of performance to be optimized, and the method accounts precisely for the impact of present actions on future values of the index of performance. BTT is basically equivalent to the calculus of variations as used in control theory. The only essential difference is that BTT includes a way of calculating the derivatives of the utility function. Its disadvantages are that it requires a model of the external environment which should be noise-free and exact and that it requires calculations backwards through time, which is not consistent with real-time learning. However, real-time learning has been implemented by dividing experience into distinct "experiments" and updating weights after each experiment is analyzed (Werbos, 1992).

Adaptive Critics Method of Neural Control

The adaptive critics method of neural control is a form of reinforcement control in which an index of performance to be optimized is supplied by the

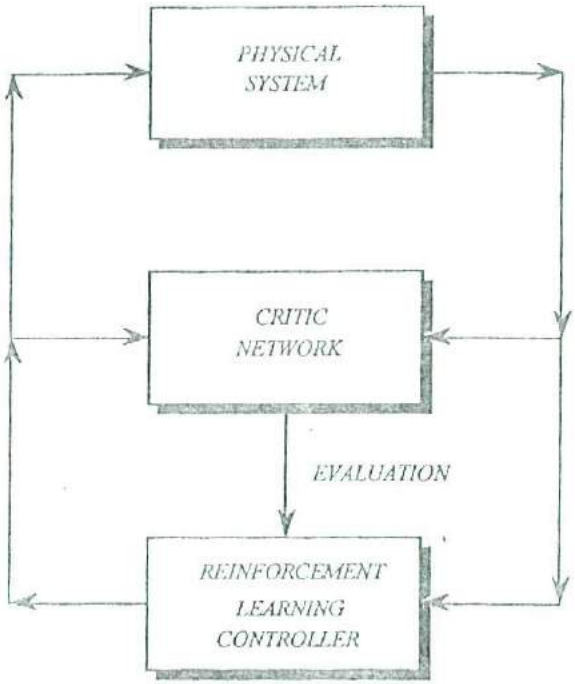


Figure 10.17 Adaptive critic reinforcement learning control system.

user. The long-term optimization problem is solved by using an additional artificial neural network (called the critic network) that evaluates the progress that the system is making and provides input to the reinforcement learning controller. This arrangement, shown in Figure 10.17, is particularly useful for situations where the model of the physical system is vague and ill-defined (e.g., the overall performance of a plant as measured by its total emissions to the atmosphere is such a system).

Example 10.2 Monitoring and Improving Heat Rate of a Power Plant. In the past few years, several systems for monitoring the heat rate (proportional to the reciprocal of efficiency) of power plants have become available. Generally, these systems are based on a first-principles model involving mass flow and energy balance equations applied to the many subsystems of a power plant. Typically, the model involves assumptions of idealized conditions, linearizations, and use of experimental correlation coefficients that are valid over limited ranges. The alternative proposed here is to take advantage of a neural network's ability to model nonlinearities and nonideal conditions inherent in any complex system. The thesis here is that a neural network

model is more realistic under "real-world" conditions and that any subsequent analysis (e.g., optimization) is more effective than similar analyses performed on first-principles models. Because of the sensitivity of cost savings to heat rate, even a small improvement in heat rate can have a large financial impact (e.g., an improvement of only 0.1% in efficiency in a 1000-MWe power plant can result in about $500,000 per year additional revenue at current prices).

Guo and Uhrig (1992) carried out the modeling of the thermodynamics of TVA's Sequoyah Nuclear Power Plant, Unit 1, using data taken weekly over about a year. Because of the complexity of the plant, use of ordinary multilayer perceptron networks trained using backpropagation was not adequate as indicated by the fact that the training error was large and the network output was not equal to the desired output used in either the training or testing sets. Rather, it was necessary to utilize a hybrid network (N-Net 210) developed by Pao (1989) in which a Kohonen neural network is used to cluster the data and then the backpropagation neural network is trained to a very small system error using the centroids of the clustered data as the inputs and the corresponding heat rates as desired outputs for training. When the original data (not the centroids) were presented to the trained network, the average error was about 0.06%.

Sensitivity Analysis. The sensitivity analysis procedure discussed in Section 8.3 was applied to the trained network. This process gave sensitivity coefficients that were ranked in order based on absolute value, since the sign only indicates the direction that heat rate moved in response to a positive change in the input perturbation. The three most important variables for heat rate based on this sensitivity analysis were (1) unexpected power deviations, (2) measured condenser backpressure, and (3) condenser circulating water inlet temperature.

Item 3 is a function of the environment and cannot be controlled. Item 2 can be controlled only to the extent that it does not go below the saturation temperature for the condenser circulating water. Item 1 is a calculated value that cannot be controlled. At this point, it appears that little has been gained from the modeling and sensitivity analysis since the three most important inputs to the network model, as far as heat rate is concerned, cannot be controlled. However, a second modeling sensitivity and analysis can be performed to determine which inputs are important to any of these three "most important" variables. This was done for item 1, unexpected power deviations.

This second modeling was carried out by using the same hybrid neural network to cluster the data and predict a new output, item 1, unexpected power deviations. Again a sensitivity analysis was performed indicating that the three most important variables, as far as unexpected power deviations is concerned, are (1) total unaccounted electrical losses, (2) condenser thermal losses, and (3) auxiliary steam loads. This tells us that unexpected power deviations, and hence the plant efficiency, can be improved by reducing

electrical losses, condenser thermal losses, and auxiliary steam loads. Each of these items can be investigated individually.

A further study of the original sensitivity analysis shows that the three variables that have the least influence on efficiency are (1) feedwater flow, (2) reactor power (within limits), and (3) impulse pressure at the turbine inlet. Since one's intuition would indicate that all three of these items might be important, this sensitive analysis has directed the plant personnel away from these items and toward those items listed in the previous paragraph that are important.

Iterative Procedure for Improving Plant Performance. Since plant conditions are always changing, keeping the plant at peak efficiency can be carried out using the following iterative process: (1) Train a neural network model of the plant using on-line data, (2) carry out a sensitivity analysis (or other analysis) to determine which two or three variables are the most important to heat rate, (3) adjust one or more of these variables in the direction indicated by the sensitivity analysis, and (4) wait for the plant to reach thermodynamic equilibrium. This four-step process could be carried out every few minutes with the interval being determined by the time necessary to reach thermodynamic equilibrium. This procedure ensures that the plant is always moving toward peak efficiency for the configuration and operating conditions that exist at the time of the analysis. If the plant condition changes (e.g., one train of feedwater heaters is taken out of service for maintenance) or the operating conditions change (e.g., the plant load changes or the condenser water temperature changes due to environmental conditions), it is still possible to move toward a more efficient (but different) configuration under the existing circumstances. □

10.6 SYSTEM IDENTIFICATION

Securing knowledge of the dynamics of a process being controlled is the first step in control. Sometimes *a priori* knowledge about the process is available in the form of a parameterized model where the parameters can be estimated from process input–output data. First-principles models are typically nonlinear, which typically must be linearized; they are usually valid only over a limited range of performance. Often these process models are relatively simple; for example, several second-order systems with lags are often adequate to represent a chemical process. In systems for where first-principles models are not available, the modeling procedure (nonparametric identification) discussed below (sometimes called "black box" modeling) is useful.

Nonparametric Identification

Nonparametric identification develops "black box" models of the input–output relationship as discussed in Section 10.3. Neural network nonparametric

process models can be seen as a nonlinear extension of the system identification problem (Tsung, 1991). For instance, the adaptive transverse filter shown earlier in Figure 10.4, when implemented by a neural network, becomes a *finite impulse response* (FIR) network, which is a nonparametric identification system. The network is provided with an input vector of weighted past samples of the variable as a means of modeling dynamic phenomena. The number of samples must be sufficient to provide an interval of time long enough that no input signal prior to x_{k-N} will have any significant effect on the response at time k.

Parametric Identification

Parametric identification identifies structural features and parameter values for models of real-world physical systems. This includes identification of the

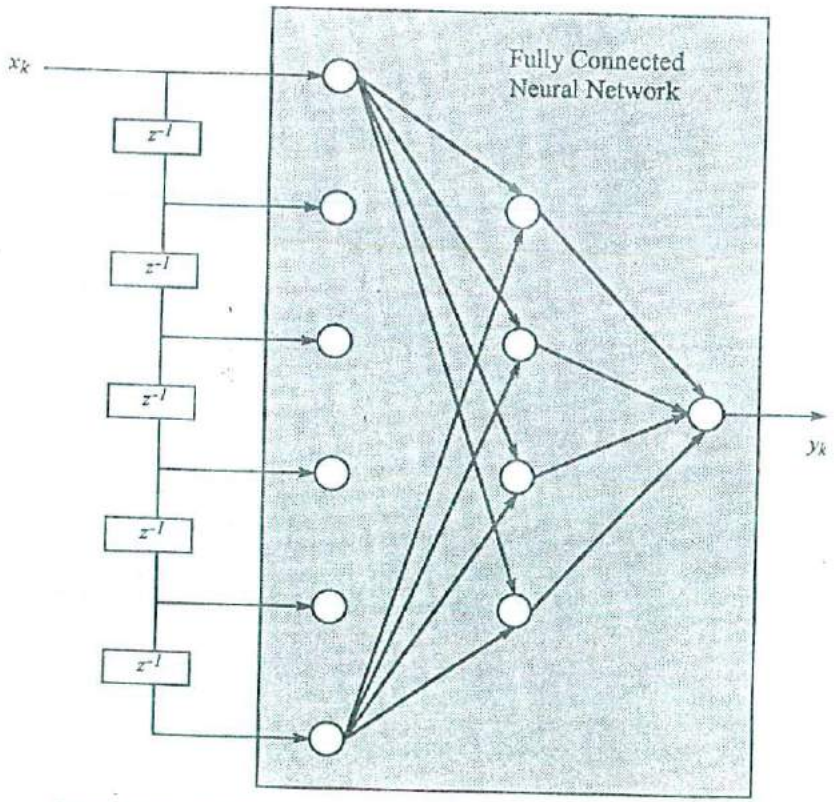


Figure 10.18 Dynamic neural network with time-delayed direct inputs.

model structure (i.e., the form of linear or nonlinear differential or difference equations as well as parameter estimation where the model structure is known). Neural networks trained through supervised learning can be used for both structure identification and parameter estimation. As structure identifiers, they can be trained to select elements of a model structure from a predetermined set. Structure identification with neural networks requires that the space of likely model structures be known in advance. Neural network parameter estimators, however, generate parameter values for a given structure or set of structures (Piovoso et al., 1991; Foslien et al., 1992). Neural network structure identifiers and parameter estimators are both trained off-line with a generalized simulation system. In structure identification, for example, the network output is a vector of structural features and network weights that are optimized to minimize the sum square error between the actual feature and its computer representation.

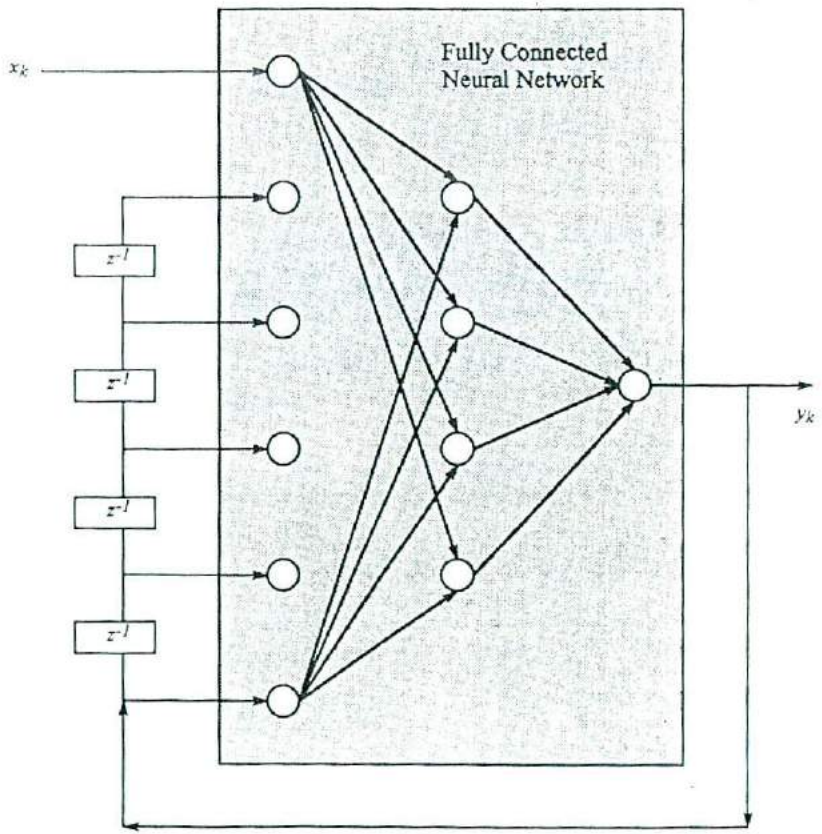


Figure 10.19 Dynamic neural network with time-delayed recurrent inputs.

Models of Dynamic Systems

Since most dynamical systems have temporal behavior, time-delayed versions of the input and/or the output signal are needed to properly model the system. Figure 10.18 shows a neural network with time-delayed versions of the input signal. When a dynamic system's current output usually depends on its previous outputs, recurrent connection with time-delayed versions of the output fed back to the inputs as shown in Figure 10.19 are needed. If the output of the system is not independent of previous inputs, then a time-delayed versions of both the input and output as shown in Figure 10.20 are needed.

In system identification, the neural network is connected in parallel with the system being modeled. Again, dynamic systems require time-delayed direct and recurrent inputs of the type shown in Figures 10.18 to 10.20.

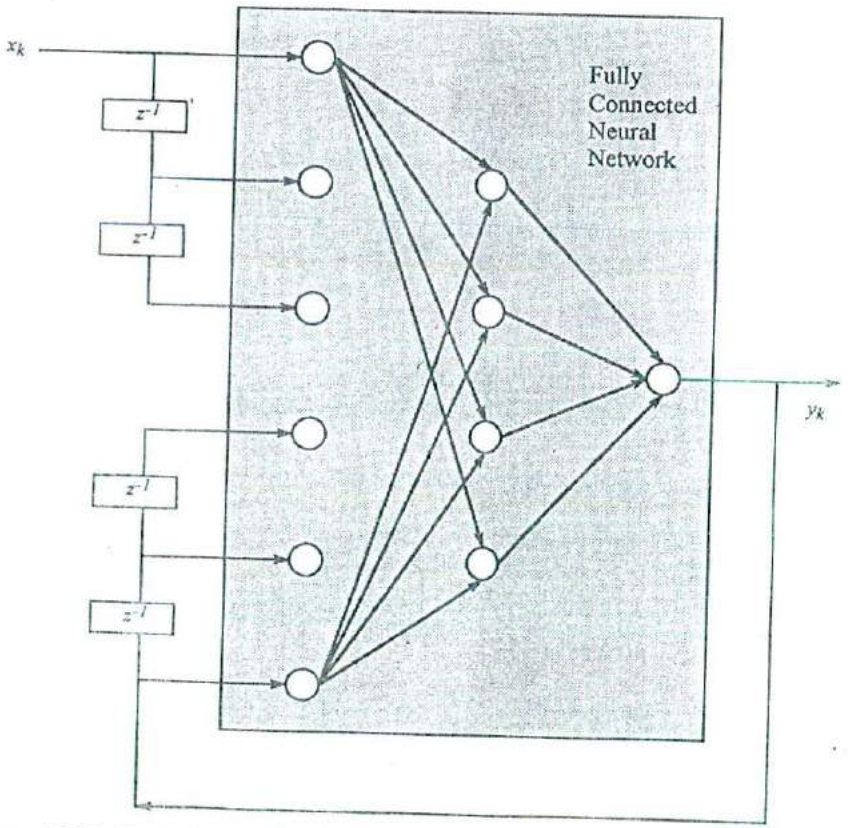


Figure 10.20 Dynamic neural network with time-delayed direct and recurrent inputs.

Figure 10.21 shows the nonrecurrent parallel identification model with time-delayed direct inputs. Figure 10.22 shows the recurrent parallel identification model with time-delayed recurrent (feedback) neurons as inputs. Unfortunately, a recurrent network can become unstable due to the feedback loop between its output and input, and there is no guarantee that the output will converge to a stable configuration. This can be solved by feeding back the signal from the system, not the neural network model output, in the recurrent series–parallel identification model shown in Figure 10.23. Finally, when there is need for both the input and output to be delayed, the general series–parallel identification model shown in Figure 10.24 is used.

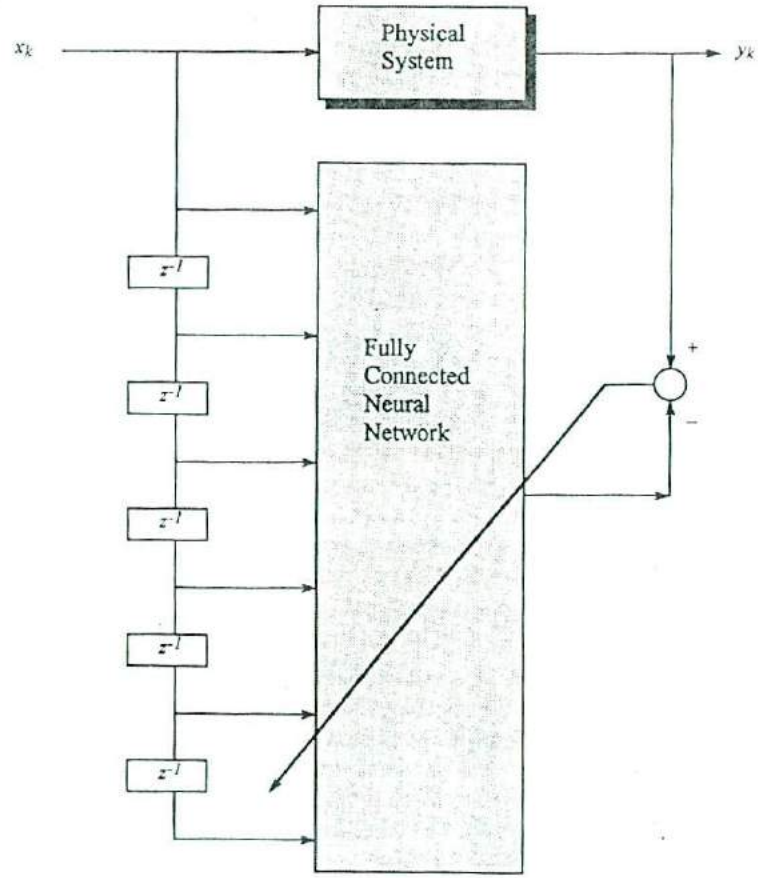


Figure 10.21 Nonrecurrent parallel identification model with time-delayed direct inputs.

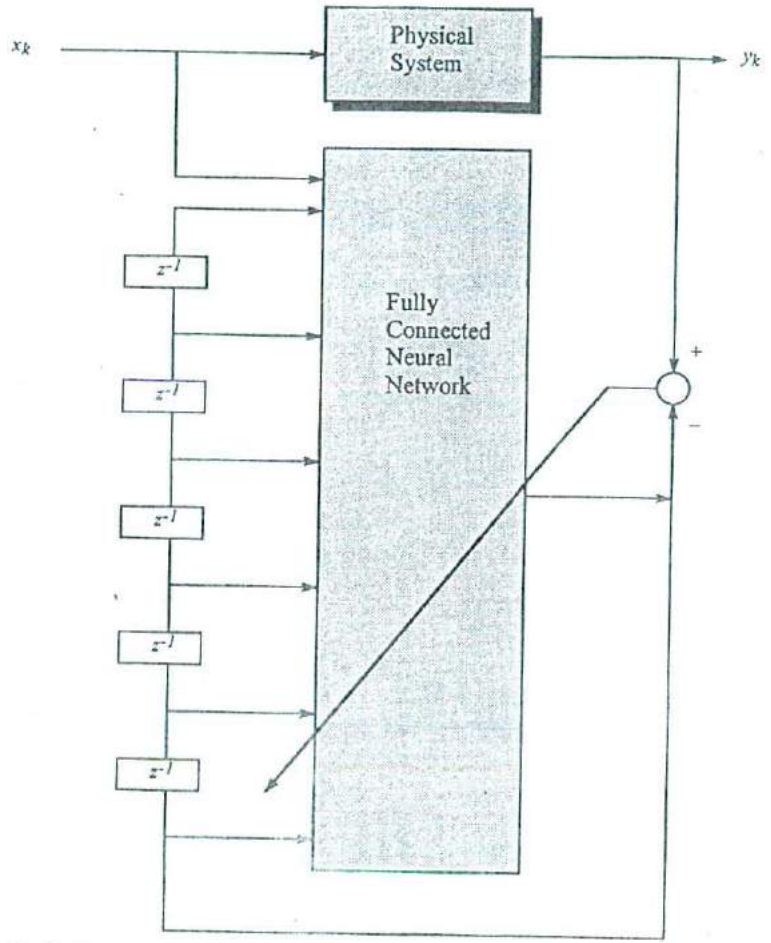


Figure 10.22 Recurrent parallel identification model with time-delayed recurrent outputs from the neural network.

10.7 IMPLEMENTATION OF NEURAL CONTROL SYSTEMS

The status of neural control is well-defined in two recent books based on symposia: *Neural Networks for Control* (Miller et al., 1990) and *Handbook of Intelligent Control—Neural, Fuzzy, and Adaptive Approaches* (White and Sofge, 1992). Particularly valuable contributions in the neural control field include Narendra and Parthasarathy (1990); Narendra (1992), Werbos (1990), Swiniarski (1990), Nguyen and Widrow (1990), Tsund (1991), and Dong and McAvoy (1994). Samad (1993) has explored neural network-based approaches

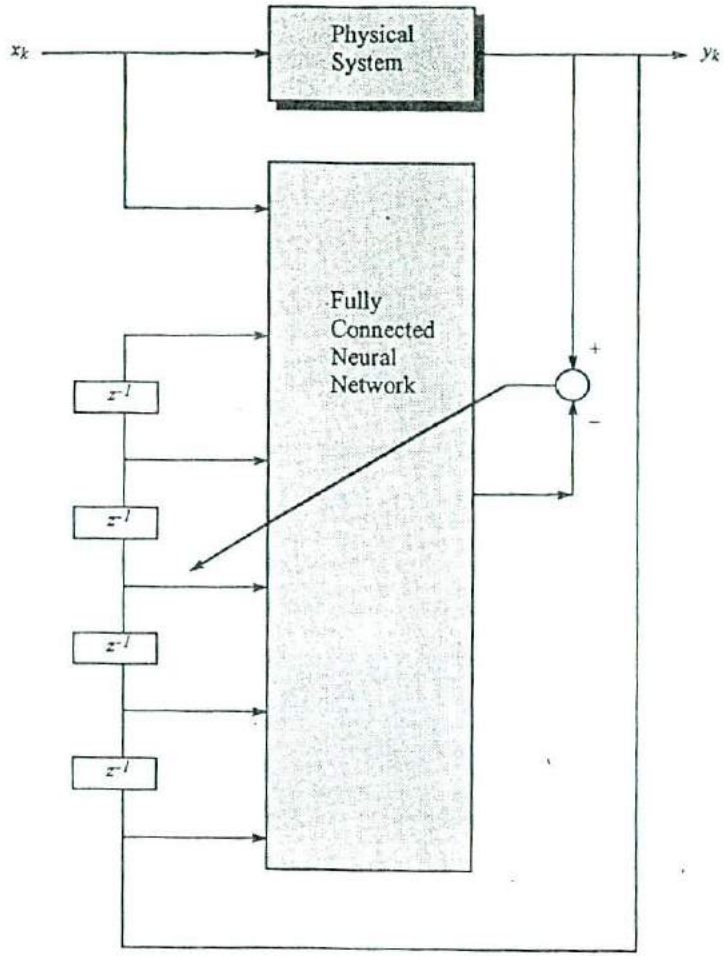


Figure 10.23 Recurrent series-parallel identification model with time-delayed recurrent outputs from the physical system.

to solving control problems, and some of his concepts are incorporated into this chapter.

