CHAPTER 2

Introduction and Single-Layer Neural Networks

All of us have a highly interconnected set of some 10^{11} neurons to facilitate our reading, breathing, motion, and thinking. Each of the biological neurons has the complexity of a microprocessor. Remarkably, the human brain is a highly complex, nonlinear, and parallel computer. It has the capability to organize its structural constituents, that is, neurons, so as to perform certain computations many times faster than the fastest digital computer in existence today.

Specifically, the human brain consists of a large number of highly connected elements (approximately 10^4 connections per element) called neurons [Hagan *et al*., 1996]. For our purposes, these neurons have three principal components: the dendrites, the cell body, and the axon. The dendrites are tree-like receptive networks of nerve fibers that carry electrical signals into the cell body. The cell body effectively sums and thresholds these incoming signals. The axon is a single long fiber that carries the signal from the cell body out to other neurons. The point of contact between an axon of one cell and a dendrite of another cell is called a synapse. It is the arrangement of neurons and the strengths of the individual synapses, determined by a complex chemical process, that establishes the function of the neural network. Figure 2.1 shows a simplified schematic diagram of two biological neurons.

Actually, scientists have begun to study how biological neural networks operate. It is generally understood that all biological neural functions, including memory, are stored in the neurons and in the connections between them. Learning is viewed as the establishment of new connections between neurons or the modification of existing connections. Then, one may have the question: Although we have only a rudimentary understanding of biological neural networks, is it possible to construct a small set of simple artificial neurons and then train them to serve a useful function? The answer is "yes." This is accomplished using artificial neural networks, commonly referred to as neural networks, which have been motivated right from its inception by the recognition that the human brain computes in an entirely different way from the conventional digital computer. Figure 2.2 shows a simplified schematic diagram of

Fundamentals of Computational Intelligence: Neural Networks, Fuzzy Systems, and Evolutionary Computation, First Edition. James M. Keller, Derong Liu, and David B. Fogel.

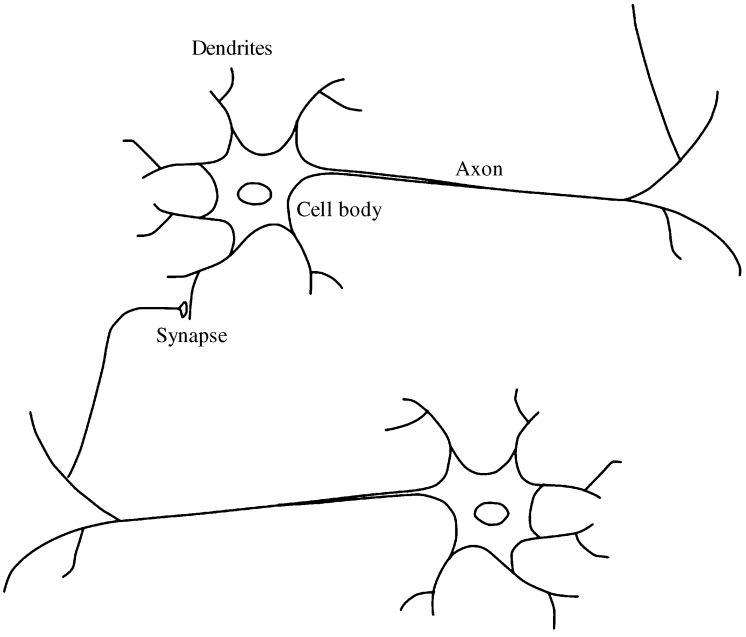

FIGURE 2.1 The schematic diagram of two biological neurons.

two artificial neurons. Here, the two artificial neurons are connected to be a simple artificial neural network and each artificial neuron contains some input and output signals.

The neurons that we consider here are not biological. They are extremely simple abstractions of biological neurons, realized as elements in a program or perhaps as circuits made of silicon. Networks of these artificial neurons do not have a fraction of the power of the human brain. However, they can be trained to perform useful functions. Note that even though biological neurons are very slow compared to electrical circuits, the brain is able to perform many tasks much faster than any conventional computer. One important reason is that the biological neural networks hold massively parallel structure and all of the neurons operate at the same time. Fortunately, the artificial neural networks share this parallel structure, which makes them useful in practice.

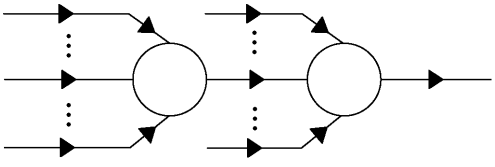

FIGURE 2.2 The schematic diagram of two artificial neurons.

Artificial neural networks do not approach the complexity of the brain. However, there are two main similarities between the biological neural networks and artificial neural networks. One is that the building blocks of both networks are simple computational devices (although artificial neurons are much simpler than biological neurons) that are highly interconnected. The other is that the connections between neurons determine the function of the network.