Inverse Modeling

Inverse modeling develops models that predict corresponding process inputs from process outputs. Inverse models are typically developed with steady-state data and used for supervisory control as feedforward controllers. The appro-

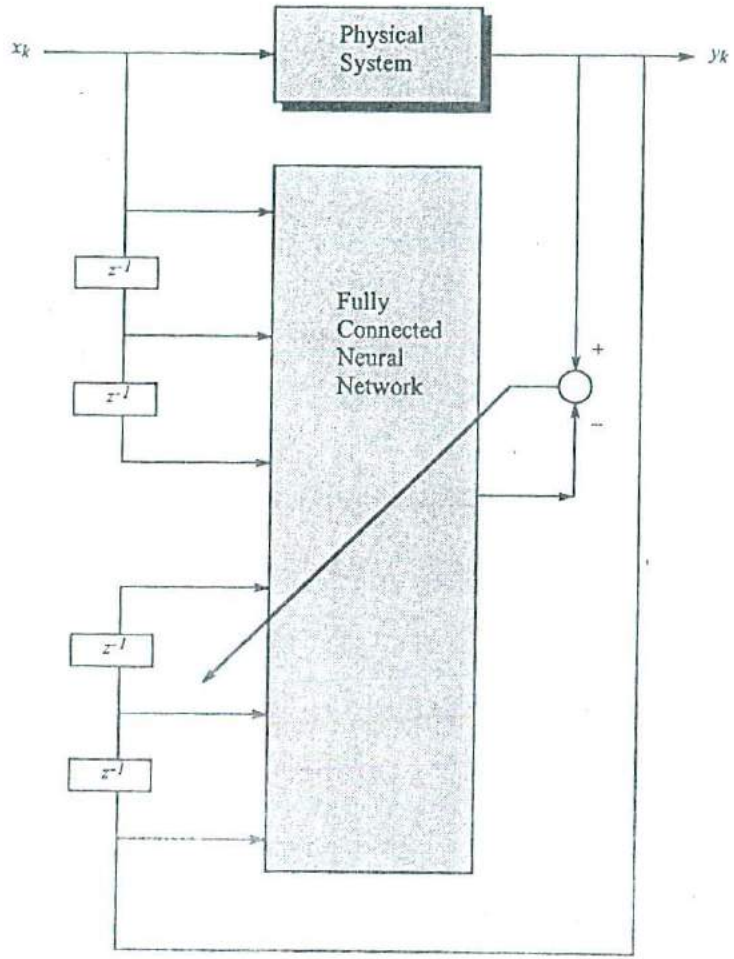


Figure 10.24 General series-parallel identification model with time-delayed direct inputs and time-delayed recurrent outputs from the physical system.

priate steady-state control signal for some setpoint can be determined immediately without the delay associated with the incremental error-correcting operation of feedback control. Neural network inverse models can capture a characteristic source of nonlinearity in many industrial processes (e.g., the variation of process gain with the operating point). Training is readily accomplished since the existing controller output is available.

There are two problems associated with inverse modeling. Many processes have transport delays that imply that any change in the input to a process will

only affect the process response after a "dead time." With the introduction of a transport delay into the inverse modeling process and some experimentation with the length of delay, this problem can be overcome in some cases. The second problem is that mapping from steady-state process output to steady-state process input may be one-to-many. Experience indicates that least-mean-squares averaging behavior of many inverse network function approximation models will in such cases lead to control actions that will likely not be effective.

Controller Autotuning

Controller autotuning estimates appropriate values for controller parameters such as PID gains as shown in Figure 10.13. Although there are many traditional methods used to determine these gain constants, neural networks offer a convenient way of dealing with the nonlinearities involved. A nonparametric neural network process model, once trained, can simulate the closed-loop process and serve as the process simulation. An optimization algorithm can then be used to adjust PID gains, in simulation, until some prespecified cost function or evaluation criterion is minimized. The disadvantage of this approach is the computational complexity, since an iterative algorithm is required and each iteration involves a closed-loop simulation using the neural network process model. However, since the algorithm is not being used for closed-loop control, real-time response is not required.

An alternative is to use a neural network as an autotuner in which the output of the neural network is the PID gains. In training the neural network, the three output gains are compared to precomputed optimal PID gains for a set of training examples. An advantage of this approach is that the network can be trained in simulation (i.e., training on actual process data is not necessary) (Swiniarski, 1990; Ruano et al., 1992).

Adaptive Control Systems

The control of a physical system is usually accomplished by a controller that takes information from the outside world and, in the case of a closed-loop system (a system with feedback), from the physical system itself. An industrial drying oven is an example of a system with a closed-loop control system where the difference between the oven temperature and the setpoint is the error signal that provides a basis for changing the power to the oven. If everything remains constant, such a closed-loop system can produce a very uniform product. On the other hand, if the electrical voltage varies from 210 to 250 volts and/or the thermal capacity of the products moving through the oven changes, the end product may not be very uniform. In other situations, some sort of unforeseen (usually nonlinear) temperature dependence may exist or develop with the aging of equipment. Adjustment is needed to take into account the unforeseen variations in the physical system being controlled

that take place, based on quality of the product produced. A system capable of adapting to unforeseen changes is an adaptive system. The adaptive processor shown in Figure 10.6 is itself a form of adaptive system. It has a performance calculator capable of providing a quality index based on the input, output, and any other variables available. It has an algorithm by which adjustments can be made, and it has a means of making those adjustments. As utilized in the applications described above, it operates as an autonomous unit to make the desired adjustments. Its functions can also be integrated into a more sophisticated control system to carry out adaptive control of a complex physical system.

The most logical approach to using adaptive control is to place an adaptive controller ahead of the physical system and utilize feedback and other inputs to evaluate the performance. One problem with this "direct adaptive" approach is that the system needs operating data to establish a basis for adaption. Suddenly introducing such a control system without adequate historical data to reach a near normal set of conditions could be disruptive. One approach is to use a conventional PID controller and connect the adaptive system in parallel until it learns to match the behavior of the PID system. A similar approach is to manually control the physical system with feedback through human observation while the adaptive controller accumulates the historical information needed to perform its tasks of performance evaluation.

Adaptive Model Control

In adaptive model control, an external model of the physical system that is not part of the control system is obtained using system identification as discussed in Section 10.5. Then the model is used to determine control inputs to the plant, which will produce the desired system outputs. (Manual control can also be used as a model in this approach.) When we apply these same control inputs to the actual physical system, the system output closely matches the desired output. While this appears to be open-loop control, the loop is actually closed through the adaptive process.

Adaptive Inverse Model Control

In adaptive inverse control, the unknown physical system can be made to track an input command signal when it is applied to a controller whose system response function approximates the inverse of its system response function. (All the following arrangements can be implemented using artificial neural networks as the adaptive inverse models.) This adaptive inverse model becomes the controller whose output is the input to the physical system. Its weight adjustment is slaved to a second adaptive inverse model of the type shown in Figure 10.8*b*, that is used to minimize the error between the plant

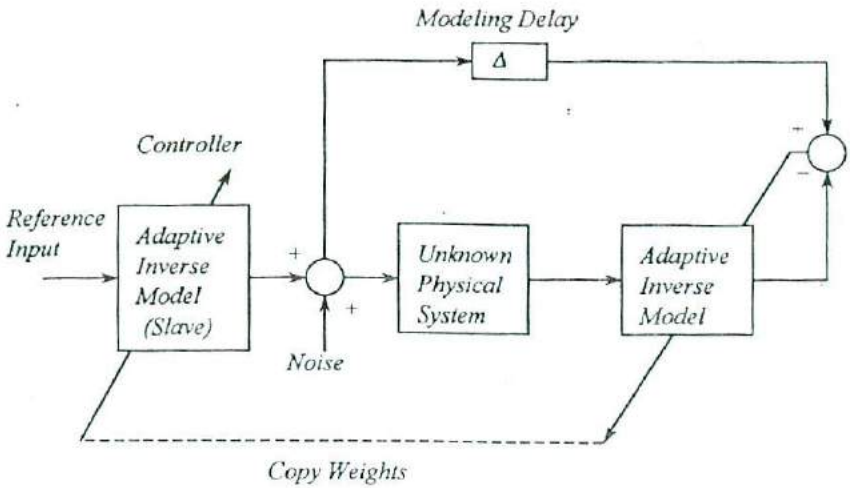


Figure 10.25 Adaptive inverse model control system.

output and the input setpoint. This arrangement is shown in Figure 10.25. An input perturbation or noise is deliberately introduced to ensure that the adaptive process occurs continuously. The introduction of the pseudorandom binary-type noise discussed in Section 10.4 can accomplish this task while at the same time offering the opportunity for cross-correlation and cross-spectral density measurements that could be useful either in the control process, in the identification of process parameters, or the reference model discussed in the next section.

Model Reference Adaptive Control

The configuration of Figure 10.25 can be modified to implement model reference adaptive control. In this concept, a physical system is adapted in such a way that its overall input–output response characteristics best match a reference model response. In this case, the reference model replaces the modeling delay. As a result, the unknown physical system and the adaptive inverse model will be matched to the reference model rather than that of a simple delay.

In some applications, the reference model is a performance model; that is, the performance (efficiency, emissions, etc.) of the system is matched with that of the model rather than matching input–output behavior. The resultant system is usually more complicated than matching input–output response as

described above, and it is usually implemented as a form of reinforcement learning as described in the next section.

Reinforcement Learning

Reinforcement learning addresses the problem of improving performance as evaluated by any index of performance the user chooses. The basis for this approach is that desired control signals exist that lead to optimization, but the learning system is not told what they are because there is no system knowledgeable enough to identify them. In reinforcement, the object is to determine desired changes in the controller output that will increase the index of performance, which is not necessarily defined in terms of the desired outputs of the system.

Reinforcement learning involves two issues: (1) how to construct a critic network capable of evaluating physical system performance consistent with the chosen index of performance and (2) how to alter the controller outputs to improve performance as measured by the critic network. These issues are discussed extensively by Barto (1990). Clearly, there is not one unique approach to these issues. Some approaches attempt to introduce knowledge known about the system to bias the learning process in a favorable direction. Others rely on statistical considerations. For instance, a class of reinforcement-learning algorithms known as *stochastic learning automata* probabilistically select actions from a finite set of possible actions and update action probabilities on the basis of evaluative feedback. It is also possible to combine stochastic learning automata with parameter estimation by mapping pattern inputs to action probabilities. As these parameters are adjusted under the influence of evaluative feedback, action probabilities are adjusted to increase the expected evaluation of the index of performance. Research is continuing in these areas.

Reinforcement learning is a very general approach to learning that can be applied when the knowledge required to apply supervised learning is not available. If sufficient information is available, reinforcement learning can readily handle a specific problem. However, it is usually better to use other methods discussed earlier in this section, because they are more direct and their underlying analytical basis is usually well understood.

10.8 APPLICATIONS OF NEURAL NETWORKS IN NOISE ANALYSIS

The integrated use of neural network and noise analysis technologies offers advantages not available by the use of either technology alone. The application of neural network technology to noise analysis offers an opportunity to expand the scope of problems where noise analysis can be used productively. The two-sensor technique, in which the related responses of two sensors on a

system whose characteristics are unknown responding to an unknown driving source, is used to illustrate such integration.

In the last three decades, vibration analysis has become a separately identified field of noise analysis that is used as a means of detecting faults in dynamic systems, estimating parameters for models of complex systems, and detecting and identifying loose parts in fluid flow systems. Commercial instruments that quantitatively evaluate the power spectra of signals from accelerometers mounted on rotating machinery and interpret the results automatically (based on a model of the system being tested) are readily available. In other cases, spectra must be interpreted by experts because of the complexity of the system and the complex vibration spectra it produces. In these cases, neural networks with their ability to learn characteristics associated with different types and sources of vibration can enhance our ability to interpret the measurements.

Two-Sensor Technique

A specific example will serve to illustrate the symbiotic relationship between vibration analysis and neural networks. The technique described here involves training a neural network to model the internal behavior of a component or system using vibration data taken from two sensors (accelerometers) located at different positions or mounted in different directions on the component or system. The power spectral density (PSD) (typically 128 values) of a sampled time series (typically 100,000 samples which produces 390 spectra) from one accelerometer is used as the input to the neural network, and the PSD of the sampled time series from the other accelerometer taken at the same time is the desired output of the neural network. The network is trained using the 390 pairs of spectra when the component or system is known to be operating properly. The trained neural network is then put into a monitoring mode to predict the output (second) sensor PSD from the input (first) sensor PSD, and a comparison is made between the predicted and actual output signal PSDs using the method described in Figure 10.26. The mean square difference Δ was used as an index of whether operation was normal or deteriorated. Significant deviations indicate that the interrelationship between the input and output signals has been modified due to a change (failure) in the component or system. The usefulness of this methodology has been demonstrated in the monitoring of the operability of check valves (Ikonomopoulos et al., 1992; Uhrig, 1993) and a pump-motor bearing (Loskiewicz-Buczak et al., 1992). These applications are described in Examples 10.3 and 10.4 later in this chapter.

Noise Analysis Considerations

In almost every situation, both sensors measure output vibrations induced by a driving function (e.g., imbalance in a rotating system or turbulence of the

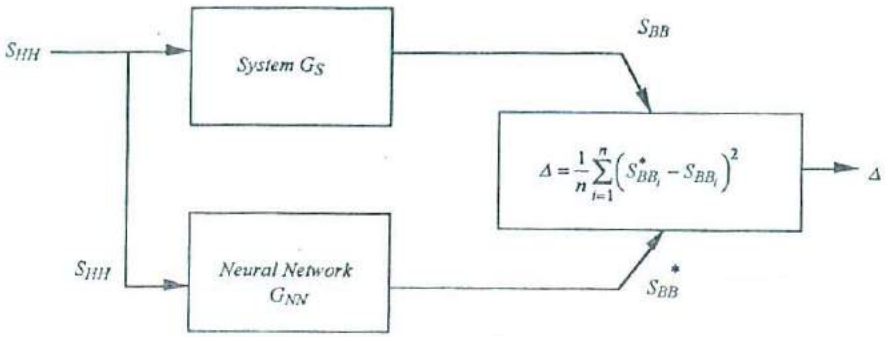


Figure 10.26 Schematic representation of the check-valve testing procedure. $\Delta \approx 0$ if the tested valve is in good condition. $\Delta > 0$ if the tested valve has some degradation. $\Delta \gg 0$ if the tested valve has significant degradation.

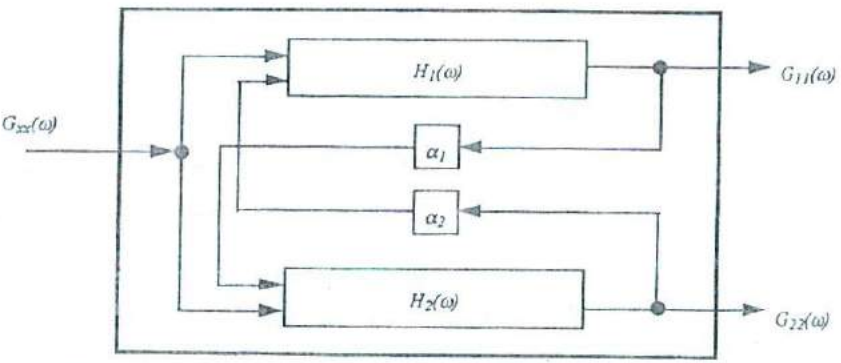


Figure 10.27 Block diagram of a one-signal-input–two-output system.

water as it flows through a pipe or valve) (Uhrig, 1995). This arrangement can be represented in the frequency domain as shown in Figure 10.27, where the PSD of the driving turbulence function is represented by $G_{xx}(\omega)$, the PSDs of the resultant vibration are represented by the output PSDs of the accelerometers $G_{11}(\omega)$ and $G_{22}(\omega)$, α_1 and α_2 are coupling coefficients for the attenuated vibration transmitted between the two sensors, and $H_1(\omega)$ and $H_2(\omega)$ are the system response functions relating the driving turbulence to the resultant outputs of the two accelerometers. Application of the input–output relationships for PSDs (Uhrig, 1970) as given in equation (10.2-9),

$$G_{ii}(\omega) = |H(\omega)|^2 G_{xx}(\omega), \qquad \text{where } i = 1 \text{ or } 2 \qquad (10.8\text{-}1)$$

when applied to the system of Figure 10.27, gives the relationship between $G_{11}(\omega)$ and $G_{22}(\omega)$; after mathematically eliminating $G_{xx}(\omega)$, this becomes

$$G_{22}(\omega) = G_{11}(\omega)\left\{\frac{\left[1 + \alpha_1|H_1(\omega)|^2\right]|H_2(\omega)|^2}{\left[1 + \alpha_2|H_2(\omega)|^2\right]|H_1(\omega)|^2}\right\} \tag{10.8-2}$$

Let us postulate a simple model of this phenomena in which these system response functions $H_i(\omega)$ are assumed to be underdamped, second-order systems (i.e., each system response function has a single peak which may be located at any frequency). The frequency response functions and their square moduli can be represented by

$$H_i(\omega) = \frac{K_i}{\left[1 - \tau_i^2\omega^2\right] + j\lambda_i\omega} \tag{10.8-3}$$

$$|H_i(\omega)|^2 = \frac{K_i^2}{\left[1 - \tau_i^2\omega^2\right]^2 + \left[\lambda_i\omega\right]^2} \tag{10.8-4}$$

where λ_i represents damping constants and τ_i represents time constants associated with the natural frequencies of the systems. Substitution of equation (10.8-4) into equation (10.8-2) gives

$$G_{22}(\omega) = G_{11}(\omega)\left\{\frac{K_2^2\left[\left[1 - (\tau_1\omega)^2\right]^2 + \left[\lambda_1\omega\right]^2 + \alpha_1K_1^2\right]}{K_1^2\left[\left[1 - (\tau_2\omega)^2\right]^2 + \left[\lambda_2\omega\right]^2 + \alpha_2K_2^2\right]}\right\} \tag{10.8-5}$$

We can obtain the general shape of the curve by considering very high and very low frequencies. For very high frequencies, the term in braces in equation (10.8-5) approaches a constant value of $[K_2^2\tau_1^4/K_1^2\tau_2^4]$. For very low frequencies, the term in the braces approach a constant value of $\{K_2^2[1 + \alpha_1K_1^2]/K_1^2[1 + \alpha_2K_2^2]\}$. For mid-range frequencies, the term in the braces is greater than for high or low frequencies. In both $H_1(\omega)$ and $H_2(\omega)$, the amplitudes of their peaks and the frequencies at which the peaks occur are dependent upon λ_i and τ_i, respectively. Hence the peak value of the term in braces and its location are dependent on the values of λ_i and τ_i.

For a more complex model in which the system response functions $H_1(\omega)$ and $H_2(\omega)$ have several peaks, many second-order systems having m and n peaks designated by subscripts i and k, respectively, can be superimposed. Equation (10.8-5) now becomes

$$G_{22}(\omega) = G_{11}(\omega)\left\{\sum_{i=1}^{m}\sum_{k=1}^{n}\frac{K_{2k}^2\left[\left[1 - (\tau_{1i}\omega)^2\right]^2 + \left[\lambda_{1i}\omega\right]^2 + \alpha_{1i}K_{1i}^2\right]}{K_{1i}^2\left[\left[1 - (\tau_{2k}\omega)^2\right]^2 + \left[\lambda_{2k}\omega\right]^2 + \alpha_{2k}K_{2k}^2\right]}\right\} \tag{10.8-6}$$

Again, the values of terms in the braces, which represent the square modulus of the overall system response function for the interior behavior of a complex system, approach constant, but different, values for very low and high frequencies, and there are many peaks over the intermediate frequencies for both $H_1(\omega)$ and $H_2(\omega)$.

This model is consistent with the experimentally measured spectra $G_{11}(\omega)$ and $G_{22}(\omega)$, indicating that the multipeaked representation of the overall system response function of the interior behavior is consistent with the neural network model used in the two-sensor neural network technique for fault identification. Furthermore, this model gives insight into the phenomena modeled by the neural network.

Alternately, the accelerometers can be near each other but mounted so that they measure acceleration in different (usually perpendicular) directions. Under perfect balance conditions for both forces and moments (a rare condition), spectra in different directions at one position would be exactly the same. In almost all real-world conditions, the spectra from sensors located in perpendicular directions are different. Furthermore, the relative shapes of the two spectra change as the systems change or deteriorate.

The diagnosis of faults usually involves measurement and analysis of small fluctuating signals that represent the dynamic behavior of a system or component. Usually this fluctuation is the output signal of an accelerometer measuring vibration or acoustics, but it can be the small random-like fluctuations of a *steady-state* variable (pressure, temperature, etc.). The objectives of such diagnostic work are to identify the existence of abnormalities/deviations and interpreting the results of the monitoring in an intelligent way to identify the fault so that noise specialists and experts are not required for interpretation. While relatively little attention to date has been given to automating these procedures, it is clear that such automation is possible and necessary when this technology is implemented in actual plants and complex industrial/scientific systems. Furthermore, the neural networks can be implemented in microchips to give almost instantaneous outputs.

Typically, measured variables from components or systems are analog variables that must be sampled and normalized to expected peak values before they can be utilized. All the normal precautions associated with digitizing analog data must be exercised to avoid the adverse effects of aliasing and nonstationarity. Often, data must be processed to put them into an acceptable form (e.g., a fast Fourier transformation of the time series to produce a spectrum). In most cases, comparison of predicted results (based on the output(s) of neural network models developed from data taken when the system was working properly) or patterns (learned by neural network models from data presented to it along with actual results or patterns involved) is utilized for fault detection.

Example 10.3 Check-Valve Monitoring. Although there are many possible failure mechanisms for check valves, the most common problems associated

with check-valve failures are due to flow induced system disturbances or system piping vibrations. These vibrations and disturbances induce measurable accelerations that produce check-valve component wear and sometimes component failure. Analysis of time records from piezoelectric accelerometers attached to check valves on a large nuclear power plant has been used to demonstrate this process. The procedure uses an autoassociative-like neural network, in which the inputs and desired outputs are values of the PSDs of two related time series representing vibration at two different positions on the valve. It was trained to produce a neural network model of the interrelationship when the valve is operating properly as described in Section 10.8. During monitoring, the output PSD of one accelerometer is used to predict the output PSD of the other accelerometer, which is then compared with the actual PSD. A significant deviation indicates failure of the check valve. The difference in the two spectra are evaluated numerically using the procedure indicated in Figure 10.26 to evaluate the mean square difference Δ.

Comparison of PSD spectra between identical 30-inch check valves (one broken and one normal), operating under identical conditions, demonstrated that this technique can identify the failed valve. Subsequent measurements taken on the broken valve after it was repaired further confirmed the validity of this technique. Tests on three 6-inch check valves (one normal and two that failed for different reasons) operating under identical test conditions has indicated that different kinds of failures give different values for the mean square difference Δ (Ikonomopoulos et al., 1992). Larger values of Δ indicate more serious problems. □

Example 10.4 Large Motor Pump Bearing Failure. The two-sensor technique was also used to analyze the progressive failure of a large (950 HP) motor pump bearing in a nuclear power plant (Loskiewicz-Buczak et al., 1992). A series of measurements of horizontal and vertical components of acceleration for a large motor-pump bearing were taken periodically at intervals of about 6 weeks throughout the operating lifetime of the bearing and as it began to fail. The power spectra of the horizontal and vertical components of acceleration on the bearing during the first four sets of measurements (when the bearing was known to be operating properly) were the input and desired output, respectively, of a neural network while it was being trained. The bearing operated normally for the next three months and then began to fail. For the next five sets of measurements, while the bearing progressed toward failure, the predicted value of the vertical component of acceleration (obtained from the neural network with the horizontal component as the input) was compared with the actual value of the vertical component. The mean square difference Δ (described in Section 10.7) between the predicted and actual vertical spectra grew as the bearing progressed toward failure.

It can be speculated that it might be possible to predict remaining life in bearings. The integrated use of neural networks and noise (vibration) analysis

has been shown to perform satisfactorily in monitoring the operability of check valves in power plants and in a large motor-pump bearing. There are many other applications in neural networks to the noise analysis field where the two technologies can be advantageously used together. □

10.9 TIME-SERIES PREDICTION

Neural networks can be used to predict future values in a time series based on current and historical values. Such predictions are, in a sense, a form of inference measurements discussed earlier. They are particularly useful to economists, meteorologists, and planners. Recently, there has been an extraordinary amount of interest in the use of neural networks to predict the stock market behavior, including the publication of at least two commercial periodicals specializing in financial investing (*Artificial Intelligence in Finance*, published by Miller-Freeman Publishing Co. since 1995, and *NeuroVe$t*, published by R. B. Caldwell since 1993). This popularity continues in spite of the fact that the predictions by neural networks cannot be explained. Perhaps the main reasons for the continuing popularity in the field are that neural networks do not require a system model and that they are relatively insensitive to unusual data patterns.

Although backpropagation neural networks are usually used for time-series prediction, it is possible to use any neural network capable of mapping an input vector into an output vector. Typically, the input of a single time series into a neural network is made as shown in Figure 10.28. The fluctuating variable is sampled at an appropriate rate to avoid aliasing, and sequential samples are introduced into the input layer in a manner similar to that used in a transverse filter. At every time increment, a new sample value is introduced into the rightmost input neuron, and a sample value in the leftmost input neuron is discarded. The main difference compared to the transverse filter is that the sample preceding those going to the input is introduced into the single output neuron as the desired output. In this way, the network will be trained to predict the value of the time series one time increment ahead based on the previously sampled values. The network can be trained to predict more than one time increment ahead, but the accuracy of the prediction decreases when predictions are further into the future. Since such systems are often used in real time, or to secure data from historic records, the amount of training data is usually very large. Even so, it is important to periodically check the training to ensure that overtraining does not occur.

Although it is possible to predict multiple outputs, it is best to predict only one value because the network minimizes the square error with respect to all neurons in the output layer. Minimizing square error with respect to a single output gives a more precise result. If multiple time predictions are needed, individual networks should be used for each prediction.

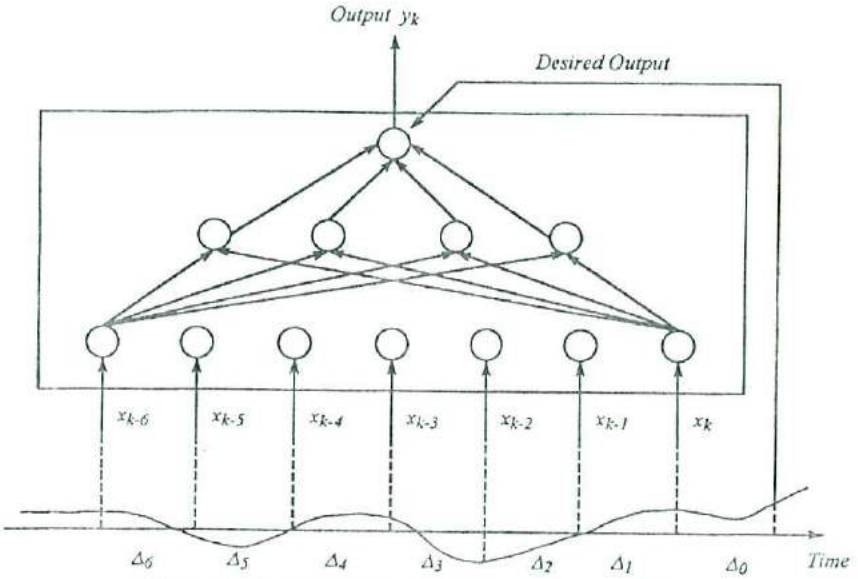


Figure 10.28 Neural network for time-series prediction.

Generally, large-scale deterministic components, such as trends and seasonal variations, should be eliminated from inputs. The reason is that the network will attempt to learn the trend and use it in the prediction. This may be appropriate if the number of input neurons is sufficient for input data to span a complete cycle (e.g., an annual cycle). If trends are important, they can be removed and then added back in later. This allows the network to concentrate on the important details necessary for an accurate prediction.

The standard method of removing a trend is to use a least-squares fit of the data to a straight line, although nonlinear fitting may be appropriate in some cases (e.g., cyclic fluctuations). An alternate method of removing trends and seasonal variations is to pass the data through a high-pass filter with a low cutoff frequency. There are alternative techniques in which a low-pass filter is used to leave only the slowly varying trend which then is subtracted from the original signal, with the difference being the value sent to the neural network input layer.

One of the interesting variations of the above technique for prediction is to use differences between successive sample values as inputs to the neural network. This effectively eliminates constant trends and slowly changing trends by converting them to a constant offset. Even seasonal trends are usually removed. Using differences in predicting is generally useful in such

fields as stock price predictions, especially if the difference is scaled relative to the total price of the stock, which is effectively using the percent price change.

REFERENCES

Andersen, K., Barnett, R. J., Springfield, J. F., Cook, G. E., Bjorgvinsson, J. B., and Strass, A. M., A Neural Network Model for the Gas Tungsten Arc Welding Process, *NASA Tech Brief MFS-26209*, George C. Marshall Space Flight Center, AL, 1991.

Barto, A. G., Connectionist Learning for Control, in *Neural Networks for Control*, W. T. Miller III, R. S. Sutton, and P. J. Werbos, eds. MIT Press, Cambridge, MA, 1990, Chapter 1.

Caldwell, R. B., Publisher, *NeuroVe$t*, Haymarket, VA, published since 1993.

Dong, D., and McAvoy, T. J., Sensor Data Analysis Using Autoassociative Neural Networks, *Proceedings of the World Congress on Neural Networks*, Vol, 1, San Diego, CA, 1994, pp. 161–166.

Foslien, W., Konar, A. F., and Samad, T., Optimization with Neural Memory for System Identification, in *Applications of Artificial Neural Networks III, Proceedings of the SPIE*, Vol. 1709, S. K. Rogers, ed., Bellingham, WA, 1992.

Guo, Z., and Uhrig, R. E., Use of Genetic Algorithms to Select Inputs for Neural Networks, in *Proceedings of Special Workshop on COGANN (Combination of Genetic Algorithms and Neural Networks)*, International Joint IEEE-INNS Conference on Neural Networks (IJCNN), Baltimore, MD, June 6, 1992.

Gyftopoulos, E. P., and Hooker, R. J., Signals for Transfer Measurements in Nonlinear Systems, in *Noise Analysis in Nuclear Reactor Systems*, R. E. Uhrig, ed., AEC Symposium Series #4 (TID-7679), June 1964, pp. 335–345.

Ikonomopoulos, A., Tsoukalas, L. H., and Uhrig, R. E., Use of Neural Networks to Monitor Power Plant Components, in *Proceedings of the American Power Conference*, Chicago, IL, April 13–15, 1992.

Lee, Y. W., *Statistical Theory of Communication*, John Wiley & Sons, New York, 1960.

Loskiewicz-Buczak, A., Alguindigue, I. E., and Uhrig, R. E., Vibration Monitoring Using Sensor Readings Predicted by a Neural Network, in *Proceedings of EPRI's 5th Predictive Maintenance Conference*, Knoxville, TN, September 21–23, 1992.

Miller, W. T., III, Sutton, R. S., and Werbos, P. J., eds., *Neural Networks for Control*, MIT Press, Cambridge, MA, 1990.

Miller-Freeman Publishing Co., *Artificial Intelligence in Finance*, San Francisco, CA, published since 1995.

Narendra, K. S., Adaptive Control of Dynamical Systems Using Neural Networks, in *Handbook of Intelligent Control*, D. A. White and D. A. Sofge, eds., Van Nostrand Reinhold, 1992.

Narendra, K. S., and Parthasarathy, K., Identification and Control of Dynamical Systems Using Neural Networks, *IEEE Transactions on Neural Networks*, Vol. 1, pp. 4–27, 1990.

Nguyen, D. H., and Widrow, B., Neural Network for Self-Learning Control Systems, *IEEE Control Systems Magazine*, pp. 18–23, April 1990.

Pao, Y.-H., *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley, Reading, MA, 1989.

Piovoso, M. J., et al., Neural Network Process Control, in *Proceedings of Analysis of Artificial Neural Networks Applications Conference*, 1991.

Ruano, A. E. B., Fleming, P. J., and Jones, D. I., Connectionist Approach to PID Tuning, *IEEE Proceedings-D*, Vol. 139, pp. 29–285, 1992.

Samad, T., Neurocontrol: Concepts and Practical Applications, in *Intelligent Control Systems: Theory and Practice*, M. M. Gupta and N. K. Sinha, eds., IEEE Press, New York, 1993.

Swiniarski, R. W., Novel Neural Network Based Self-Tuning PID Controller Which Uses Pattern Recognition Technique, in *Proceedings of the American Controls Conference*, 1990, pp. 3023–3024.

Tsung, F. S., Learning in Recurrent Finite Difference Networks, in *Connectionist Models: Proceeding of the 1990 Summer School*, D. E. Touretzky et al., eds., Morgan Kaufmann, San Mateo, CA, 1991.

Uhrig, R. E., Tsoukalas, L. H., Ikonomopoulos, A., Essawy, M., Black, C., and Yancey, S., Using Neural Networks to Monitor the Operability of Check Valves, in *Proceedings of the Conference on Expert System Applications for the Electric Power Industry*, Phoenix, AZ, December 8–10, 1993.

Uhrig, R. E., Integrating Neural Network Technology and Noise Analysis, special edition of *Progress in Nuclear Energy* on IMORN-25 Meeting on Reactor Noise, June 13–15, 1994, published May 1995.

Uhrig, R. E., *Random Noise Techniques in Nuclear Reactor Systems*, Ronald Press (now part of Prentice-Hall), New York, 1970.

Werbos, P. J., Overview of Design and Capabilities, in *Neural Networks for Control*, W. T. Miller III, R. S. Sutton, and P. J. Werbos, eds., MIT Press, Cambridge, MA, 1990, Chapter 2.

Werbos, P. J., Neurocontrol and Supervised Learning; An Overview and Evaluation, in *Handbook of Intelligent Control*, D. A. White and D. A. Sofge, eds., Van Nostrand Reinhold, New York, 1992, Chapter 3.

White, D. A. and Sofge, D. A., eds., *Handbook of Intelligent Control*, Van Nostrand Reinhold, New York, 1992.

Widrow, B., and Hoff, M. E., Adaptive Switching Circuits, 1960 IRE WESCON Convention Record, 96-104, New York, 1960.

Widrow, B., and Stearns, S. D., *Adaptive Signal Processing*, Prentice-Hall, Englewood Cliffs, NJ, 1985.

PROBLEMS

1. Show how Equations 10.1-10 and 10.1-11 are obtained from Equations 10.1-8 and 10.1-9, respectively, for white noise inputs.
2. For the system in Figure 10.2, add a white noise $q(t)$ to the input. Derive the equations for the power spectral density of the output and the

cross-spectral density between the input and the output comparable to Equations 10.1-23 and 10.1-26.

3. In Figure 10.11*b* the control system is located after the actual system and controls by feedback whereas the reverse arrangement is given in Figure 10.12. Discuss the merits of the two arrangements.

4. The difference between the systems in Figures 10.22 and 10.23 is the source of the signal for the recurrent feedback (i.e., from the neural network output in Figure 10.22 and from the system output in Figure 10.23). If the neural network were perfectly trained, these two signals would be identical. Discuss the difference that the source of this signal makes and explain why there is a difference.

5. In time-series prediction, the quality of the predicted signal deteriorates as the time increment into the future increases. Discuss how the quality (by whatever criterion you choose) decreases with future time. How do you determine a practical limit? (*Note*: This problem is also discussed in the context of fuzzy control in Section 15.4.)

11

PRACTICAL ASPECTS OF USING NEURAL NETWORKS

11.1 SELECTION OF NEURAL NETWORKS FOR SOLUTION TO A PROBLEM

Perhaps the best approach to determine whether an application of neural networks is appropriate is to compare its characteristics to those that have been successful in other application. Bailey and Thompson (1990a) have cited a survey of successful neural-network applications developers and gave the following heuristics for successful applications:

- Conventional computer technology is inadequate.
- Problem requires qualitative or complex quantitative reasoning.
- Solution is derived from highly interdependent parameters that have no precise quantification.
- The phenomena involved depend upon multiple-interacting parameters.
- Data are readily available but are multivariate and intrinsically noisy or error-prone.
- There is a great deal of data from specific examples available for modeling the system.
- Some of the data may be erroneous or missing.
- The phenomena involved are so complex that other approaches are not useful, too complicated, or too expensive.
- Project development time is short, but sufficient network training time is available.

Most successful applications of neural networks involve pattern recognition, statistical mapping, or modeling. Successful applications can include signal

validation, process monitoring, diagnostics, signal and information processing, and control of complex (often nonlinear) systems. However, problems that can be solved using conventional computer methodologies, especially those that require high precision or involve mathematical rigor, are usually not appropriate for an artificial neural network approach.

Choice of Neural Network Type

The appropriate choice of the type of neural network (supervised, unsupervised, or reinforced) depends on data available. Supervised learning requires pairs of data consisting of input patterns and the correct outputs, which are sometimes difficult to obtain. Unsupervised training classifies input patterns internally and does not need expected results. The data requirements for unsupervised training are thus much easier and less costly to meet, but the capability of the network is significantly less than for supervised learning. A compromise between supervised and unsupervised training is reinforcement learning, which requires an input and only a grade or reward signal as the desired output.

Time required for both the training and recall are also important in the development of neural networks. Most neural networks have relatively long training times, but the recall involves only a single pass through the network. When the neural network is implemented in hardware with the neurons operating in parallel, the recall time is virtually instantaneous. On the other hand, certain paradigms, such as the probabilistic neural network, radial basis function, and general regression neural network, train in a single pass through the network, but the execution time is essentially the same as the training time. Hence, the need to meet on-line requirements (e.g., in an active control system) may dictate the type of neural network used, or it may require that the network be implemented in hardware.

11.2 DESIGN OF THE NEURAL NETWORK

Size of Neural Networks

Neural network size is sometimes related to the experience of the user as much as the nature of the problem. Beginners tend to stick with small networks and reduce the size of the application accordingly. Those with considerable experience with neural networks are usually willing to let the nature of the problem decide the size of the network. With the neural network simulation software available for personal computers and workstations today, a neural network with a thousand neurons and perhaps a hundred thousand connections may no longer be a practical upper limit for nonstatistical paradigms such as backpropagation or counterpropagation.

Choice of Output

The type of output is usually determined by the nature of the application. The activation of the output neurons may be either binary or gray scale (many individual values). Real-number outputs translate into values such as dollars, time units, or distances and may be given in binary form or gray scale. Each of the four common interpretations of neural network outputs—classifications, patterns, real numbers, and optimal choice—has its own specific requirements. For example, since classifications statistically map input patterns into discrete categories, there will usually be two or more output neurons with only one having an output for a given input. In contrast, neural networks that identify patterns such as spectra often have multiple output neurons, all usually active at the same time, which form a pattern in response to the input. Optimization problems usually yield a special pattern that can be interpreted as a set of decisions (Bailey and Thompson, 1990b).

Neuron Activation function

Typically, the activation function is a continuous function that increases monotonically between a lower limit and an upper limit (0 and 1 or -1 and $+1$) as the weighted summation increases in magnitude. Since one of the primary purposes of the activation function is to keep the outputs of the neurons within reasonable limits, it is sometimes called a "squashing" function. By far the most common activation function is the logistic function discussed in Chapter 7, but virtually any function meeting the sigmoidal requirements stated in Chapter 8 will work satisfactorily. Step or signum (threshold) functions are often used for the activation function when the inputs and outputs are binary (0 and 1) or bipolar (-1 and $+1$).

Activation functions that have been used include linear, clamped linear, step, signum, sigmoid, arctangent, and hyperbolic tangent functions. The choice is usually based on both the types of input and output and the learning algorithm to be used. Certain paradigms such as backpropagation require that the derivative of the activation function be continuous, which eliminates step, signum, and clamped linear functions. Many binary (0 and 1) and bipolar (-1 to $+1$) input–output pairs use networks with step and signum functions, respectively, for the activation function. Continuous valued outputs use linear or sigmoidal (or other S-shaped) activation functions.

Number of Layers

Backpropagation networks typically have three layers, but more may be advantageous under some circumstances. It is sometimes better to use two smaller hidden layers rather than one bigger layer. Some neural network paradigms commonly used have a predetermined number of layers. Adalines,

Madalines, Hopfield networks, ART-1, Kohonen self-organizing feature maps, and bidirectional associative memories all require either one or two layers.

Hidden layers act as layers of synthesis, extracting features from inputs. Usually a larger number of hidden layers increases the processing power of the neural network but requires significantly more time for training and a larger number of training examples to train the network properly. As indicated in Chapter 7, one hidden layer (i.e., a three-layer network) with sufficient neurons is capable (theoretically) of representing any mapping. Additional hidden layers should be added only when a single hidden layer has been found to be inadequate. Cascade correlation neural networks start with two layers and add as many one-neuron layers as necessary to satisfy its convergence criterion.

Number of Neurons in Each Layer

The number of neurons in the input and output layers are determined by the nature of the problem. For instance, a problem that utilizes a 128-point power spectral density function as an input and classifies the inputs into 10 categories requires 128 neurons in the input layer and 10 neurons in the output layer. Determining the proper number of neurons for the hidden layer is often accomplished through experimentation. Too few neurons in the hidden layer prevent it from correctly mapping inputs to outputs, while too many impede generalization and increase training time. Too many neurons may allow the network to "memorize" the patterns presented to it without extracting the pertinent features for generalization. Thus, when presented with new patterns, the network is unable to process them properly, because it has not discovered the underlying principles of the system.

For a network with a single hidden layer, it is common practice to initially make the number of neurons equal to about two-thirds of the number in the input layer (Bailey and Thompson, 1990a). When there is more than one hidden layer, the number of neurons is significantly smaller in each hidden layer. Experimentation with greater and smaller numbers of neurons in the hidden layer(s) may change the training time as well as the ability of the neural network to generalize. Often there is a wide range in the number of neurons in the hidden layer that can be used successfully. Harp, Samad and Guha (1989) has utilized an optimization methodology for determining the optimal number of neurons in a single hidden layer of a neural network based on a genetic algorithm optimization process.

Multiple Parallel Slabs

Another method of increasing a neural network's processing power is to add multiple slabs within a single hidden layer. A multiple parallel slab arrangement may use different types of activation functions and different numbers of

neurons, because this architecture is attempting to force each slab to extract different features simultaneously.

11.3 DATA SOURCES AND PROCESSING FOR NEURAL NETWORKS

A successful neural network requires that the training data set and training procedure be appropriate to the problem. This includes making the training data set representative of the kinds of patterns the operational network will have to recognize. Furthermore, the training set must span the total range of input patterns sufficiently well so that the trained network can generalize about the data. In order to have extrapolation and interpolation capabilities, neural networks must be trained on a wide enough set of input data to generalize from their training sets. Although most of what is presented here deals with neural networks using the backpropagation training paradigm, much of what is said applies to other, less common neural network paradigms as well.

All data that in any way relate to the application should be reviewed and purged of any data that are considered to be unreliable or impractical for technical or economic reasons. Combining and/or preprocessing data to make it more meaningful can be extremely beneficial. For example, power spectral density functions are much more useful than a time series from sampled time records as inputs to neural networks.

Errors in Databases

Databases are rarely perfect. Hence, a database for 100,000 homes may contain a few with entries such as "200 people in a home or a child aged 1975" Protecting neural networks from such gross errors is essential, because it doesn't take many ridiculous values to distort a neural network's training, especially when the importance of errors is increased by squaring those errors. The use of elementary statistical analysis and time plots of data can help detect such errors.

Subtle data errors that don't involve grossly out-of-range values also occur. Checking the consistency of units will eliminate such errors as one office reporting production in units and another reporting it in dozens. The only way to find errors like this is to remain alert to the possibility of data errors and investigate any suspicions that develop about the sensibleness of the data. Clustering data can often help identify discrepancies. If erratic data cannot be fixed, the impact of discarding the data should be investigated. For instance, discarding sales for the month of December when Christmas sales are very large, even when it is necessary because of erroneous data, can distort the results of an analysis.

Incomplete Data Sets

In spite of the rhetoric that neural networks can work with incomplete data sets, missing data can create serious problems. If the data cannot be found (missing data often has merely been placed in the wrong field or misnamed), the common sense (and technically correct) thing to do is to replace every missing value with the best estimate of what it would have been were it not missing. (Crooks 1992) suggests several ways in which this can be done. The simplest method is to replace missing values with the expected value of the variable. If all other variations in the example are ignored, the expected value of a real-valued variable is the mean of the variable across the sample of cases. If the variable is an arbitrary categorization, the most common value or mode is appropriate. For ordinal values, the median value for the population is the expected value. More sophisticated methods are available, but they invariably involve the assumption that the process is time stationary and that the underlying conditions do not change during the time the missing data are important. Whether this is true is dependent upon the individual situation. The pragmatic approach of using whatever technique seems to make the model train and predict better is usually best, but there is need to ensure that future predictions have some general and reliable basis by using an appropriate test data set.

Time Variations in Data

A neural network can detect trends in time-oriented data such as sales data. Although recurrent neural network models have some sustaining memory of previous data, most networks consider only one example at a time. Since there is no explicit memory of the example from an earlier time, it is not possible to simply present data in a sequential order (i.e., first Monday's data, then Tuesday's data, and so on) and expect it to find the trend. Time-oriented problems such as predicting tomorrow's sales requires that the sales for the last week or two (or some appropriate time period) be utilized. A productive approach that is often used in training is to present input training that spans several time periods (days, weeks, hours, months, etc.) and use data for the next time period as the desired output.

Pictorial Data

When the information is pictorial, the data for the neural network are best suited to a nondistributed representation. For example, in a black-and-white picture, each input neuron receives a number representing the intensity of one pixel (picture element) of the visual field for every point in the picture, as well as parameters indicating the location of the pixel. The biggest problem with pictures is that too many pixels are needed to train a neural network in a reasonable amount of time. If a camera image is 1024 pixels on

a side, the number of necessary input neurons is more than a million, and the training time would probably be prohibitive. One approach for feature extraction uses Fourier descriptors of the items to be recognized and feeds them into a neural network for recognition and translation into meaningful results. Characters and graphics have frequency magnitude and phase "signatures" that can be recognized by a neural network. The neural network's output must be formatted into an appropriate form for training and recall.

Data Acquisition

Applications requiring sensory data input to the neural network are impeded when information transfer from the equipment is disrupted. Sensors and data acquisition facilities must be thoroughly tested before being used to provide data to a neural network. Indeed, the influence of sensors and data acquisition facilities on the overall information processing systems needs to be evaluated prior to including them in the system. Situations sometimes arise in which one missing piece of information disrupts the flow (timing) of the system and causes undesirable, sometimes unpredictable, and almost always erroneous results.

11.4 DATA REPRESENTATION

Data may have to be converted into another form to be meaningful to a neural network. How data are represented and/or translated also plays an important role in the network's ability to grasp the problem, that is, a network can learn more easily from some representations than from others. Certain kinds of data (e.g., the time-oriented data used in such problems as forecasting) are especially difficult to handle.

Continuous Valued Versus Binary Representations of Data

Data may be continuous-valued or binary. Sometimes data can be represented either as a single continuous value or as a set of ranges that are assigned binary representations. For instance, temperature could be represented by the actual temperature values or as one of five possible values: frozen, chilled, mild, warm, or hot. When there are naturally occurring groups, the binary categories are often the best method for making correlations. When the values are very continuous, artificially breaking them up into groups can be a mistake, because it is often difficult for the network to learn examples that have values on or near the border between two groups.

Arbitrary Numerical Codes

Using continuous-valued inputs to represent unique concepts can cause problems. Although it may seem perfectly reasonable to represent the months

of the year as numbers from 1 to 12, the neural network will presume such data to be continuous-valued and as having "more or less" or "better or worse" qualities. Since the month July (represented by 7) is not more or better than June (represented by 6), individual inputs are required for each month. Discontinuities such as going from 12 for December to 1 for January are also troublesome. Zip codes, bar codes, and marital status are examples of data that require more than one input (Lawrence, 1991).

Variable values represented with numbers don't always behave like numbers, because they sometimes don't reflect any specific sequence or order. For example, although there is some obscure plan behind postal zip codes, it is not possible to add, subtract, or compare zip codes and infer meaningful results. (Generally, a larger zip code indicates the post office is further west in the United States, but there is certainly no sensible interpretation of the sum of two zip codes.) Arbitrary numerical codes should be treated in the same way as mutually exclusive nonnumeric codes (like male/female or apple/orange/pear/banana), that is, assign one input neuron for every possible value. In any single case, only one of the neurons should be set to one, and all of those representing other values should be set to zero. If a categorization has too many possible values, like the states of the United States, it may be necessary to combine some of the categories to produce a taxonomy of fewer values (i.e., combining states into Northwest, Southwest, Northeast, Southeast, and Midwest categories).

"Fairly Continuous" Data

Lawrence (1991) points out that the choice between binary and continuous data representations may not be simple. If the data are fairly continuous but not evenly distributed over the entire range, even the best representation can be tricky. For example, a network that predicts the income level of individuals based upon demographics and personal history might have inputs for the person's education level with values from 0 to 20 years. Alternately, natural groupings occur around traditionally recognized levels of achievement (i.e., high school, baccalaureate, masters, and doctoral graduations), and the data could be grouped into ranges such as less than 13, 13–16, 17–18, and 19–20 years. However, if significant differences occur within a group (e.g., "less than 13" could mean either high-school graduate or grade-school dropout), the representation may not be valid. On the other hand, if one continuous-valued input representing the actual number of years of schooling is used, the neural network might have trouble. For instance, if there is a significant difference in the effect on monetary savings between having a high-school diploma and not having one, the network may not pick it up, because 12 and 11 look very similar in the range 0–20. The best representation may be a combination of the two approaches (e.g., several groups, each continuous-valued). Experimentation with several representations may be necessary to determine the best representation.

Ordering of Variables

At the opposite extreme from nominal numeric values are real or continuous values. Most measurements of natural quantities result in real numbers. Two real values for the same variable can be compared, and the difference will be meaningful. In this context, it is not continuity that matters as much as the orderings possible for the set of numbers used.

Falling between nominal and continuous are variables whose numeric values imply a real ordering, but with undefined intervals between the values. For example, if all of the soldiers in a platoon were lined up and ranked in order from the shortest to the tallest, you could say that soldier #10 was taller than soldier #5, but you couldn't say by how much, and you certainly couldn't say soldier #10 is twice as tall as soldier #5. Crooks (1992) points out that rankings like this are especially troublesome because the rank value depends not only on the height of a particular soldier, but also on how many and which soldiers are compared.

Rank orderings are often handled by converting a ranking into a percentile (more precisely, a percentile divided by 100 to keep the value under 1.0) to make the values independent of the number of cases in the sample taken. Alternately, all observations can be divided into quintiles representing values from the highest quintile as 0.9 with second quintile values at 0.7, and so on, down to 0.1 for the lowest quintile. The approach is the same, but the number of categories is reduced from 100 to five by the use of quintiles. It is essential that boundaries for percentiles and quintiles be based on a good sample of the population of cases to be modeled. It is more important that percentile and quintile values be reliable and repeatable throughout the training and use of a neural network than that they be accurate.

Changes in Values Versus Absolute Values

Another important factor in representing continuous-valued data is whether to use actual values or changes in values. One reason for using changes in values is that the smaller the range, the more meaningful small-value differences are to the network. However, the range of some data, such as the Dow Jones industrial average (DJIA), will probably change over time. The day-to-day change in the DJIA over a month (with rare exceptions) is not likely to exceed ± 200 points. On the other hand, the change over a year might be 1000 points and maybe 3000 points over a decade. The decision whether to use the absolute value or the change depends upon the nature of the problem. If small changes in the day-to-day values of the DJIA are an important consideration in the problem being investigated, then the change in DJIA should be used. If the trend over a decade or over a few years is important, then the actual values should be used, scaled to spread the values between the expected maximum and minimum values over the operating range of the neural network simulator (see Section 11.5).

Distributed Versus Nondistributed Information

A decision whether to describe the information as unique items (i.e., gender, minority status, etc.) or as a set of descriptive qualities (such as height, age, weight, etc.) is necessary. Information that exclusively categorizes a thing or person into one of several possible categories is called a nondistributed representation. Only one neuron is needed when the choice is between two categories (e.g., male or female), but one neuron is needed for each category when there are more than two alternatives (e.g., minority status; Black, Hispanic, Native American, etc.). Using nondistributed information may increase the size of a neural network with resultant training and generalizing problems.

Distributed information involves using a few pieces of information to define a unique pattern. For example, by using three primary color inputs (red, blue, and yellow), many possible color combinations can be represented without adding neurons. Such a distributed input scheme reduces the number of neurons needed to represent a large number of patterns that share common qualities and enhance the generalization ability as well, but there has to be a means of interpreting the results. However, there are potential problems with using a distributed approach for the output. A network with a distributed output layer also has less learning capacity because it has fewer weights. Such a network output sometimes must be decoded twice: first, from neuron activations to the distributed qualities and then to the nondistributed output. For example, if color were expressed as a distributed output pattern such as 0.2 blue, 0.7 yellow, and 0.4 red, this result would have to be decoded again by some external observer or program to designate the color "brown."

Advantages of a distributed output network are that it uses fewer neurons in the output and the hidden layers, has fewer connections, does less computation, and runs faster. Generally speaking, neural networks with a greater number of inputs than outputs perform better. More outputs make it harder to train the neural network to be accurate. Overall error, rather than the error in individual outputs, is minimized.

Encoding Data

An encoding algorithm's function is to take input data and convert it into a form suitable for presenting to the network. A decoding algorithm takes the values of the output layer neurons and converts them into a meaningful answer. Encoding and decoding algorithms are neural-network-specific, but some guiding principles can be applied. Neurons operate with numeric inputs and outputs that correspond to the activation values of the neurons—that is within the range neurons understand (usually 0 to 1, or -1 to $+1$). The input encoding must interpret the raw data—that is, turn it into a sequence of numeric values that the network can understand. The output decoding

must take a sequence of numbers that corresponds to the output neurons' values and turn them into the form required for the final output.

As an example, consider a three-neuron output with a binary (0,1) output. This neuron output can represent eight categories of output (i.e., 101 represents the fifth category which can be arbitrarily defined). Since outputs are not likely to be exactly 0 or 1, an output in the range 0.8 to 1 could be interpreted as 1 and an output in the range 0 to 0.2 could be interpreted as 0. Values between 0.2 and 0.8 would then represent ambiguous results. Some investigators arbitrarily split the outputs between 0 and 1 at some arbitrary threshold (not necessarily 0.5).

Fourier analysis of waveforms can also be used for the analysis of acoustical waves, vibration, motion, or electrocardiograph records. The frequency content of the digitally recorded waveform is obtained using the fast Fourier transform technique. The value presented to each input neuron represents the amplitude of the signal at a particular frequency range.

11.5 SCALING, NORMALIZATION, AND THE ABSOLUTE MAGNITUDE OF DATA

Neural networks are very sensitive to absolute magnitudes. If one input ranges from 1000 to 1,000,000 and a second one ranges from 0 to 1, fluctuations in the first input will tend to swamp any importance given to the second, even if the second input is much more important in predicting the desired output. To minimize the influence of absolute scale, all inputs to a neural network should be scaled and normalized so that they correspond to roughly the same range of values. Commonly chosen ranges are 0 to 1 or -1 to $+1$.

Even though one of the great strength of neural networks is that they work well in nonlinear situations, linear relationships are the easiest for neural networks to learn and emulate. Therefore, minimizing the effects of nonlinearity of a problem pays off in terms of faster training, a less complicated network, and better overall performance. Hence, one goal of data preparation is to reduce nonlinearity when its character is known and let the network resolve the hidden nonlinearities that are not understood.

Data Normalization

Numeric data must be normalized or scaled if it has a natural range that is different than the network's operating range. Normalization is simply dividing all values of a set by an arbitrary reference value, usually the maximum value. Use of the maximum value will limit the maximum value to unity. This process, although very commonly used, carries with it the potential for loss of information. It can also distort the data if one or a few values are much

larger than the rest of the data (e.g., anomalous spikes) or when all the data are within a narrow band. Scaling, on the other hand, is establishing a linear relationship between two variables over the desired range of each. Normalization is a special case of scaling where the minimum value of both variables is zero.

Data Scaling

Scaling has the advantage of mapping the desired range of a variable (with a range between the minimum and maximum values) to the full "working" range of the network input. For example, let us assume that the values between the minimum and maximum (called the range Δ) must be scaled into the range 0.1 to 0.9 for the neural network input. This linear scaling is shown in Figure 11.1, where the straight line has the form

$$y = mx + b \tag{11.5-1}$$

where m is the slope and b is the y intercept. If we substitute the values

$$y = 0.1 \qquad \text{when } x = x_{\min} \tag{11.5-2}$$

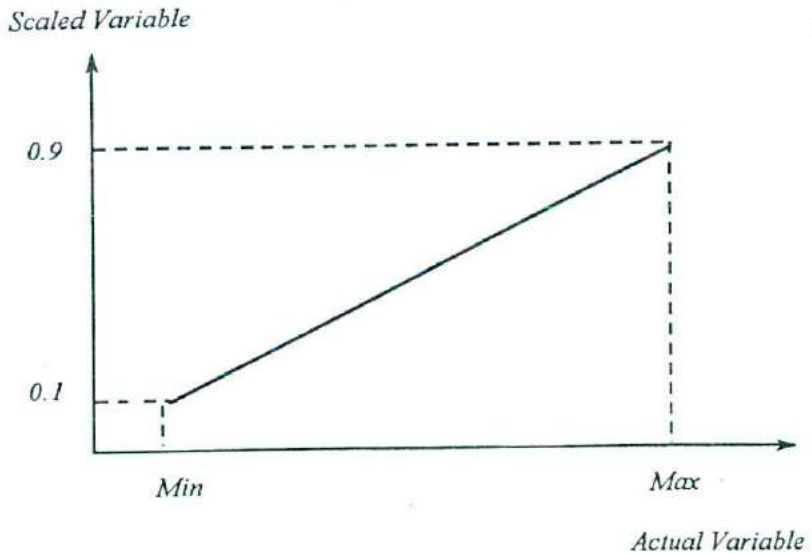


Figure 11.1 Scaling of input variables for artificial neural networks.

and

$$y = 0.9 \qquad \text{when } x = x_{\max} \tag{11.5-3}$$

we can solve for the constants m and b to be

$$m = 0.8/(x_{\max} - x_{\min}) = 0.8/\Delta, \tag{11.5-4}$$

and

$$b = [0.9 - 0.8x_{\max}/\Delta] \tag{11.5-5}$$

where

$$\Delta = x_{\max} - x_{\min} \tag{11.5-6}$$

Equation (11.5-1) then becomes

$$y = (0.8/\Delta)x + (0.9 - 0.8x_{\max}/\Delta) \tag{11.5-7}$$

Scaling of the variable between 0.1 and 0.9 is often used to limit the amount of the sigmoid activation function used in the representation of the variables in order to avoid "network paralysis" in the training process. Many neural network simulation software systems perform such scaling automatically. Even so, it is necessary to understand what is occurring so that unforeseen scaling factors are not inadvertently introduced into the process. For instance, if the input is scaled to its maximum and minimum values and the desired output is scaled to its maximum and minimum values which are different, then the recall output has a scale factor that is the ratio of the two input scale factors. This can be avoided by using a single-scale factor for both input and desired output that is based on the maximum and minimum values that occur in both the input and desired output variables. Most commercial software packages automatically use a single-scale factor unless directed to do otherwise.

Crooks (1992) points out that scaling to similar magnitudes is not always adequate. For instance, if one input variable to a neural network fluctuates from 50 to 1000 and a second input variable changes only from 950 to 1000 (even though it may have been very low in the past), it is clear that the region of typical variation is much different for these two variables, even though they have similar magnitudes and historical ranges. Since networks pay attention not only to the magnitude of inputs but to their variability as well, the greater variability of the first variable would tend to distract the network relative to the smaller, but perhaps more important, variation in the second variable.

Z-Scores

An appropriate approach for some problems is to compensate for variability in the scaling of variables. The common way to do this is to scale inputs to their "Z scores" (the number of standard deviations above or below the mean). To perform Z-score scaling on one variable, first calculate the mean and standard deviation for the variable across all of the examples in the data set. Then convert each example value to a Z score by subtracting the mean and dividing the difference by the standard deviation. This procedure partially compensates for both different magnitudes and variabilities. Z-score scaling does not take away some useful information, but instead, it makes the information independent of units of measure.

Crooks (1992) gives two precautions that must be observed when working with Z scoring. First, the calculated mean and standard deviations for an input variable are merely estimates of the mean and standard deviations for the entire population sampled by the data at hand. Hence, if more than one set of training data is used, the scaling for the two sets may be different because the estimated mean and standard deviations will be different for the different samples. This often presents a practical problem when comparing results from different training sets. The solution is to select one estimate for the population mean and one estimate for the population standard deviation and then use them uniformly to scale all data sets in the same way for the selected variable. Second, if one output value is scaled using the Z-score method, the output neurons must represent values throughout the range of about -3.0 to $+3.0$. Often, output neuron sigmoids prevent outputs greater than $+1.0$ or less than 0.0 (or -1.0), which would make it impossible for the network's output to reach 2.0 (or -2.0). Clearly Z scores are not appropriate for cases where this problem arises and is not addressed properly.

Input Transformations

Some fairly simple input transformations, such as ratios, can save a network a good deal of work. While the network can learn to do the division by itself, networks normally perform division by effectively converting the numerator and denominator each to logs, subtracting them, and taking the antilog. Although these three nonlinear operations are often performed by networks, it is more productive to carry out the division in data preprocessing and let the neural network concentrate on establishing relationships from the data.

Besides ratios, nonlinearity in a problem may be reduced by the use of logarithmic scaling for inputs of an exponential or compounding character or the use of exponential scaling for inputs with a logarithmic pattern. Nonlinear scaling is also used to emphasize a particular range of variables. For instance, logarithmic scaling is often used to compress the scale for larger values whereas exponential scaling is used to expand the scale for smaller values.

When a problem may have a geometrical aspect, precalculating relevant distances, areas, and volumes is helpful. In short, when an input is known to have a specific nonlinear tendency, try to counteract the tendency with scaling that will yield a more linear input to the neural network and simplify its operation.

Redundancy in Input Data from Monitored Variables

Experience has shown that the existence of a high degree of redundancy in the data from the monitored variables of a complex process or system can and usually does have an adverse influence of the results of neural network modeling. Decorrelation of the input variables using ordinary statistical methods (Jurik, 1993) can be quite effective in improving the validity of the model. In effect, the methodology of Jurik typically identifies a few special variables in which a high fraction (typically 95%) of the information is contained. These few special variables y_i, (which are, in fact, principal components) have the form

$$y_i = \sum_{j=1}^{N} \alpha_i x_i \tag{11.5-8}$$

where i is the index for the number of special variables (principal components) used, and j is the index for the N input variables being monitored in the complex system or process. Once the coefficients a_i for these principal components have been determined using the decorrelation code, they can be implemented by an additional input, network ahead of the usual input network with the connecting weights set to equal the values of a_i.

These principal components can also be obtained using autoassociative neural networks as discussed in Section 8.4, where the data are extracted from the "bottleneck" layer. A network of the type shown in Figure 8.14*a* can be used to provide the principal components as inputs, after the complete network has been trained as an autoassociative neural network.

Genetic algorithms have also been used to select the most important variables for a neural network by Guo and Uhrig (1992) (see Example 17.2 in Chapter 17) and later by Harp, Samad and Guha (1989, 1990).

11.6 DATA SELECTION FOR TRAINING AND TESTING[1]

Kinds of Data

All that is needed to train a neural network is an adequate amount of the kind of information that is important in solving a problem. If there is

[1] Many of the suggestions in this section were given by Lawrence (1991).

uncertainty whether specific data are important, it is usually best to include it because a neural network can learn to ignore inputs that have little or nothing to do with the problem, provided that enough examples are provided. Using too much or too many kinds of data is seldom a problem if there is adequate data. If inadequate data are used, correlations become difficult to find. Training time may become excessive when not enough kinds of data exist to make proper associations. This is often the case with backpropagation networks with a very large number of hidden neurons. The end result is memorization of the individual values, and the network trains well but tests poorly on new data.

Difference in Data Requirements in Supervised and Unsupervised Learning

Lawrence (1991) points out that there is a big difference in how data get organized between supervised networks and unsupervised networks. Supervised neural networks are generally used for prediction, evaluation, or generalization. They basically learn to associate one set of input data with the corresponding set of output data, For example, a neural network can associate an increase in agricultural crop yield with certain types of weather patterns; to predict the crop yield, the weather pattern (rainfall, temperature, humidity, cloud cover, etc.), including historical patterns, would be specified as inputs to the neural network. Unsupervised networks, such as Kohonen networks, are best applied to classification or recognition types of problems (e.g., descriptions of diseases can be stored; when a new medical case comes in with a partial description of the symptoms, the Kohonen neural network would look at the description and provide as an output the stored diagnosis that most closely matches one of the descriptions stored in the network).

Generally, the more example sets that are presented to a network for training and testing, the better the training will be. However, there must be enough examples of a sufficient variety for training that the network will be able to make valid correlations and generalizations for unfamiliar cases. The variety must include a good distribution of possible inputs and outputs. Lawrence (1991) cites the following example: If a network is to perform an evaluation such as the operational readiness of an aircraft, examples of good and bad situations should be used in fairly even proportions. However, if 1000 examples of the aircraft being ready and 10 of it not being ready are provided for training, the network will probably not be able to learn those 10 cases. Even if it does learn them, the network may predict that the airplane is ready more often than it should be.

Cases Where Inadequate Data Are Available

There should be sufficient training sets so that a random sampling of data examples can be set aside for testing the neural network. If an inadequate number of training examples are available, creating a data set from simulator

runs or using expert evaluations of situations may be necessary and acceptable. Several experts can rate examples, and a single network might be trained on the sum total of the expert's views. Alternately, a network might be trained for each expert's opinion to see which network gives the best results after training.

For fabricated examples, use of "border" patterns (examples in which the output just begins to be different) can be very effective. Research has shown that the success rate of a trained neural network increases rapidly as the number of border training patterns used increases. A manufactured training set using both border patterns and diverse-valued training patterns is substantially better.

If there are only a few data examples, a technique called "leave one out" can be used to train several networks, each with a different subset of most of the examples. Then each network is tested with a different subset. Leaving a different set of examples out of the training set and subsequently testing on a different set will greatly improve assessing the network's effectiveness and may show where more examples are needed. It should also indicate whether a network trained with all of the examples can generalize well.

Randomly chosen training patterns, although often used, may inadvertently emphasize the wrong conceptual points. Since the most easily identifiable patterns must be included for the network to learn the basics, a randomly chosen training set may not include these basic patterns in the proper proportions (Lawrence, 1991).

Data that cover too long a time span can include changes of equipment or other events that make the process nonstationary or even discontinuous. When the behavior has changed over time, data collection should be limited to a time period of similar behavior. For example, the strongest influences on the value of gold today may not be the same as those before the breakup of the Soviet Union. Adding a neuron indication as to whether the data examples were before or after breakup will solve this problem. When the changes are long term rather than associated with specific events, throwing out the oldest data and adding newly collected examples to a training set can be very helpful.

11.7 TRAINING NEURAL NETWORKS

Backpropagation Training

Backpropagation is a gradient descent system that tries to minimize the mean squared error of the system by moving down the gradient of the error curve. In a real situation, however, the network is not a simple two-dimensional system, and the error curve is not a smooth bowl-shaped figure. Instead, it is usually a highly complex, multidimensional, and more-or-less bowl-shaped curve that can have all kinds of bumps, valleys, and hills that the network must negotiate before finding its lowest point (the minimum mean-squared error position). The number of iterations through the data set required to

achieve a given level of training will generally increase as the size of the training set increases, as the number of layers increases, and as the size of the middle layer increases. In general, the bigger the network, the slower each pass through the training data, when the network is simulated on a serial-type digital computer.

Dealing with Local Minima

Despite backpropagation's widespread use, it is sometimes difficult to use, and training times are often excessive. Caudill (1991a) has offered some tips on training techniques that have been found to be useful. It is important to note that these suggestions are specific to backpropagation networks and may be unsuitable for other paradigms.

Perhaps the "easiest" way to deal with a neural network that is stuck in local minima created by the hills and valleys and will not train is to start over by reinitializing the weights to some new set of small random values. Geometrically, this changes the starting position of the network so that it has a new set of obstacles and traps to negotiate to get to the bottom of the error surface. It is expected (but certainly not guaranteed) that as a result of starting from a new position there will be fewer obstacles in reaching the global minimum of the error surface. The difficulty is that the user must be willing to forego any progress in training and start over on a path that may be no better, or even worse, than the first path.

A less drastic approach is to "shock" the neural network by modifying the weights in some small random or systematic way. Again, it is expected (but not guaranteed) that a small move in the error surface will provide a path to the global minimum. A good rule of thumb is to vary each weight by adding a random number of as much as 10% of the original weight range (e.g., if the weights range from -1 to $+1$, add random values to each weight in the range -0.1 to $+0.1$). Generally, this technique is used when the network has learned most of the patterns before stalling, whereas starting over is used when the network has been unable to learn very few of the patterns. Such changes should be made only after an integral number of epochs have been presented to the network.

Applying the proper amount of momentum to a backpropagation network is probably the single easiest thing to do to make the network train faster. The momentum term helps a backpropagation network keep moving down the error surface, even when it meets a temporary upward surface. In effect, momentum ensures that if the weights were changing so the error decreased last time, there will be a "force" to make the next weight change reduce the error further.

Another effective way to reduce a network's training time is to use slightly noisy data. Oddly enough, networks actually train faster with noisy data. For example, an input that is a matrix-binary representation (0s and 1s) of an alphabetical letter that is being mapped into an ASCII code for the letter will train faster if the "pure" binary representation is corrupted by the addition

of 10–15% random noise. The network never sees two images of the letter that are exactly alike. Hence, this technique forces the network to generalize, a key goal in the training of neural networks.

Monitoring the Training Process

Monitoring the training process includes looking for local minima, overtraining, and network paralysis. Eliminating local minima or overtraining may involve introducing specific or random changes into the weights and often adjusting training parameters (e.g., increasing the momentum or changing the learning and/or activation function constants). If these techniques do not produce results in a reasonable time, it may be necessary to reinitialize the weights and start the training over.

The method of presenting the training set to the network can affect the training results in certain learning algorithms. To mitigate these effects, neural network simulation software often change the order in which the training cases are presented to the network (e.g., present the test cases randomly or in some predetermined order) or delay the adjustment of the weights until an integral number of epochs of training data (or a specific number of data sets) are presented to the network.

After the network has been trained, it is important to test it against both the training set and examples that the network has never encountered before. Increasing the size of the hidden layer usually improves the network's accuracy on the training set, but decreasing the size of the hidden layer generally improves generalization, and hence the performance on new cases. An optimal size can be attained by a balance between the objectives of accuracy and generalization for each particular application. Creating a functioning neural network that provides the most accurate, consistent, and robust model possible requires iterative building, training, and testing to refine the neural network.

Overtraining is probably the most common error in training neural networks. The best method of ensuring that overtraining does not occur is to monitor periodically the sum square error for both the training data and the test data. It is normal for the sum square error for the training data to continue to decrease with training. However, this may be forcing the neural network to fit the noise in the training data. To avoid this problem, stop periodically the training, substitute the test data for one epoch, and record the sum square error. When the sum square error of the test data begins to increase, the training should be stopped. Indeed, if the weights at the previous monitoring are available, they should be used.

Another form of testing uses special inputs to study the neural network's responses (similar to the use of impulse or step functions in testing electrical circuits under specific conditions). Activating (either positively or negatively) an input node and then examining the input–output relationship (e.g., the ratio of the change in a specific output for a given change in a specific input) can give the sensitivity of input–output relationships. If significant problems

are found, the neural network or the training process must be debugged. Every aspect of the network and the training process must be examined, including the quality, representativeness, and accuracy of the training data sets, the constants within the learning algorithms, and especially the normalization and scaling (including denormalization and descaling) processes.

Role of the Hidden Layer in Training

The traditional explanation of the functions of the hidden layer of a well-trained three-layer network is that it views the input pattern to determine which features are present in the pattern, and the output layer considers what output should be generated for the particular combination of features identified by the hidden layer. Unfortunately, the hidden layer may memorize the input patterns rather than learning the features, especially if the number of neurons in the hidden layer exceed the number of training cases. If the network memorizes its response instead of generalizing features, it may give the perfect answer for the training input and have no idea at all what to generate for a test-input pattern. If a single neuron responds to a particular input pattern, it is called a "grandmother" cell. Unfortunately, this concentration of information into a single neuron makes the network, which is usually robust, very vulnerable to the failure of a single neuron. Memorization can be prevented by ensuring that the network never sees exactly the same input pattern more than once. This can be done by adding random noise to the each input pattern.

Setting the number on neurons in the middle layer equal to the number of patterns in the training set can encourage the network to assign one neuron in the middle layer to each training pattern, which obviously does not encourage general feature detection and generalization. Solutions include adding noise to the input or reducing the number of neurons in the hidden layer.

Applying a little noise to the training set will generally produce a network that is robust to noisy inputs. Although a network trained with no noise may still do well with noisy inputs in the real world, one trained with an appropriate level of noise will do much better. The exact type and amount of noise depends on the data, but a general rule is that 10–15% perturbation of the signal is a good starting point.

In general, the exact size of the middle layer isn't a critical parameter, and training times don't vary significantly for similar-sized middle layers. Sometimes increasing the size of the middle layer will provide more feature detectors. When the middle layer is just too small, increasing it by 10% or 20% may make a huge difference. However, too large a middle layer will lengthen the training process, and extra degrees of freedom may allow the neural network to "overtrain" (i.e., the neural network will be trained to the point that it fits the noisy fluctuations in the mapping relationship).

REFERENCES

Anderson, J. A., Data Representation in Neural Networks, *AI Expert*, Vol. 5, No. 6, pp. 30–37, 1990.

Bailey, D., and Thompson, D., How to Develop Neural Networks, *AI Expert*, Vol. 5, No. 6, pp. 38–47, 1990a.

Bailey, D., and Thompson, D., Developing Neural Network Applications, *AI Expert*, Vol. 5, No. 9, pp. 34–41, 1990b.

Caudill, M., Using Neural Nets: Diagnostic Expert Nets, *AI Expert*, Vol. 5, No. 9, pp. 43–47, 1990.

Caudill, M., Neural Network Training Tips and Techniques, *AI Expert*, Vol. pp. 56–61, 1991a.

Caudill, M., Evolutionary Neural Networks, *AI Expert*, pp. 28–33, March 1991.

Coleman K. G., and Watenpool, S., Neural Networks in Knowledge Acquisition, *AI Expert*, Vol. 7, No. 1, pp. 36–39, 1992.

Crooks, T., Care and Feeding of Neural Networks, *AI Expert*, Vol. 7, No. 7, pp. 36–41, 1992.

Guo, Z., and Uhrig, R. E., Using Modular Neural Networks to Monitor Accident Conditions in Nuclear Power Plants, *Proceedings of the S.P.I.E. Technical Symposium on "Intelligent Information Systems," Applications of Artificial Networks III*, Orlando, FL, April 20–24, 1996.

Harp, S., Samad, T., and Guha, A., Towards the Genetic Synthesis of Neural Networks, in *Proceedings of the Third International Conference on Genetic Algorithms*, Morgan Kaufmann, San Mateo, CA, 1989.

Harp S., Samad T., and Guha A., Designing Application Specific Neural Networks Using the Genetic Algorithm, in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, ed., Morgan Kaufmann, San Mateo, CA, 1990.

Jurik, M., *User's Manual for Decorrelator Computer Software*, Jurik Associates, Los Angeles, CA, 1993.

Klimasauskas, C. C., Applying Neural Networks, Part 2: A Walk Through the Application Process, *PC AI*, pp. 27–34, March/April, 1991.

Lawrence, J., Data Preparation for a Neural Network, *AI Expert*, Vol. 6, No. 11, pp. 34–41, 1991.

Pao, Y., Functional Link Nets: Removing Hidden Layers, *AI Expert*, Vol. 4, No. 4, pp, 60–68, 1989.

Tveter, D. R., Getting a Fast Break with Backprop, *AI Expert*, Vol. 6, No. 7, pp. 36–43, 1991.

III

INTEGRATED NEURAL–FUZZY TECHNOLOGY

12

FUZZY METHODS IN NEURAL NETWORKS

12.1 INTRODUCTION

Although fuzzy and neural systems are structurally different, they share a rather complementary nature as far as strengths and weaknesses are concerned. In this chapter we will examine the possibilities of introducing fuzzy operations within individual neurons and networks. Improving the overall expressiveness and flexibility of neural networks is what is sought. In the next chapter we will bring neuronal learning capabilities into fuzzy systems. Making fuzzy systems capable of on-line adaptation would be the desirable objective there. Neuronal enhancements of fuzzy systems as well as the fuzzification of neural systems aim at exploiting the complementary nature of the two approaches through their integration into a soft computing paradigm that permits a certain tolerance for imprecision and uncertainty.

Applying fuzzy methods into the workings of neural networks constitutes a major thrust of neurofuzzy computing (Gupta and Rao, 1994; Gupta, 1994; Pedrycz, 1993; Hirota and Pedrycz, 1993b). Although the field is an active area of research undergoing major changes, in this chapter, at the risk of omitting important research findings and developments, we attempt to introduce some fundamental notions and applications. To begin with, we briefly review the basic model of the artificial neuron we presented in Chapter 7 and then proceed with the "fuzzification" of its workings. Mathematical models of fuzzy neurons employ adaptive fuzzy relations and operators at the synapses in order to convert the external inputs into the synaptic output. Fuzzy logic operators such as min ($\wedge$) and max ($\vee$), and more generally T norms and S norms, are used to perform the confluence and aggregation of dendritic inputs to a neuron's main body, or *soma*. Although fuzziness may be intro-

duced at all aspects of the workings of an artificial neuron (i.e., inputs, weights, aggregation operation, transfer function, and output), the main thrust of fuzzy neural nets has focused on (a) the fuzzification of the dendritic inputs and (b) the aggregation operation of a conventional neuron. The result is a variety of fuzzy neurons differing in properties according to whether, for example, instead of summation we aggregate the inputs through max, min, or some other T-norm and S-norm operation. At the end of the chapter we present a set of applications and a summary of recent developments. It should be stressed, however, that since there is a rapidly growing volume of research dealing with fuzzified neural networks, our survey is partial and unfortunately incomplete.

12.2 FROM CRISP TO FUZZY NEURONS

As we have seen in Chapter 7, a neural network consists of densely interconnected information processing units called *artificial neurons*. The structure of an artificial neuron is schematically reviewed in Figure 12.1. It consists of *external inputs*, *synapses*, *dendrites*, a *soma*, and an *axon* through which individual neural output is transmitted to other neurons. Let us call this the jth neuron of the network. We recall that a vector of external inputs $[x_1, x_2, \ldots, x_n]^T$ enters the jth neuron and gets modified by weights $w_{1j}, w_{2j}, \ldots, w_{nj}$ representing the synaptic junctions of the neuron. In earlier chapters, we considered these weights as simple gains—that is, scalars modifying via multiplication the external input vector $[x_1, x_2, \ldots, x_n]^T$. In general, however, the synaptic weights may be functions of the external

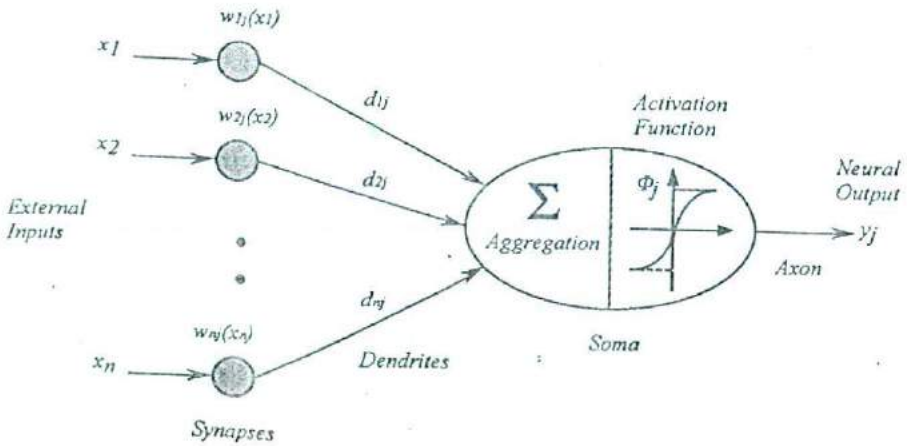


Figure 12.1 Simplified model of a neuron as an information processor.

inputs—that is, $w_{1j}(x_1), w_{2j}(x_2), \ldots, w_{nj}(x_n)$. Each synaptic output constitutes an input to the soma, called the *dendritic input*. Thus, the input to the jth neuron's soma is the vector of dendritic inputs $[d_{1j}, d_{2j}, \ldots, d_{nj}]^T$, where each dendritic input is a transformed version of an external input x_i; that is,

$$d_{ij} = w_{ij}(x_i) \tag{12.1-1}$$

The weighting function $w(\cdot)$ that models the synaptic junction between the axon of the transmitting neuron and the dendrite of the receiving neuron is thought of as a memory of the neuron's past experience, capable of adapting to new experiences through learning.

The neuron produces an output response when the aggregate activity of all dendritic inputs exceeds some threshold level T_j. Computing this aggregate input activity is an essential somatic operation as seen in Figure 12.1. Mathematically, this is usually expressed as

$$I_j = \sum_{i=1}^{n} d_{ij} \tag{12.2-2}$$

where n is the number of dendritic inputs to the neuron. It should be mentioned, however, that there is nothing sacred about summation as the aggregation operator in equation (12.2-2). We could, and indeed we will, use other aggregation operators—for example, min, max, and more generally T norms and S norms—in place of summation.[1]

Finally, the output y_j of the jth neuron is produced by the other essential operation within a neuron's soma, which is that performed by the activation (or *transfer*) *function* Φ_j (really a *decision function*). The neural output y_j is mathematically expressed as

$$y_j = \Phi_j[I_j, T_j] \tag{12.2-3}$$

where Φ_j is the activation function that describes the degree to which the jth neuron is active, I_j is the total aggregate input activity incident on the *soma* of the neuron, and T_j is the inherent threshold level for this neuron. The *perceptron*, for example, is an artificial neuron with a neural output given by

$$y_j = \operatorname{sign}\left[\sum_{i=1}^{N} w_{ij}x_i + T_j\right] \tag{12.2-4}$$

where the activation function is assumed to be a binary "on–off" function given by sign $[\cdot]$, the aggregation operator is the summation of weighted inputs, and the inherent threshold T_j is a negative bias value. For the

[1] See Appendix for a discussion of T norms and their co-norms, called S norms.

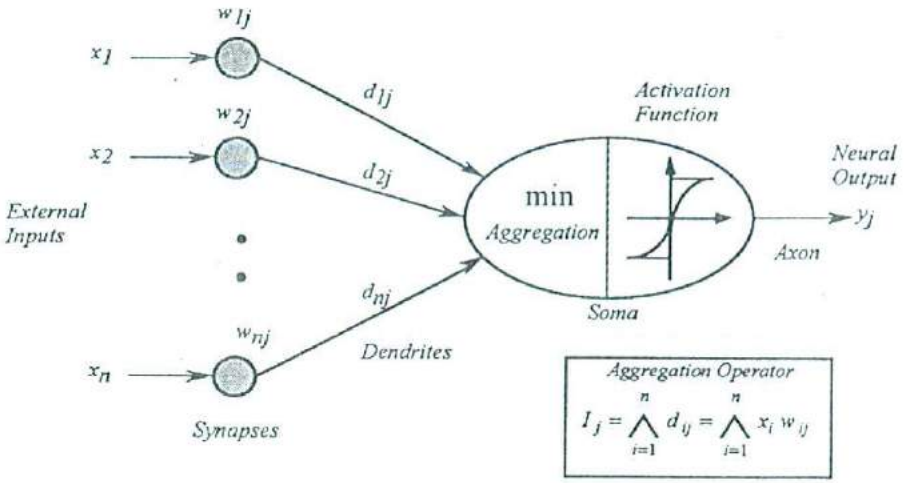


Figure 12.2 An example of a min (*AND*) fuzzy neuron.

perceptron, all external inputs and the resultant neural response are assumed to be binary (± 1). The synaptic weights, w_{ij}, may be either positive (*excitatory*) or negative (*inhibitory*) real numbers. Both the synaptic weights and the threshold level are assigned to the neuron during *training*.

Fuzzy Neurons and Fuzzy Neural Networks

A fuzzy neuron has the same basic structure as the artificial neuron shown in Figure 12.1, except that some or all of its components and parameters may be described through the mathematics of fuzzy logic. There are many possibilities for fuzzification of an artificial neuron and hence one encounters a variety of fuzzy neurons in the literature, all possessing interesting logic-oriented information processing properties. Figure 12.2 shows a fuzzy neuron where the external input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in R^n$ is defined over the unit hypercube $[0, 1]^n$ and is comprised of fuzzy signals bounded by graded membership over the unit interval $[0, 1]$.[2] The external inputs, after being modified by synaptic weights w_{ij} (also defined over the unit interval), become dendritic inputs d_{ij} to the soma. Input modification may be done through straightforward multiplication $d_{ij} = x_i w_{ij}$ or taking the maximum between input and weight $d_{ij} = x_i \vee w_{ij}$ (i.e., like an *OR-gate*).

The dendritic inputs are processed by an aggregation operator I_j that selects the minimum ($\wedge$) of the product (or max) modifications; for example,

$$I_j = \bigwedge_{i=1}^{n} d_{ij} = \bigwedge_{i=1}^{n} x_i w_{ij} \tag{12.2-5}$$

[2] For simplicity we will use x_i instead of μ_i, and so on, when referring to fuzzy signals.

This type of fuzzy neuron may be thought of as the implementation of fuzzy conjunction (*AND-gate*). Generally, fuzzy neurons use aggregation operators such as min and max and more generally T norms and S norms instead of summation as in equation (12.2-2).

As far as the meaning and purpose of neuronal fuzzification goes, we can say that each fuzzy neuron may be thought as the representation of a linguistic value such as *LOW*, *MEDIUM*, and so on.[3] Hence the output of the neuron y_j in Figure 12.2 could be associated with membership to some linguistic value; that is, y_j expresses the degree to which the input pattern $[x_1, x_2, \ldots, x_n]^T$ belongs to a given linguistic category. In other words, the output y_j is a real value in the interval [0, 1] indicating the degree to which the applied external inputs are able to generate the given linguistic value. The jth neuron after receiving n inputs $[x_1, x_2, \ldots, x_n]^T$ and producing an output y_j can subsequently convey this degree to the $m - 1$ other fuzzy neurons in a network consisting of m neurons.

The synaptic operations, but most importantly the aggregation operator, and the activation function determine the character of a fuzzy neuron. Using different aggregation operators and activation functions results in fuzzy neurons with different properties. Thus, many different types of fuzzy neurons can be defined. Consider, for example, the following neurons (Kwan and Cai, 1994).

Max (*OR*) Fuzzy Neuron

A *max fuzzy neuron* is a neuron that uses an aggregation function that selects the maximum ($\vee$) of the dendritic inputs to the soma; that is,

$$I_j = \bigvee_{i=1}^{n} x_i w_{ij} \tag{12.2-6}$$

(A max fuzzy neuron is an implementation of a *logical OR*; hence we can also call this an *OR fuzzy neuron*.[4])

Min (*AND*) Fuzzy Neuron

A *min fuzzy neuron* is a neuron that uses an aggregation function that selects the minimum ($\wedge$) of the dendritic inputs; that is,

$$I_j = \bigwedge_{i=1}^{n} x_i w_{ij} \tag{12.2-7}$$

[3]Actually fuzzy neurons may model *if/then* rules also, as we shall see later on.

[4]A special class of *OR* and *AND* fuzzy neurons that has been defined by Pedrycz in terms of T norms will be examined later in the chapter.

(A min fuzzy neuron is an implementation of a logical *AND*; hence it can also be called an *AND fuzzy neuron*.)

In addition, one can define *input fuzzy* neurons such as the *fan-in neurons* that we have seen in Chapter 7 whose purpose is simply to distribute input signals to other neurons. An *input fuzzy neuron* is an element used in the input layer of a fuzzy neural network, and it has only one input x such that

$$y = x \tag{12.2-8}$$

In general, the weights, the activating threshold, and the output functions which describe the interaction between fuzzy neurons could be adjusted via a learning procedure resulting in neurons that are adaptive. The aim is, of course, to synthesize fuzzy neural networks capable of learning from experience.

12.3 GENERALIZED FUZZY NEURON AND NETWORKS

Let us consider a neural network consisting of m fuzzy neurons, each admitting n inputs. As noted in the previous section, fuzziness may be introduced at the synaptic inputs (weights), the aggregation operation, and the transfer function of individual neurons. Thus, fuzzy sets can be used to describe various aspects of neuronal processing (Gupta and Knopf, 1992). The following are conventions frequently encountered in fuzzy neural networks.

Synaptic Inputs. The input vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]^T \in R$ to a fuzzy neuron may be thought of as grades of membership to a fuzzy set. For simplicity we do not employ the usual symbol for membership (μ_i). Rather the individual inputs $x_i \in [0, 1]$ are taken to represent fuzzy signals bounded by a graded membership over the unit interval.

Dendritic Inputs. For each jth neuron in the network ($j = 1, 2, \ldots, m$) the dendritic inputs are also bounded by a graded membership over the unit interval. Thus, if we let $u \in [0, 1]$ designate an element of a generic universe of discourse $[0, 1]$, we would use for defining fuzzy quantities the dendritic inputs are fuzzy sets

$$d_{ij} = \sum_{j-1}^{m} \mu_{d_{ij}}(u)/u, \qquad u \in [0, 1] \tag{12.3-1}$$

Aggregated Values. The output of the aggregation operator in each of the m fuzzy neurons of the network can also be thought of as graded membership over the unit interval. Thus we have

$$I_j = \sum_{j-1}^{m} \mu_{I_j}(u)/u, \qquad u \in [0, 1] \tag{12.3-2}$$

Neuron Output. Finally, the output y_j of each of the m fuzzy neurons may also be thought of as a grade of membership to a fuzzy set; that is,

$$y_j = \sum_{j=1}^{m} \mu_{y_j}(u)/u, \qquad u \in [0, 1] \tag{12.3-3}$$

The weighting function w_{ij} transforming an external input x_i into dendritic signal d_{ij} for the jth fuzzy neuron does not have to be just a simple gain. It can, in general, be a *fuzzy relation* defined over the Cartesian product $w_{ij} = x_i \times d_{ij}$. Such a synaptic junction fuzzy relation between the external inputs x_i and dendritic inputs d_{ij} may assume many forms, with the simplest and actually the most common being $d_{ij} = x_i w_{ij}$. More generally, however, the dendritic inputs d_{ij} may be given by the composition $x_i \circ w_{ij}$ of fuzzy input signals and the weight relation; that is,

$$d_{ij} = x_i \circ w_{ij} \tag{12.3-4}$$

The concept of fuzzy negation is used in order to produce both *excitatory* and *inhibitory* inputs to a fuzzy neuron.[5] Consider a jth fuzzy neuron such as the one shown in Figure 12.3. The synaptic outputs, d_{ij}, may be modified to produce *excitatory* or *inhibitory* effects by defining a new variable δ_{ij} to denote both excitatory and inhibitory inputs received by the soma, and a negation operation that modifies d_{ij} as follows:

$$\delta_{ij} = \begin{cases} d_{ij} & \text{(for excitatory inputs)} \\ \bar{d}_{ij} & \text{(for inhibitory inputs)} \end{cases} \tag{12.3-5}$$

where the inhibitory inputs are fuzzy complements of the excitatory inputs

$$\bar{d}_{ij} = 1 - d_{ij} \tag{12.3-6}$$

Consider, for example, the fuzzy neuron shown in Figure 12.3. This neuron receives four dendritic inputs: $d_{1j}, d_{2j}, d_{3j}, d_{4j}$. The first two are *excitatory* inputs sent to the aggregation operator just as they are, while the second two are *inhibitory* inputs, which are complemented in a fuzzy sense according to equation (12.3-6). We graphically indicate the inhibitory signals by small (white) circles at the end of the corresponding arrows as shown in Figure

[5]Since we use the [0, 1] range, it is not possible to use negative values for inhibitory inputs.

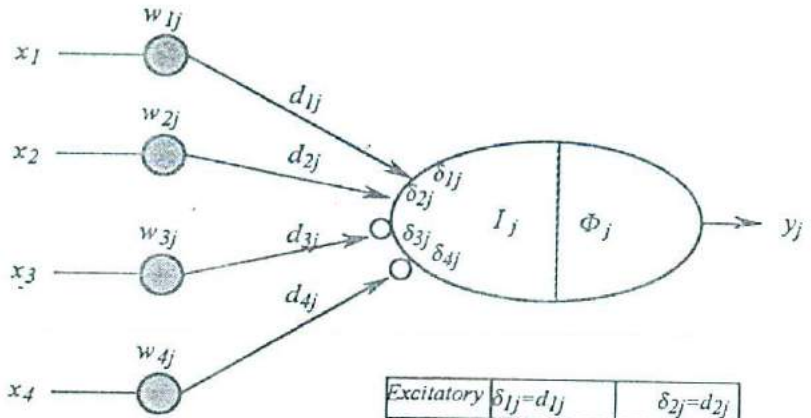


Figure 12.3 *Excitatory* and *inhibitory* dendritic inputs to a fuzzy neuron.

12.3. Thus this neuron's aggregation operator I_i will aggregate the following signals:

$$\begin{aligned} \delta_{1j} &= d_{1j} \\ \delta_{2j} &= d_{2j} \\ \delta_{3j} &= 1 - d_{3j} \\ \delta_{4j} &= 1 - d_{4j} \end{aligned} \tag{12.3-7}$$

The result of the aggregation will be subsequently modified by the function Φ_j to produce the neuron's output y_j.

12.4 AGGREGATION AND ACTIVATION FUNCTIONS IN FUZZY NEURONS

In a fuzzy neuron the aggregation operator I_j may be a T norm (see Appendix) mathematically expressed as

$$I_j = \mathop{T}_{i=1}^{n} \delta_{ij} \tag{12.4-1}$$

Often, but not always, fuzzy neurons do not explicitly use a threshold; thresholding may instead be contained within the choice of the activation function. The activation function Φ_j is a mapping operator that transforms

the membership of the aggregate fuzzy set I_j into the fuzzy set of the neuronal response y_j. In a sense, this mapping operation corresponds to a linguistic modifier such as *VERY* and *MORE-OR-LESS* (see Chapter 2). The role of this modification is to enhance or diminish the degree to which the external inputs give rise to the fuzzy value represented by the jth fuzzy neuron, before becoming an external input to neighboring neurons. Thus, a general expression of the response of the jth fuzzy neuron may be written as

$$y_j = \Phi_j[I_j] = \Phi_j\left[\underset{i=1}{\overset{n}{T}} \delta_{ij}\right] \tag{12.4-2}$$

where each dendritic input δ_{ij} is given by equations (12.3-5) and (12.3-6).

If the activation function is assumed to be a linear relationship with unit slope (i.e., $y_j = I_j$), we have an interesting special case (we will see more of it in the following sections), a simplified fuzzy neuron whose response can be written as

$$y_j = \underset{i=1}{\overset{n}{T}} \delta_{ij} \tag{12.4-3}$$

The concepts of T norm and S norm (or T conorm), originally used in the field of probability theory, provide a means for generalizing and parametrizing fuzzy set operations such as *union* and *intersection* as well as implication operators, fuzzy inferencing, and fuzzy neurons (Dubois and Prade, 1980; Gupta and Qi, 1991; Terano et al., 1992; Terano et al., 1994).

A T norm can be thought of as a circuit (gate) with two inputs (x_1, x_2) and one output $T(x_1, x_2)$, also written as $x_1 T x_2$. The most widely used T norm is the min; that is, $T(x_1, x_2) = x_1 \wedge x_2$ (but also *algebraic product*, *bounded product*, *and drastic product* are all T norms; see Appendix).

An S norm can be thought of as a circuit with two inputs (x_1, x_2) and one output $S(x_1, x_2)$, also written as $x_1 \; S \; x_2$. A very common S norm is the *logical sum* or max; that is, $S(x_1, x_2) = x_1 \vee x_2$ (but also the *algebraic sum*, *bounded sum*, and *drastic sum* are some other S norms).

The relationship between T norms and S norms is given by *fuzzy De Morgan's laws*, which may be written as

$$\begin{aligned} T(x_1, x_2) &= \bar{S}(\bar{x}_1, \bar{x}_2) \\ S(x_1, x_2) &= \bar{T}(\bar{x}_1, \bar{x}_2) \end{aligned} \tag{12.4-4}$$

where T is the T norm and S is the S norm and the bar over the symbols indicates negation.

Let us consider the simplified fuzzy neuron, shown in Figure 12.4*a*, using a linear transfer function and output given by equation (12.4-3). For simplicity we assume that the dendritic inputs are directly received from the external inputs ignoring any weight function modifications. This neuron can be

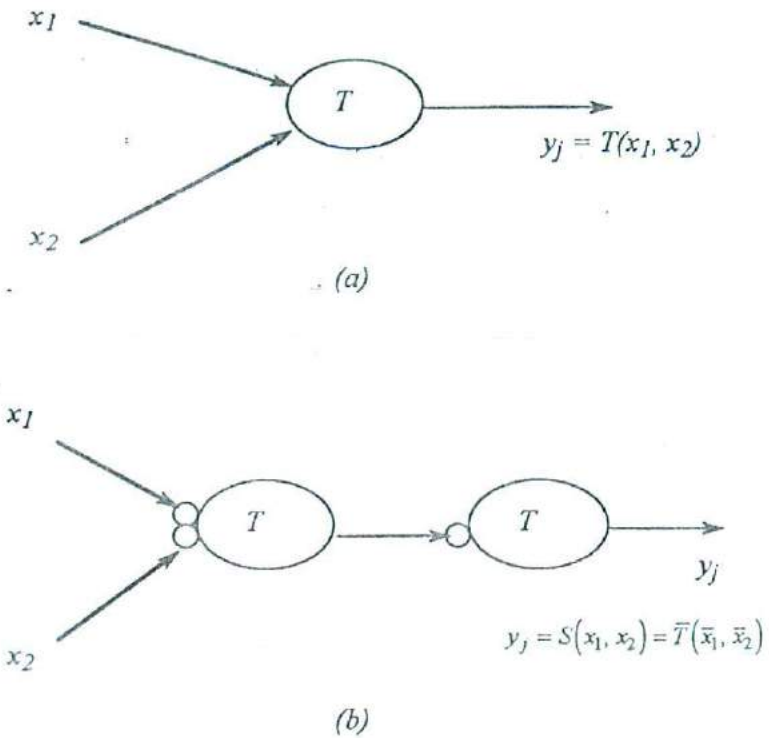
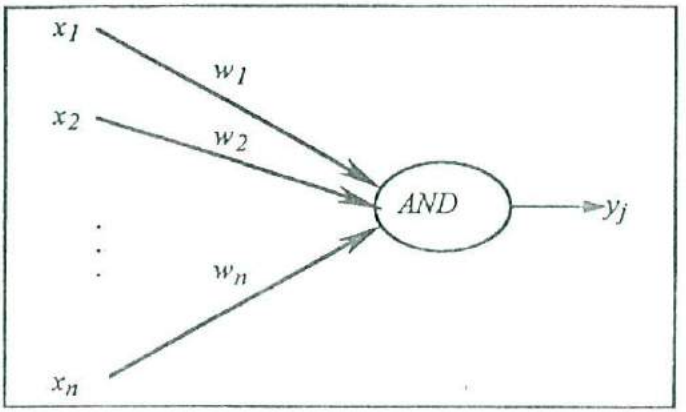


Figure 12.4 A simplified fuzzy neuron using T-norm aggregation and a linear transfer function can perform both (*a*) T norm and (*b*) S norm operations on the signals (x_1, x_2).

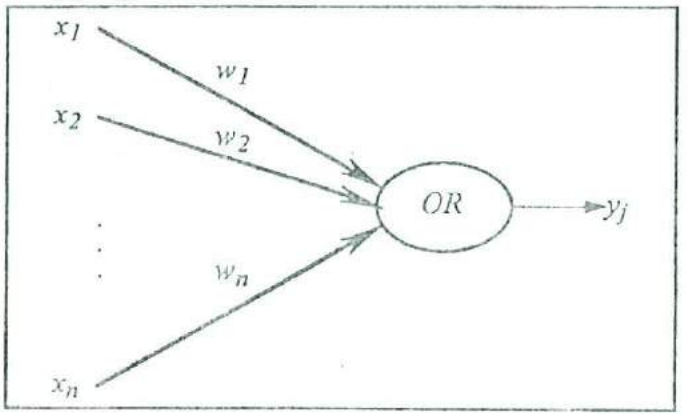
thought of as the realization of a T norm operation. With the aid of equations (12.4-4), this simplified fuzzy neuron can be used to construct a network of neurons, such as the one shown in Figure 12.4*b*, that realizes an S norm. As indicated by the small circles in the left neuron of Figure 12.4*b* the inputs are first negated in a fuzzy sense and then aggregated by a T-norm aggregation. The output of this neuron is complemented again by the second neuron, in accordance with equation (12.4-4). Thus the network of neurons in Figure 12.4*b* is a realization of an S norm, made out of cascaded neurons that individually use T norm for the aggregation operation.

12.5 *AND* AND *OR* FUZZY NEURONS

A special class of fuzzy neurons are the *AND* and *OR* neurons shown in Figure 12.5. These neurons employ the T norm and S norm operations for



(a)



(b)

Figure 12.5 (a) An *AND* fuzzy neuron and (b) an *OR* fuzzy neuron.

forming dendritic inputs to the soma and for aggregating them (Pedrycz, 1993, Rueda and Pedrycz, 1994; Pedrycz and Rocha, 1993).

The *AND* neuron first uses equation (12.3-4) to perform an S norm or *OR* operation between external input x_i and corresponding weight w_{ij}; that is,

$$d_{ij} = x_i \; OR \; w_{ij} \tag{12.5-1}$$

Subsequently, it uses a T norm or *AND* operation to carry the following

aggregation of its dendritic inputs (assuming a linear activation function):

$$y_j = (x_1 \; OR \; w_{1j}) \; AND \; (x_2 \; OR \; w_{2j}) \; AND \; \cdots \; AND \; (x_n \; OR \; w_{nj}) \quad (12.5\text{-}2)$$

It should be noted that *AND* and *OR* are generally realized by taking any T norm and S norm—for example, logical product (min), logical sum (max), algebraic product, algebraic sum, and so on. In practice, however, the min and max interpretations are most commonly used.

The output of an *AND* neuron can succinctly be written using T norms and S norms as

$$y_j = \underset{i=1}{\overset{n}{T}} (x_i \; S \; w_{ij}) \quad (12.5\text{-}3)$$

The *OR* neuron, on the other hand, performs a complementary computation:

$$y_j = \underset{i=1}{\overset{n}{S}} (x_i \; T \; w_{ij}) \quad (12.5\text{-}4)$$

Both the *AND* and *OR* neurons given above are intrinsically excitatory in their behavior; that is, higher values for the x_i's imply higher values for y_j. To allow for inhibitory behaviors of such *AND* or *OR* fuzzy neurons (and still maintain the standard [0, 1] range of the grades of membership) we include negated values of x_{ij}—that is, $1 - x_{ij}$ [as we have seen before in equation (12.3-6)]—thus potentially doubling the size of the input vector. The *AND* and *OR* neuron can now handle both inhibitory and excitatory behaviors, depending on the numerical values of the connections.

Now let us look at some interesting boundary cases, say in the *AND* neuron (Pedrycz, 1993). First, suppose that all the weights of a neuron equal zero—that is, $w_{ij} = 0$. Then we should have $x_i \; S \; w_{ij} = x_i$ (e.g., $x_i \; S \; 0 = x_i$). Second, if all the weights are unity, $w_{ij} = 1$, we have $x_i \; S \; 1 = 1$; that is, the input does not have any influence on the output. To deal with such extremes, a bias term may be added as an additional term in (12.5-3) driven by a constant input signal always equal to 0, say $0 \; S \; w_{0j}$, where w_{0j} denotes the connections associated with this input. The *AND* neuron incorporating such a bias is given by

$$y_j = \underset{i=0}{\overset{n}{T}} (x_i \; S \; w_{ij}) \quad (12.5\text{-}5)$$

where, by convention, we put $x_0 = 0$. A similar bias term may be added to an

OR neuron, making equation (12.5-4) look like

$$y_j = \underset{i=0}{\overset{n}{S}} (x_i \, T \, w_{ij}) \tag{12.5-6}$$

In equations (12.5-5) and (12.5-6) we have assumed that the neuronal output is produced immediately after aggregation; in other words, the activation function used is linear. However, a nonlinear activation function such as a sigmoidal function may also be used.

12.6 MULTILAYER FUZZY NEURAL NETWORKS

The fuzzy neurons discussed in the previous section can be put together to construct more general computational structures with enhanced representational capabilities. While in Part II of the book we used networks composed of identical neurons, the networks built out of fuzzy neurons are often heterogeneous; that is, they are composed of neurons with different computational characteristics—for example, *AND* or *OR* fuzzy neurons—organized into several layers (most commonly three).

Let us look at a three-layer neural network built out of *AND* and *OR* neurons [originally proposed by Pedrycz (1993)]. Each layer in the network is constructed out of neurons of the same type (i.e., *AND* or *OR* only). A hidden layer is used to enhance the representational capabilities of the entire structure. In Figure 12.6*a* the hidden layer is made out of *AND* neurodes, while in Figure 12.6*b* the hidden layer has only *OR* neurons. These are actually two different types of networks: One uses *AND* neurons in the hidden layer, with the output layer consisting of a single *OR* neuron, whereas the other uses *OR* neurons in its hidden layer and a single *AND* neuron in the output layer. As seen in Figure 12.6*a*, the first network has an input layer consisting of $2n$ input neurodes; both networks use direct signals and their complements, namely, $x_1, x_2, \ldots, x_n, \bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n$. Because the neurons of the first layer are fan-in neurodes [see equation (12.2-8)], (they simply distribute the input signals to all the nodes of the hidden layer.

The hidden layer itself is composed of p *AND* neurons, each one of them sending to the output layer signal

$$z_l = AND(w_l, x), \qquad l = 1, 2, \ldots, p \tag{12.6-1}$$

The weight vector of connections, w_l, captures information about the connections between the lth node of the hidden layer and the input nodes; that is,

$$z_l = \left[\underset{i=1}{\overset{n}{T}} (x_i \, S \, w_{li}) \right] T \left[\underset{i=1}{\overset{n}{T}} (\bar{x}_i \, S \, w_{l(n+i)}) \right] \tag{12.6-2}$$

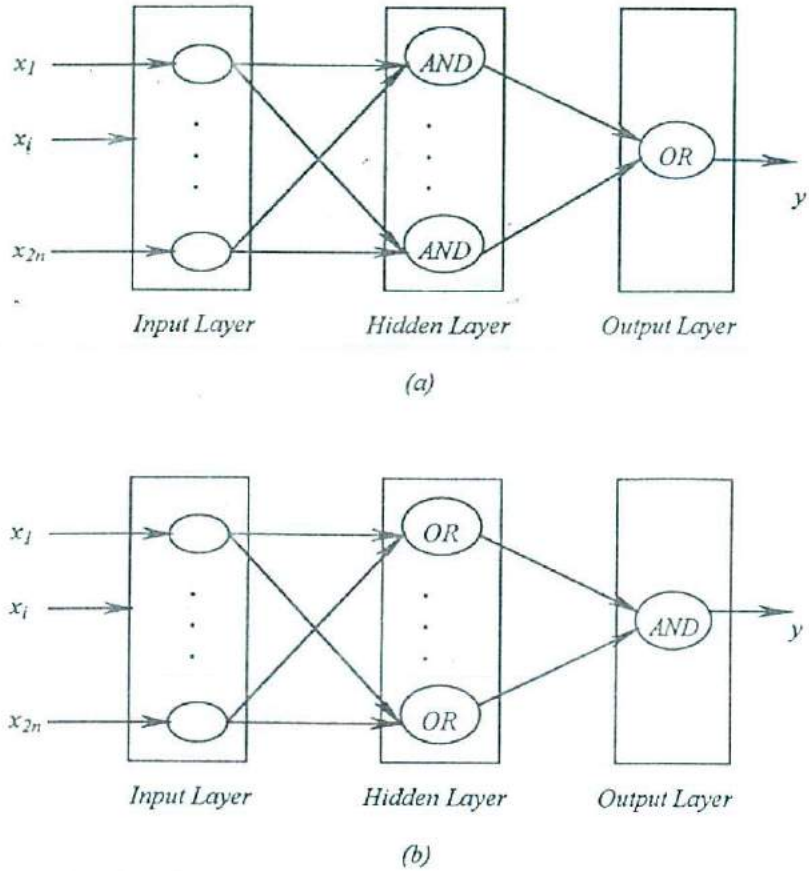


Figure 12.6 Three-layer networks with fuzzy (*a*) *AND* and (*b*) *OR* neurons in the hidden layer.

where $l = 1, 2, \ldots, p$. The output layer consists of a single *OR* performing an aggregation of z's:

$$y = \underset{l=1}{\overset{p}{S}} (z_l \, T \, v_l) \tag{12.6-3}$$

If we put *OR* neurons in the hidden layer and an *AND* neuron at the output layer (see Figure 12.6*b*, we perform a similar sequence of computations, except we interchange T norm and S norm operations in equations (12.6-2) and (12.6-3).

Other network architectures are also possible. Consider, for example, the homogeneous network shown in Figure 12.7. Here only T-norm aggregating (AND) neurons are used to realize a network structure of three layers emulating a system of m fuzzy *if/then* rules, that is a fuzzy rule base, receiving n inputs and producing one output. The first layer consists of m fuzzy neurons, with each neuron being a representation of an *if/then* rule. As seen in Figure 12.7, the output of each fuzzy neuron in layer 1 becomes an external input to a single OR neuron realized as cascaded AND neurons (see Figure 12.4) comprising layers 2 and 3. This three-layered neural network architecture can be used to simulate a situation when n fuzzy inputs are applied to m fuzzy inference rules (Gupta and Knopf, 1992).

12.7 LEARNING AND ADAPTATION IN FUZZY NEURAL NETWORKS

The process of learning in fuzzy neural networks consists of modifying their parameters by presenting them with examples of their past experience. How can this be done in practice? Typically by adjusting the weights of the networks so that a certain performance index is optimized (maximized or minimized). This requires that a collection of input–output pairs be specified and also requires a performance index that expresses how well the network maps inputs x_k into the corresponding target values of the output t_k.

Let us recall that an important difference between a crisp (nonfuzzy) neuron and a fuzzy neuron lies in the model of the synaptic connection.

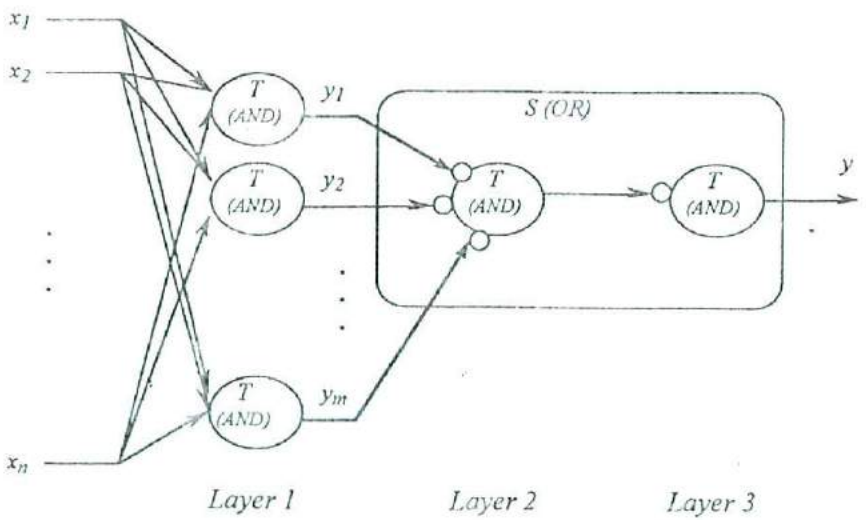


Figure 12.7 A fuzzy neural network architecture for m fuzzy rules accepting n inputs.

Synaptic connections in a crisp neuron are linear gains multiplying inputs x_i. Any adaptation or learning occurring within an individual neuron involves modifying the values of these gains by adjusting w_{ij}. For a fuzzy neuron, synaptic connections are represented as a two-dimensional fuzzy relation between synaptic inputs and outputs. Hence, learning in fuzzy neurons, in the most general case, involves changing a two-dimensional relation surface at each synapse.

Consider one synaptic connection to the jth neuron as shown in Figure 12.8. For a given external input to this synapse at time k, $x_i(k)$, we want to determine the corresponding fuzzy relation, $w_{ij}(k)$, such that we have minimum error $e_j(k)$ between the fuzzy neuron response and the desired target response $t_j(k)$. In order to achieve this, we can employ the following adaptation rule to modify the fuzzy relation surface:

$$w_{ij}(k+1) = w_{ij}(k) + \Delta w_{ij}(k) \tag{12.7-1}$$

The term $\Delta w_{ij}(k)$ is the change in the fuzzy relation surface given as a function $F[\cdot]$ of the error $e_j(k)$; that is,

$$\Delta w_{ij}(k) = F\left[e_j(k)\right] = F\left[t_j(k) - y_j(k)\right] \tag{12.7-2}$$

In multilayer networks, learning involves matching t_k (up to some error) with the output of the entire network y. For this purpose, a distance function—for example, Euclidean distance between y and t_k—may be used. Then a performance index Q (a global error term to be minimized) may be defined as follows

$$Q = \sum_{k=1}^{N} \left[y(\mathbf{x}_k) - t_k\right]^2 \tag{12.7-3}$$

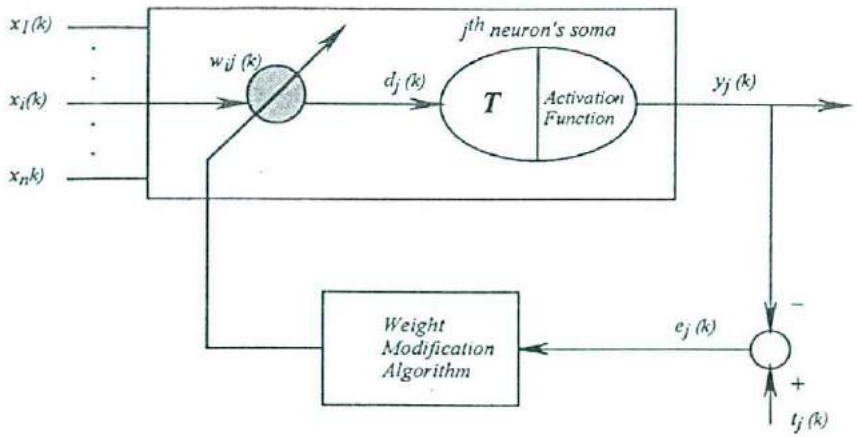


Figure 12.8 Learning at the level of the individual neuron.

Q in Equation (12.7-3) reflects quantitatively the state of the networks learning process. Optimization includes all the weights of the network between the input layer and the hidden layer as well as between the hidden layer and the output layer. The simplest update scheme is that in which the modifications are driven by a gradient of the performance index taken with respect to the connections themselves (see Chapter 8). The learning formula can be expressed as

$$\Delta(\text{connections}) = -\eta \frac{\partial Q}{\partial(\text{connections})} \tag{12.7-4}$$

where η denotes a *learning factor*, $\eta \in (0, 1)$. Detailed computations can be performed once the performance index Q and a parametric description of the network have been defined, as is done in the example that follows.

Example 12.1 Learning and Adaptation in *AND / OR* Neurons. Given a three-layer network having a hidden layer of *AND* neurons and output with an *OR* fuzzy neuron (as shown in Figure 12.6*a*) and an error based on sum of squared errors, we want to derive an on-line learning algorithm for modifying its weights. The network's neurons use *algebraic sum* for the S norm and use *product* for the T norm [see Appendix and Pedrycz (1993)].

A single pair of input–output data involves $\mathbf{x}$ and t.[6] For the hidden layer (see Figure 12.6*a*) we have the following intermediate outputs based on equation (12.6-2):

$$z_h = \left[\mathop{T}_{i=1}^{n} (x_i \, S \, w_{hi}) \right] T \left[\mathop{T}_{i=1}^{n} (\bar{x}_i \, S \, w_{h(n+i)}) \right] \tag{E12.1-1}$$

While the output layer gives [according to equation (12.6-3)]

$$y = \mathop{S}_{h=1}^{p} (z_h \, T \, v_h) \tag{E12.1-2}$$

where p denotes the dimension of the hidden layer. In order to adjust the weights, we differentiate the performance (error) function of (12.7-3) with respect to hidden layer weights, that is,

$$\frac{\partial Q}{\partial w_{hj}} = \frac{\partial Q}{\partial y} \cdot \frac{\partial y}{\partial w_{hj}} = \frac{2[y(x) - t]\partial y}{\partial w_{hj}}, \quad h = 1, 2, \ldots, p, \quad j = 1, 2, \ldots, 2n \tag{E12.1-3}$$

and then we differentiate the error function with respect to the intermediate

[6]Since we develop an on-line learning version [where each pair $(\mathbf{x}_k, t_k)$ immediately affects the connections of the network], the index k denoting the element in the training set can be dropped.

weights (we use the letter v for those);

$$\frac{\partial Q}{\partial v_h} = \frac{2[y(x) - t]\partial y}{\partial v_h}, \qquad h = 1, 2, \ldots, p \tag{E12.1-4}$$

Now we have that

$$\frac{\partial y}{\partial v_h} = \frac{\partial}{\partial v_h}\left[\underset{h=1}{\overset{p}{T}} (z_h \, T \, v_h)\right] = \frac{\partial}{\partial v_l}[A \, S \, (z_l \, T \, v_l)] \tag{E12.1-5}$$

where A is a shorthand notation defined as $A \equiv \underset{h \neq l}{\overset{p}{S}} (z_h \, S \, v_h)$.

Since we use algebraic sum and product for the T norm and S norm, we have

$$\frac{\partial}{\partial v_l} = [A + v_l z_l - A v_l z_l] = z_l(1 - A) \tag{E12.1-6}$$

$$[A \, S \, (z_{h_1} \, T \, v_{h_1})] = [A \, S \, z_{h_1} v_{h_1}] = A + z_{h_1} v_{h_1} - A z_{h_1} v_{h_1} \tag{E12.1-7}$$

Hence we can compute that the output change with respect to input weights is given by the following expression:

$$\frac{\partial y}{\partial w_{hj}} = \sum_{h=1}^{p} \frac{\partial y}{\partial z_h} \frac{\partial z_h}{\partial w_{hj}} \tag{E12.1-8}$$

The above sum reduces to single component, since only one term contributes; that is,

$$\frac{\partial z_h}{\partial w_{h_l j}} = 0, \qquad \forall h \neq h_l \tag{E12.1-9}$$

Hence, we obtain

$$\frac{\partial y}{\partial z_{hl}} = \frac{\partial}{\partial z_{hl}}\left[\underset{i=1}{\overset{p}{S}} (z_i \, T \, v_i)\right] = \frac{\partial}{\partial z_{hl}}[B \, S \, (z_{hl} \, T \, v_{hl})] = \frac{\partial}{\partial z_{hl}}[B \, S \, z_{hl} v_{hl}]$$

$$= \frac{\partial}{\partial z_{hl}}[B + z_{hl} v_{hl} - B z_{hl} v_{hl}] = v_{hl} - B v_{hl}$$

$$\frac{\partial y}{\partial w_{h_i j}} = \frac{\partial y}{\partial z_{h_l}} \frac{\partial z_{h_i}}{\partial w_{h_i j}}$$

$$\frac{\partial y}{\partial z_{h_i}} = v_l(1 - B), \qquad B = \underset{h \neq h_1}{\overset{p}{S}} (v_h \, S \, z_h)$$

$$\frac{\partial z_{h_i}}{\partial w_{h_i j}} = \frac{\partial}{\partial w_{h_i j}}\left[\prod_{i=1}^{n} (w_{h_i i} + x_i - w_{h_i i} x_i) \prod_{i=1}^{n} \left(w_{h_i (n+i)} + \bar{x}_i - w_{h_i (n+i)} \bar{x}_i\right)\right]$$

(E12.1-10)

Thus we can write

$$\frac{\partial_{h_i}}{\partial w_{h_i j}} = \begin{cases} C_1(1 - x_j), & \text{if } j \le n \\ C_2(1 - \bar{x}_j), & \text{if } j > n \end{cases} \qquad \text{(E12.1-11)}$$

where C_1 and C_2 stand for the product terms in (E12.1-10), not including x_j or its complement.

If we use nondifferentiable T norm and S norm such as minimum and maximum, the derivatives must be judiciously defined since they can severely affect the learning algorithm as we saw in Chapter 8. For example, the derivative of $(x \wedge w)$ with respect to w is

$$\frac{\partial(x \wedge w)}{\partial w} = \begin{cases} 1, & \text{if } x \ge w \\ 0, & \text{if } x < w \end{cases} \qquad \text{(E12.1-12)}$$

This type of "on–off" weight updates can easily be affected, however, by peculiarities in the connections and the data encountered during learning. One possibility for ameliorating this problem is to replace the above two-valued situation, that is, 0 or 1 in equation (E12.1-12), by some smooth, although very similar, function; for example, (Pedrycz, 1993)

$$\tfrac{1}{2}\left[(x + w) - \sqrt{(x - w)^2 + \delta^2} + \delta\right] \qquad \text{(for minimum)}$$

$$\tfrac{1}{2}\left[(x + w) + \sqrt{(x - w)^2 + \delta^2} - \delta\right] \qquad \text{(for maximum)}$$

where the parameter δ is typically a small positive constant (about 0.05). □

Example 12.2 Steering Control for an Automobile. Let us look at an example of an automatic steering control mechanism (Maeda and Murakani, 1989; Sugeno and Nishida, 1985) and its equivalent fuzzy neural network architecture (Gupta, 1994) (Gupta and Knopf, 1992)

The controller is based on a driver's ability to manipulate both the *position* and *direction* of a moving automobile on a straight highway (assumed for simplicity to travel down the middle of the road). The approximate position and direction for the vehicle with respect to the road edge is used, and hence *position-from-left-side position-from-right-side, direction-angle and change-in-direction-angle* will be fuzzy variables. The output of the controller is another fuzzy variable, namely, the steering-*angle* by which the steering wheel should be turned (Gupta and Knopf, 1992). Figure 12.9 shows the position and direction variables employed in modeling the situation.

The steering control rules consist of two rule bases. The first rule base, called the positioning algorithm, involves a linguistic description that positions the vehicle in the middle of the road, and the second, called the

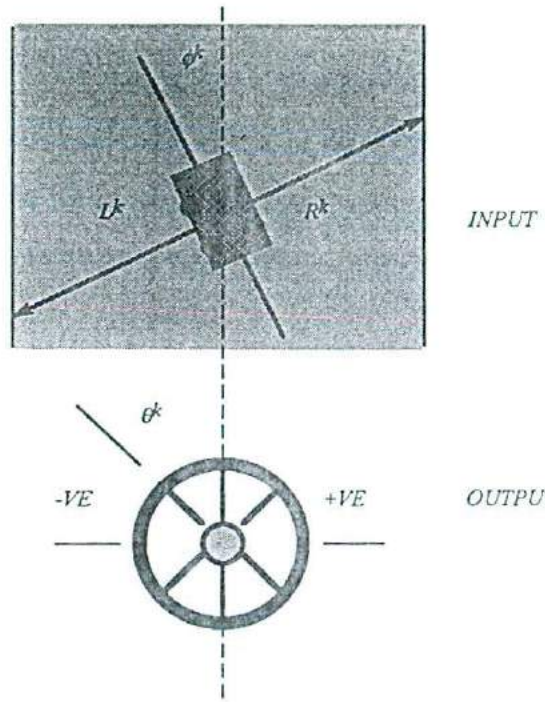


Figure 12.9 The position and direction fuzzy variables in the automobile steering control problem are left and right distances from road side, position, and steering wheel angles.

direction algorithm, involves rules to ensure that the vehicle is parallel to the edge of the road.

We can write the 16 rules of the positioning algorithm succinctly as

$$\begin{cases} \text{if } \textit{position-from-left-side} \text{ is } L_j^k \textit{ AND position-from-right-side} \text{ is } R_j^k \\ \text{then } \textit{steering-angle} \text{ is } \theta_j^k, \forall\, j = 1, \ldots, 16 \end{cases} \tag{E12.2-1}$$

The direction algorithm has nine rules that can be succinctly written as

$$\begin{aligned} &\text{if } \textit{direction-angle} \text{ is } \phi_j^k \textit{ AND change-in-direction-angle} \text{ is } \Delta\phi_j^k \\ &\text{then } \textit{steering angle} \text{ is } \theta_j^k, \forall\, j = 1, \ldots, 9 \end{aligned} \tag{E12.2-2}$$

where L_j^k, R_j^k, ϕ_j^k, $\Delta\phi_j^k$, and θ_j^k are linguistic values for the jth rule and k is the time index. These fuzzy sets are defined as follows:

L^k, R^k	Fuzzy values describing the approximate distances between the road edge and the vehicle (R, right; L, left)
ϕ^k, $\Delta\phi^k$	Fuzzy values describing the angle and change in angle for the direction of the vehicle with respect to centerline
θ^k	Fuzzy value for the output steering angle at time k

The linguistic labels of these values are as follows:

ZE	Approximately zero
S	Small
M	Medium
L	Large
P	Positive
N	Negative
PS	Positive small
PM	Positive medium
PL	Positive large
NS	Negative small
NM	Negative medium
NL	Negative large

The membership functions for the input and output fuzzy sets are shown in Figure 12.10.

Each rule in the fuzzy positioning and direction algorithms above may be represented by a single fuzzy neuron, and the collection of rules in its entirety by a neural network. Hence, for the automatic steering control mechanism the control rules are represented as the network of fuzzy neurons shown in Figure 12.11.

The outputs from the neurons of the first layer in Figure 12.11 become the inputs to one of the two neurons located in the second layer. To obtain the collective decision from either the position or direction control rules we require each neuron in the second layer to perform an S-norm operation. This is achieved by defining the inputs to both neurons as inhibitory. The expression for the position control neuron is

$$y_1 = \underset{j=1}{\overset{16}{T}} \left[N\left(\theta_j^k\right) \right] \qquad \text{(E12.2-3)}$$

and the expression for the direction control neuron is

$$y_2 = \underset{j=1}{\overset{9}{T}} \left[N\left(\theta_j^k\right) \right] \qquad \text{(E12.2-4)}$$

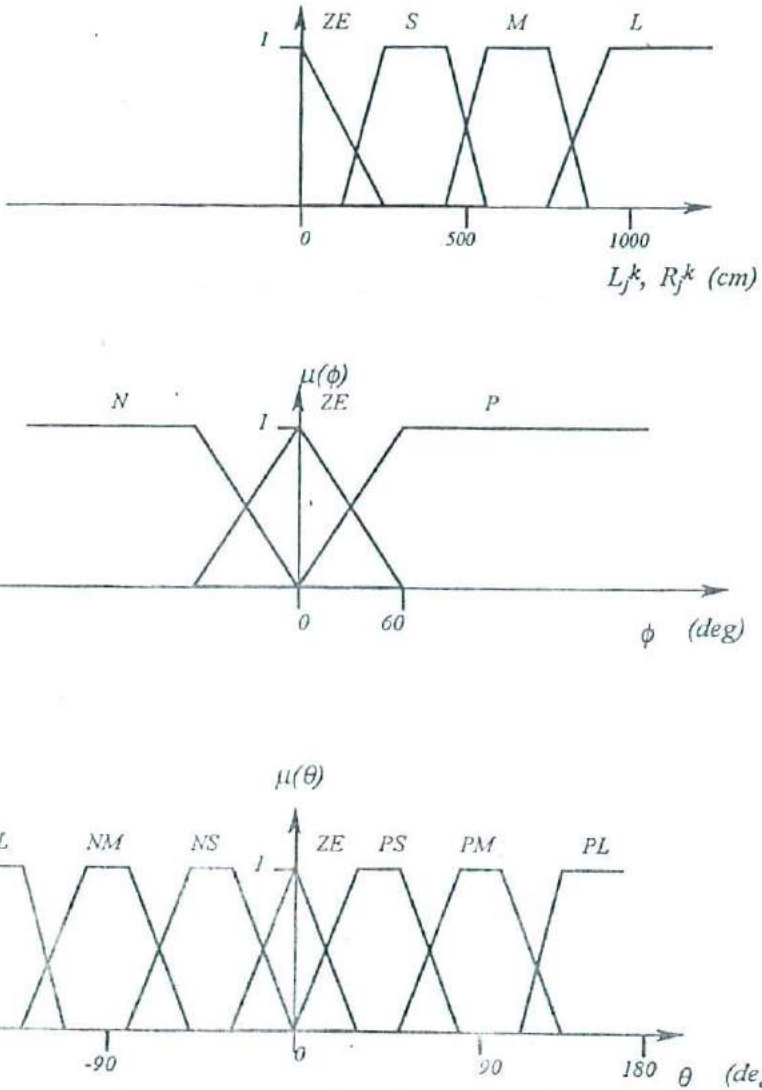


Figure 12.10 Fuzzy values for input–output variables.

The outputs from both fuzzy neurons are then transmitted to a single neuron located in a third layer as shown in Figure 12.11, producing the following output:

$$y_3 = T(y_1, y_2) \qquad \text{(E12.2-5)}$$

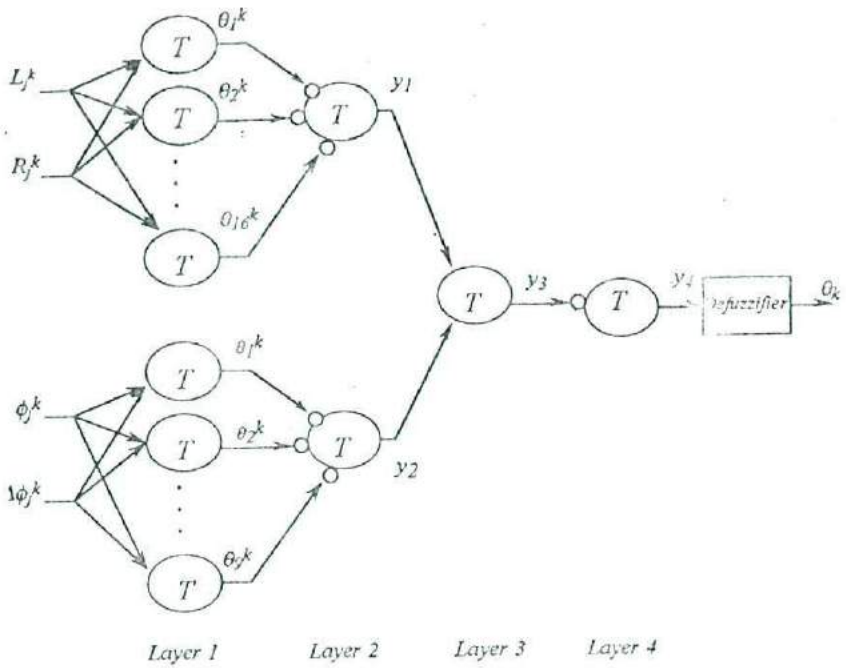


Figure 12.11 Fuzzy-neural network representation of automobile steering controller.

Finally the response of this neuron in the third layer becomes an inhibitory input of a neuron situated in the fourth layer, giving

$$y_4 = T(N(y_3)) \tag{E12.2-6}$$

y_4 is generally a fuzzy set, hence the final decision is reached through a defuzzified version of the fuzzy membership function representing y_4. □

12.8 FUZZY ARTMAP

In Chapter 9, we briefly discussed the features of adaptive resonance theory neural networks (ART) with emphasis on its unique ability to create new categories of arbitrary accuracy to accommodate inputs that did not fit into the existing categories. With the introduction of fuzzy concepts, Fuzzy ARTMAP (the MAP refers to mapping inputs to outputs), a synthesis of ART-1 and fuzzy logic, capable of accepting either analog or binary inputs, was developed by Carpenter and Grossberg (1994). Furthermore, it is able to deal with nonstationary time series as inputs.

Fuzzy ARTMAP is a self-organizing architecture that is capable of rapidly learning to recognize, test hypotheses about, and predict the consequences of virtually any input. (There is no ambiguity about the initial configuration since the network literally grows from scratch.) It involves a combination of neural and fuzzy operations that together give these useful capabilities. Like other versions of ART, its use is almost exclusively for classification, and it has only one user-selectable parameter (*vigilance*) which determines the fineness or coarseness of the patterns into which the inputs are fitted. It can learn virtually every training pattern in a few training iterations in an unsupervised mode. Yet, it can use predictive disconfirmations to supervise learning of categories that fit the statistics of patterns being categorized.

Fuzzy ARTMAP operates by autonomously determining how much compression or generalization is needed for each input category to fit the categories of choice. The more general categories have more fuzziness in the feature values that are accepted by the specific category. The acceptable range (or fuzziness) of a particular category is learned through a series of iterations that involve the use of fuzzy logic operators. The fuzzy *AND* (min) and *OR* (max) operators are used to define the range of values that are tolerated by a category for each linguistic variable or feature. The membership functions over the range from 0 to 1 (discussed in Chapters 2 through 5) directly relevant to this determination of the acceptability of an input pattern in a particular category. The min operator helps define features that are "critically present," whereas the max operator helps define features that are "critically absent." The min operator can be realized by nodes that are turned on by an external input, whereas the max operator is realized by nodes that are turned off by an external input. Thus the min and max operators can be introduced at appropriate positions in the neural network by externally controlled on–off switches. The category that best matches an input pattern is chosen by the operation of fuzzy subsethood. Fuzzy logic provides a method by which fuzzy ARTMAP adaptively categorizes analog, as well as binary, input patterns. Hence fuzzy ARTMAP can autonomously learn, recognize, and make rare events, large nonstationary databases, morphological variable types of events, and many-to-one and one-to-many relationships. These features and many other details of fuzzy ARTMAP are discussed extensively by Carpenter and Grossberg (1994).

Although fuzzy ARTMAP has proven itself as a supervised incremental learning system in pattern recognition and M- to N-dimensional mappings by comparison with other techniques, a simplified fuzzy ARTMAP (Kasuba, 1993) has been introduced. It reduces the computational overhead and architectural redundancy of fuzzy ARTMAP with no loss of pattern-recognizing capability. This description follows that of Kasuba (1993).

Normally, when backpropagation neural networks are used for pattern classification, a single output node is assigned to each category of objects that the network is expected to recognize. The creation of these categories are left up to network in both fuzzy ARTMAP and its simplified derivative. Figure

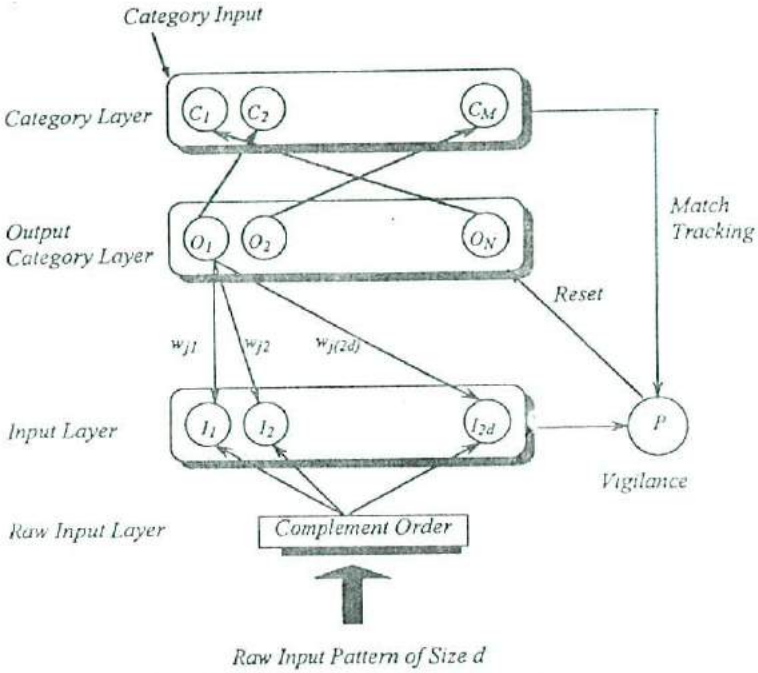


Figure 12.12 Simplified fuzzy ARTMAP structure.

12.12 shows the structure of the simplified fuzzy ARTMAP to be a two-layer network (input and output category layers) with connection weights, shows a category layer to interpret the results of output layer, and shows a "complement coder" to preprocess the raw input data. This "complement coder" normalizes the input and stretches it to twice its original size to help the network form its decision regions. The vigilance feature (0 to 1) determines the fineness of the categories and thus determines the number of categories to be created.

The expanded input (I) from the "complement coder" then flows to the input layer. Weights (w) from each of the output category nodes hold the names of the M number of categories that the network has to learn. Since a single output node can only encode a single category, it can only point to a single position in the category layer. Category input is only supplied to the category layer during the supervised training. The "match tracker" portion of the network lets it self-adjust its vigilance during learning from the level set by the user in response to errors in classification during training, thereby controlling the creation of new categories.

Complement coding is an input normalization process that represents the presence or absence of a particular feature vector a with d components in

the input. Its complement $\bar{\mathbf{a}} = 1 - \mathbf{a}$ is valid since a has a value between 0 and 1. Therefore the complement-coded input vector I is given by the $2d$-dimensional vector

$$\mathbf{I} = [\mathbf{a}, \bar{\mathbf{a}}] = [a_1, a_2, \ldots, a_d, \bar{a}_1, \bar{a}_2, \ldots, \bar{a}_d] \tag{12.8-1}$$

For instance, the three-dimensional vector (0.2, 0.8, 0.4) is transformed into the six-dimensional vector (0.2, 0.8, 0.4, 0.8, 0.2, 0.6) through complement coding. This process automatically normalized the input vectors, indicating that the norm of any vector is just the sum of all elements in the vector. Hence, the sum of the elements of a complement-coded vector is equal to the dimensionality of the original non-complement-coded input vector.

When this network is presented with an input pattern, all output nodes become active to some degree. This output activation is denoted by $\mathbf{T}_j$ for the jth output node and its weights $\mathbf{w}_j$. The function to produce this activation is given by

$$\mathbf{T}_j(\mathbf{I}) = \frac{|\mathbf{I} \wedge \mathbf{w}_j|}{\mathbf{a} + |\mathbf{w}_j|} \tag{12.8-2}$$

where a is a small value near zero, usually about 0.0000001. The winning output node is the node with the highest activation; that is, the winner is $\max \mathbf{T}_j$. Hence, the category associated with the winning output node is the network's classification of the current input pattern.

The match function is used to compare the complement-coded input features and a particular output node's weight to help determine if learning has occurred. It calculates the degree to which I is a fuzzy subset of $\mathbf{w}_j$—that is, whether the match function value indicates that the current input is a good enough match or whether a new output category should be generated. If this match function is greater than the vigilance function, the network is said to be in a state of *resonance*. A mismatch occurs if the match function value is less than the vigilance, indicating that the current output node does not meet the encoding granularity of the vigilance. Once a winning output node j has been selected to learn a particular input pattern I, the top-down vector $\mathbf{w}_j$ from the output node is updated. The simplified fuzzy ARTMAP neural network is a general-purpose classifier with top-down weight's decision-making facilities so transparent that its classification rules can literally be read out of the network. It can be compared to a self-learning expert system in that it learns from example.

12.9 FUZZY-NEURAL HYBRID DATA REPRESENTATION

During the last few years there has been a large and energetic upswing in research efforts aimed at synthesizing fuzzy logic with neural networks.

Neural networks possess advantages in the areas of learning, classification, and optimization, whereas fuzzy logic has advantages in areas such as reasoning on a high (semantic or linguistic) level. The two technologies nicely complement each other, and a number of synergisms have been proposed [see Bezdek (1995) and Saleem (1994)]. In addition to the fuzzy neurons and networks we have seen, several applications have focused on utilizing and processing fuzzy inputs and outputs in conjunction with conventional networks (Travis and Tsoukalas, 1994; Werbos, 1992). An additional variation is using fuzzy logic to control crisp neural network processes. Let us take a look at some of these.

Fuzzy Representations of Variables that are Inputs and Outputs of Neural Networks

Sometimes dealing with all possible outputs of a neural network requires a large number of neurons, thereby increasing the complexity and training time. For instance, if we consider the temperatures between freezing boiling of water, even on the centigrade scale, there would be 100 integral values. The number can be reduced by grouping these 100 values into groups of 10 successive values and representing each group of 10 values with a single value (e.g., the 10 values in the range 21° to 30° could be represented by 15°). Hence, the scale would become 5°, 15°, 25°, 35°, . . . , 95°. Such groupings lead us to considering fuzzy or linguistic representation of the variable, where 0° to 10° might be "extremely cold," 10° to 20° might be "very cold," 20° to 30° might be "cold," 30° to 40° might be "slightly cold," and so on. If one views these temperatures from the standpoint of human comfort, as opposed to the distance along a scale between the freezing and boiling point of water, a nonuniform distribution with fewer values might be more appropriate—that is, 0° to 15°, 16° to 20°, 21° to 23°, 24° to 30°, and 31° to 100°. In linguistic terms, these ranges might be designated *too cold*, *cold*, *comfortable*, *hot*, and *too hot*. Generally, the sequence of events that are involved in utilizing fuzzy data in neural networks is as follows:

1. Crisp (or fuzzy) data are converted into membership functions or sets.
2. These memberships or sets are then subject to fuzzy logic operations.
3. The resultant sets are then defuzzified into crisp data that are presented to the neural network.
4. The neural network may also have their direct inputs that are crisp and do not need the fuzzy processing.
5. The output of the neural network is a crisp set that utilizes a membership function to convert it into a fuzzy variable.
6. This fuzzy output is then operated on fuzzy logic.
7. The fuzzy logic output is then defuzzified to produce a crisp output.

Fuzzy "One-of-*n*" Coding of Neural Network Inputs

Typically, an input variable is represented by a single input node in the neural network. When an input variable has a special relationship with other variables over only a small portion of its range, the training process of the neural network is made especially difficult. Sometimes a nonlinear transformation is used to emphasize the particular region, but this is usually not a satisfactory process. The difficulty can be overcome by providing the neural network with neurons that focus on one region of the variables' domain. The domain is divided into n regions (where n is typically 3, 5, or 7), and each is assigned a fuzzy set having a triangular membership function. (Of course, the lowest and highest sets have horizontal extensions starting at the minimum and maximum expected values, respectively.) The membership value in each fuzzy set determines the activation level of its associated input neuron. This "one-of-n" coding expands the range of the variable into n network inputs, each covering a fraction of the domain. While the resulting specialization often facilitates learning, the increase in the number of neurons tends to slow down learning. This technique is advantageous only when the importance of the variable changes significantly across its domain.

There is a tendency to want more measurements of imprecise (or linguistic) data in order to compensate for lack of precision. Let us consider the case of two time signals that are to be sampled, digitized, and fast-Fourier-transformed so that one fast Fourier transform (FFT) is the input to a neural network and the other one is the desired output. If we have 100,000 simultaneously sampled data points for each variable and are dealing with spectra that have 128 points each (and another 128 points in the negative frequency range), dividing 100,000 points by 256 points per spectrum gives 390 complete spectra for each evaluation. The traditional approach with such FFTs is to average the 390 spectra to obtain an average spectrum with a high degree of confidence for each variable and to apply these two spectra to the neural network for training. A much better alternative would be to train the neural network using each of the 390 individual spectra, even though each of them is much less precise and would be considered "noisy" or perhaps "fuzzy." Subjecting the 128 components of the input and desired output vectors to "one-of-n" coding in the manner described above is another alternative that should be considered.

Fuzzy Postprocessing of Neural Network Outputs

A neural network can be trained to produce the desired final product, but there are often advantages to training the network to present intermediate values with postprocessing to obtain the desired results. The advantages are that the neural network may be easier to train and that the necessity for retraining if other outputs are desired can be avoided. An example of such a

postprocessor might be control of the electrical output of a gas-fired plant when there is competition for the gas with residential users and industrial users, both of which have a higher priority for the gas. A neural network with such inputs as current air temperature and at several earlier times and at several locations, overall demand for industrial products now and at several earlier times, the competitiveness of the products, plant efficiencies as a function of power output, and so on, could be trained to predict the available gas. However, intermediate values such as future temperatures at several locations and future industrial output may be more appropriate since they can reasonably be obtained using an ordinary neural network. However, the relationship between the availability of gas and the intermediate network outputs are fuzzy and should be treated as such.

Fuzzy Control of Backpropagation Learning

Numerous methods of speeding up the learning in backpropagation neural networks have been attempted with varying degrees of success. One of the most common methods have been to adjust the learning rate during the training using an adaptive method that satisfies some index of performance. (The "delta-bar-delta" training procedure is such a method.) Wang and Mendel (Wang, 1994) (Wang and Mendel, 1992) have shown that fuzzy systems may be viewed as a layered feedforward network and have developed a backpropagation algorithm for training this form of fuzzy system to match the input and desired output pairs of patterns or variables. Haykin (1994) has described a method in which an on-line fuzzy logic controller is used to adapt the learning parameters of a multilayer perceptron with backpropagation learning. The system uses the classical four-step fuzzy control process of (1) scaling and fuzzification of the crisp input, (2) development of a fuzzy rule base, (3) fuzzy inference using the fuzzy rule base, and (4) rescaling and defuzzification to give a crisp result or recommended action. The idea is to implement heuristics in the form of fuzzy *if/then* rules that are used for the purpose of achieving a faster rate of convergence. The heuristics (as is the case of almost all supervised training) are based on the behavior of the instantaneous sum of squared errors.

12.10 SURVEY OF ENGINEERING APPLICATIONS

Fuzzy neural networks aspiring to integrate neural learning with the knowledge representation capabilities of fuzzy systems have been actively investigated in recent years. A growing number of researchers in a number of fields have proposed and tested several types of fuzzy neurons. By far the greatest number has turned to the rather simple *AND* and *OR* neurons of Section 12.5 in building fuzzy neural networks. The networks are typically heteroge-

neous in order to best reflect the logic of a given problem. The layers and nodes of such fuzzy neural networks can be interpreted as a realization of fuzzy *if/then* rules.

Pedrycz and Rocha (1993) introduced a number of neurofuzzy models, using logic operators (*AND*, *OR*, *NOT*) encountered in the theory of fuzzy sets superimposed in neural structures. *Aggregation neurons* (*AND* and *OR* neurons) and *referential neurons* (for *matching*, *dominance*, *inclusion*) were designed using T norms and S norms and inhibitory and excitatory characteristics captured by embodying direct and complemented (negated) input signals. The researchers have proposed a number of topologies of neural networks put together with the use of these neurons and demonstrated straightforward relationships between the problem specificity and the resulting architecture of the network (Pedrycz, 1993).

Hirota and Pedrycz (1993a) have also proposed a distributed computational structure called *knowledge-based network* that allows for an explicit representation of domain classification knowledge. The knowledge-based network is composed of basic *AND* and *OR* neurons and has been used in pattern classification problems. Logic-based neurons have also been investigated in conjunction with new architectural aspects of fuzzy neural networks, including those aimed at representing and processing uncertainty associated with the input data (Pedrycz, 1993). Hybrids such as, for example, a multivariable hierarchical controller for an N-degrees-of-freedom robot manipulator for control tracking problems implemented as a fuzzy-neural network, whose purpose is to select activation levels for local regulators implemented as PD controllers, have also been developed and analyzed (Rueda and Pedrycz, 1994). Lin and Song (1994) have proposed a similar three-layer fuzzy neural network with different types of fuzzy neurons.

The terms *fuzzy-neural* or *neurofuzzy networks* very often in the literature refer to hybrid combinations of fuzzy logic and neural tools—for example, giving fuzzy inputs to a crisp network and extracting fuzzy outputs as well. Recently, Srinivasan (1994) has reported on a forecasting approach using fuzzy inputs to a neural network, in electric load forecasting problems. Expert knowledge represented by fuzzy rules is used for preprocessing input data fed to a neural network. The method effectively deals with trends and special events that occur annually. The fuzzy-neural network was trained on real data from a power system and evaluated for forecasting next-day load profiles based on forecast weather data and other parameters and according to the researchers has demonstrated very good performance.

A fuzzy-neural network approach developed by Cooley, Zhang, and Chen (1994) utilizes a hybrid consisting of a parameter-computing network, a converting layer, and a backpropagation-based one for classification problems with complex feature sets. The approach has been applied to satellite image classification and lithology determination. Lee and Wang (1994) have also proposed a neural network for classification problems with fuzzy inputs.

A fuzzy input is represented as an LR-type fuzzy set, and the network structure is automatically generated with the number of hidden nodes determined by the overlapping degree of training instances. Two sample problems, heart disease and knowledge-based evaluator, have been addressed by the researches to illustrate the working of the model. Sharpe et al. (1994) have also presented a hybrid method using fuzzy logic techniques to adapt a conventional network configuration criteria.

In another interesting hybrid application, fuzzy logic has been used by Hu and Hertz (1994) for controlling the learning processes of neural networks. Since the convergence of multilayer feedforward neural networks using the backpropagation training algorithm may be slow and uncertain due to the iterative nature of the dynamic process of finding the weight matrices with static control parameters, Hu and Hertz use a fuzzy logic controller during the course of training to adjust the learning rate dynamically according to the output error of a neuron and a set of heuristic rules. Comparative tests reported by the investigators have showed that such fuzzy backpropagation algorithms stabilized the training processes of these neural networks and, produced two to three times more converged tests than the conventional backpropagation algorithms. Kuo (1993) has also reported a new learning scheme which integrates the standard backpropagation learning algorithm and fuzzy rules, which are able to dynamically adjust the learning rate, momentum, and steepness of activation function.

A fuzzification layer to a conventional feedforward neural network has been added by Zhang and Morris (1994a, b) for on-line process fault diagnosis. The fuzzification layer converts the increment in each on-line measurement and controller output into three fuzzy sets: "*INCREASE*," "*STEADY*," and "*DECREASE*," with corresponding membership functions. The feedforward neural network then classifies abnormalities, represented by fuzzy increments in on-line measurements and controller outputs, into various categories.

Kwan and Cai (1994) have defined four types of fuzzy neurons similar to those we have seen in Section 12.2, and they have proposed a structure of a four-layer feedforward fuzzy neural network and its associated learning algorithm. The proposed four-layer fuzzy neural network performs well in several pattern recognition problems. In a biotechnology application, a five-layer fuzzy neural network was developed for the control of fed-batch cultivation of recombinant *Escherichia* (Ye et al., 1994).

Karayiannis and Pai (1994) have developed a family of fuzzy algorithms for learning vector quantization and introduced feedforward neural networks inherently capable of fuzzy classification of nonsparse or overlapping pattern classes. On the other hand, a three-layer radial basis function (RBF) network has been developed by Halgamuge et al. (1994) to extract rules and to identify the necessary membership functions of the inputs for a fuzzy classification system.

REFERENCES

Bezdek, J. C., Hybrid Modeling in Pattern Recognition and Control, *Knowledge-Based Systems*, Vol. 8, N. 8, pp. 359–371, 1995.

Carpenter, G., and Grossberg, S., Fuzzy ARTMAP: A Synthesis of Neural Networks and Fuzzy Logic for Supervised Categorization and Nonstationary Prediction, in *Fuzzy Sets, Neural Networks, and Soft Computing*, R. R. Yager and L. A. Zadeh, eds., Van Nostrand Reinhold, New York, 1994.

Cooley, D. H., Zhang, J., Chen, L., Possibility Function Based Fuzzy Neural Networks: Case Study, in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Vol. 1, IEEE, Piscataway, NJ, 1994, pp. 73–78.

Dubois, D., and Prade, H., *Fuzzy Sets and Systems: Theory and Applications*, Academic Press, Boston, 1980.

Gupta, M. M., Rao, D. H., eds., *Neuro-Control Systems Theory and Applications*, a selected reprint volume, IEEE Press, New York, 1994.

Gupta, M. M., and Knopf, G. K., Fuzzy Neural Network Approach to Control Systems, in *Analysis and Management of Uncertainty: Theory and Applications*, B. M. Ayyub, M. M. Gupta, and L. N. Kanal, eds., pp. 183–197, Elsevier, Amsterdam, 1992.

Gupta, M. M., and Qi, J., Theory of *T*-Norms and Fuzzy Inference Methods, *Fuzzy Sets and Systems*, Vol. 40, pp. 431–450, 1991.

Gupta, M. M., *Fuzzy Neural Networks: Theory and Applications*, *Proceedings of SPIE*, Vol. 2353, The International Society for Optical Engineering, 1994, pp. 303–325.

Halgamuge, S. K., Poechmueller, W., Pfeffermann, A., Schweikert, P., and Glesner, M., New Method for Generating Fuzzy Classification Systems Using RBF Neurons with Extended RCE Learning, in *Proceedings of the 1994 IEEE International Conference on Neural Networks*, 1994, pp. 1589–1593.

Haykin, S., *Neural Networks: A Comprehensive Foundation* (IEEE Computer Society Press), Macmillan, New York, 1994.

Hirota, K., and Pedrycz, W., Knowledge-Based Networks in Classification Problems, *Fuzzy Sets and Systems*, Vol. 59, No. 3, pp. 271–279, 1993a.

Hirota, K., and Pedrycz, W., Neurocomputations with Fuzzy Flip-Flops, in *Proceedings of the International Joint Conference on Neural Networks*, Vol. 2, pp. 1867–1870, 1993b.

Hu, Q. and Hertz, D. B., Fuzzy Logic Controlled Neural Network Learning, *Information Sciences Applications*, Vol. 2, N. 1, pp. 15–33, 1994.

Ishibuchi, H., Fujióka, R., and Tanaka, H., Neural Networks That Learn from Fuzzy If-Then Rules, *IEEE Transactions on Fuzzy Systems*, Vol. 1, No. 2, pp. 85–97, 1993.

Karayiannis, NB., and Pai, P.-I., Family of Fuzzy Algorithms for Learning Vector Quantization, in *Proceedings of the Artificial Neural Networks in Engineering Conference* (ANNIE '94), St. Louis, MO, November 13–16, 1994.

Kasuba, T., Simplified Fuzzy ARTMAP, *AI Expert*, Vol. 8, N. 11, pp. 18–25, November, 1993.

Keller, J. M., and Tahani, H., Implementation of Conjunctive and Disjunctive Fuzzy Logic Rules with Neural Networks, *International Journal of Approximate Reasoning*, Vol. 6, pp. 221–240, 1992.

Kosko, B., *Neural Networks and Fuzzy Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1992.

Kulkarni, A. D., Coca, P., Giridhar, G. B., and Bhatikar, Y., Neural Network Based Fuzzy Logic Decision System, in *Proceedings of World Congress on Neural Networks*, 1994 INNS Annual Meeting, San Diego, CA, June 5–9, 1994.

Kuo, R. J., Fuzzy Parameter Adaptation for Error Backpropagation Algorithm, in *Proceedings of the International Joint Conference on Neural Networks*, Vol. 3, 1993, pp. 2917–2920.

Kwan, H. K., and Cai, Y., Fuzzy Neural Network and Its Application to Pattern Recognition, *IEEE Transactions on Fuzzy Systems*, Vol. 2, No. 3, pp. 185–193, 1994.

Lee, H-M., Wang, W-T., Architecture of Neural Network for Fuzzy Teaching Inputs, *Proc. of the 5th Intern. Conf. on Tools with A.I., TAI '93*, Publ. by IEEE, Boston, MA, Nov. 8–11, 1993, pp. 285–288.

Likas, A., Blekas, K., and Stafylopatis, A., Application of the Fuzzy min–max Neural Network Classifier to Problems with Continuous and Discrete Attributes, in *Proceedings of the 4th IEEE Workshop on Neural Networks for Signal Processing* (NNSP '94), Ermioni, Greece, September 6–8, 1994.

Lin, J.-N., Song, S.-M., Novel Fuzzy Neural Network for the Control of Complex Systems, in *Proceedings of IEEE International Conference on Neural Networks*, Vol. 3, 1994, pp. 1668–1673.

Maeda, M., and Murakami, S., Steering Control and Speed Control of an Automobile with Fuzzy Logic, *Proc. of the 3rd IFSA Congress*, Seattle, Washington, Aug. 6–11, 1989, pp. 75–78.

NeuralWorks Manual, *Fuzzy ARTMAP Classification Network*, pp. NC-157–NC170.

Pedrycz, W., *Fuzzy Control and Fuzzy Systems*, second (extended) edition, John Wiley & Sons, New York, 1993.

Pedrycz, W., and Rocha, A. F., Fuzzy-Set Based Models of Neurons and Knowledge-Based Networks, *IEEE Transactions on Fuzzy Systems*, Vol. 1, No. 4, pp. 254–266, 1993.

Rueda, A., and Pedrycz, W., Hierarchical Fuzzy-Neural-PD Controller for Robot Manipulators, in *Proceedings of IEEE International Conference on Fuzzy Systems*, Vol. 1, 1994, pp. 673–676.

Saleem, R. M., and Postlethwaite, B. E., Comparison of Neural Networks and Fuzzy Relational Systems in Dynamic Modeling, *Proc. of the International Conference on CONTROL '94*, Vol. 2, No. 389, IEE Conference Publication, 1994, pp. 1448–1452.

Sharpe, R. N., Chow, M. Y., Briggs, S., and Windingland, L., Methodology Using Fuzzy Logic to Optimize Feedforward Artificial Neural Network Configurations, *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 24, No. 5, pp. 760–767, 1994.

Srinivasan, D., Liew, A. C., and Chang, C. S., Forecasting Daily Load Curves Using a Hybrid Fuzzy-Neural Approach, *IEE Proceedings Generation, Transmission and Distribution*, Vol. 141, N. 6, 1994, pp. 561–567.

Sugeno, M., and Nishida, M., Fuzzy Control of a Model Car, *Fuzzy Sets and Systems*, Vol. 16, pp. 103–113, 1985.

Terano, T., Asai, K., and Sugeno, M., *Fuzzy Systems Theory and Its Applications*, Academic Press, Boston, 1992.

Terano, T., Asai, K., and Sugeno, M., *Applied Fuzzy Systems*, Academic Press, Boston, 1994.

Travis, M., and Tsoukalas, L. H., Application of Fuzzy Logic Membership Functions to Neural Network Data Representation, Internal Report, University of Tennessee, Knoxville, TN, 1992. Included as Appendix A in Uhrig et al., Application of Neural Networks, EPRI Report TR-103443-P1-2, January 1994.

Wang, L.-X., *Adaptive Fuzzy Systems and Control*, Prentice-Hall, Englewood Cliffs, NJ, 1994.

Wang, L. X., and Mendel, J. M., Backpropagation Fuzzy Systems as Nonlinear Dynamic System Identifiers, in *IEEE International Conference on Fuzzy Systems*, San Diego, CA, 1992, pp. 1409–1418.

Werbos, P. J., Neurocontrol and Fuzzy Logic: Connections and Designs, *International Journal of Approximate Reasoning*, Vol. 6, pp. 185–219, 1992.

Ye, K., Jin, S., and Shimizu, K., Fuzzy Neural Network for the Control of High Cell Density Cultivation of Recombinant *Escherichia coli*, *Journal of Fermentation and Bioengineering*, Vol. 77, No. 6, pp. 663–673, 1994.

Zhang, J., Morris, A. J., On-Line Process Fault Diagnosis Using Fuzzy Neural Networks, *Intelligent Systems Engineering*, Vol. 3, No. 1, pp. 37–47, 1994a.

Zhang, J., Morris, A. J., Process Fault Diagnosis Using Fuzzy Neural Networks, in *Proceedings of the American Control Conference*, American Automatic Control Council, Vol. 1, 1994, pp. 971–975.

PROBLEMS

1. Explain the assumptions made and describe the forms of the dendritic input, the aggregation operator, and the activation function in the perceptron [Eq. (12.2-4)].

2. Besides summation, how else could the dendritic inputs to a neuron be aggregated? List at least three operators that could be used for aggregation.

3. Assuming a [0, 1] range for the input values to the fuzzy neuron shown in Figure 12.2, show that the output will also be in the [0, 1] range. What happens to the output of the synaptic modifications are made through a max operator or any other S norm? Can the same be said when the synaptic modification is done through a T norm?

4. Explain why and how a bias term may be incorporated in the fuzzy neurons described by Equations (12.5-3) and (12.5-4).

5. Consider a fuzzy neuron having as input the fuzzy set $A = 0.5/1 + 1.0/2 + 0.5/3$ and a weight fuzzy relation given by

$$w_{ij}(x, y) = 0.33/(2,5) + 0.5/(2,6) + 0.5/(2,7) + 0.5/(2,8) + 0.33/(2,9) + 0.33/(3,5) + 0.67/(3,6) + 1.0/(3,7) + 0.67/(3,8) + 0.33/(3,9) + 0.33/(4,5) + 0.5/(4,6) + 0.5/(4,7) + 0.5/(4,8) + 0.33/(4,9)$$

What is the dendritic input to the neuron's soma? State all assumptions clearly.

6. How can excitatory and inhibitory inputs be taken into account in the fuzzy neuron described in Problem 5?

7. Consider a three-layer fuzzy neural network having *AND* neurons in the hidden layer as shown in Figure 12.6*a*. Show that the network's output is given by Equation (12.6-3). Suppose next that the network has *OR* neurons in the middle layer as shown in Figure 12.6*b*. What is its output?

8. For the three-layer network of Figure 12.6*a* using probabilistic sum for S norm and product for the T norm, show that the rate of output change with respect to input weights is given by Equation (E12.1-7).

9. Derive an expression for the rate of output change with respect to input weights in Problem 8 when min and max are used for T norm and S norm, respectively.

10. Develop a fuzzy-neural network representation similar to the one shown in Figure 12.11 for the fuzzy algorithm described in Example 6.3 (rules given in (E6.3-1)).