2.1 SHORT HISTORY OF NEURAL NETWORKS

Generally speaking, the history of neural networks has progressed through both conceptual innovations and implementation developments. However, these advancements seem to have occurred in fits and starts, rather than by steady evolution [Hagan *et al.*, 1996].

Some of the background work for the field of neural networks occurred in the late nineteenth and early twentieth centuries. The primarily interdisciplinary work was conducted by many famous scientists from the fields of physics, psychology, and neurophysiology. In this stage, the research of neural networks emphasized general theories of learning, vision, conditioning, and so on, and did not include specific mathematical models of the neuron operation.

Then, the modern view of neural networks began in the 1940s with the work of Warren McCulloch and Walter Pitts, who showed that networks of artificial neurons could, in principle, compute any arithmetic or logical function. Notice that this important work is often regarded as the origin of the neural network community. Then, scientists proposed a mechanism for learning in biological neurons.

The first practical application of neural networks came in the late 1950s, with the invention of the perceptron network and the associated learning rule [Rosenblatt, 1958]. In this stage, the great success brought large interest to the research of neural networks. However, it was later shown that the basic perceptron network could only solve a limited class of problems. At the same time, scientists introduced a new learning algorithm and used it to train an adaptive linear neural network, which is still in use today. In fact, it was similar in structure and capability to Rosenblatt's perceptron.

Unfortunately, Rosenblatt's networks suffered from the same inherent limitation of what class of problems could be learned. Though Rosenblatt was aware of the limitation and proposed new networks to overcome it, he was not able to modify the learning algorithm to accommodate training more complex networks. Therefore, many people believed that further research on neural networks was a dead end. What's more, considering the fact that there were no powerful digital computers to conduct experiment, the research on neural networks was largely suspended.

Interest in neural networks faltered during the late 1960s because of the lack of new ideas and powerful computers with which to experiment. However, during the 1980s, these impediments were gradually overcome. Hence, research on neural networks increased dramatically. In this stage, new personal computers and workstations, which rapidly grew in capability, became widely available. More importantly, some

new concepts were introduced. Among those, two novel concepts were most responsible for the rebirth of the neural network field. One was the use of statistical mechanics to explain the operation of a certain class of recurrent networks, which could be used as an associative memory. The other was the development of the backpropagation algorithm, which was introduced for helping to train multilayer perceptron networks [Rumelhart *et al*., 1986a, 1986b; Werbos, 1974, 1994].

These new developments reinvigorated the neural network community. In the last several decades, thousands of excellent papers have been written. The neural network technique has found many applications. Now, the field is buzzing with new theoretical and practical work. It is important to notice that many of the advances in neural networks have been related to new concepts, such as innovative architectures and training rules. In addition, the availability of powerful new computers, which test the new concepts, is also of great significance [Hagan *et al*., 1996].

It is apparent that a neural network derives its computing power through its massively parallel distributed structure and also its ability to learn and therefore generalize. The characteristic of generalization refers to the neural network producing reasonable outputs for inputs that were not encountered during training. These two information processing capabilities make it possible for neural networks to solve complex and large-scale problems that are currently intractable. However, in practice, neural networks cannot provide the solution by working individually. Instead, they need to be integrated into a consistent system engineering approach. Specifically, a complex problem of interest is decomposed into a number of relatively simple tasks, and some neural networks are assigned a subset of the tasks that match their inherent capabilities. However, it is important to recognize that we still have a long way to go before we can build a computer architecture that mimics a human brain. Consequently, how to achieve the true brain intelligence via artificial neural network is one of the main research objectives of scientists.

2.2 ROSENBLATT'S NEURON

Artificial neural networks, commonly referred to as "neural networks," represent a technology rooted in many disciplines: neurosciences, mathematics, statistics, physics, computer science, and engineering. Neural networks are potentially massively parallel distributed structures and have the ability to learn and generalize. Generalization denotes the neural network's production of reasonable outputs for inputs not encountered during learning process. Therefore, neural networks can be applied to diverse fields, such as modeling, time series analysis, pattern recognition, signal processing, and system control.

The neuron is the information processing unit of a neural network and the basis for designing numerous neural networks. A fundamental neural model consists of the following basic elements [Haykin, 2009]:

- A set of synapses, or connecting links, each of which is characterized by a weight or strength of its own.

- An adder for summing the input signals, weighted by the respective synaptic strengths of the neuron.
- An activation function for limiting the amplitude of the output of a neuron.
- An externally applied bias, which has the effect of increasing or lowering the net input of the activation function.

The most fundamental network architecture is a single-layer neural network, where the "single-layer" refers to the output layer of computation neurons. Note that we do not count the input layer of source nodes because no computation is performed there.

In neural network community, a signal flow graph is often used to provide a complete description of signal flow in a network. A signal flow graph is a network of directed links that are interconnected at certain points called nodes. The flow of signals in the various parts of the graph complies with the following three rules [Haykin, 1999]:

1. A signal flows along a link only in the direction indicated by the arrow on the link.
2. A node signal equals the algebraic sum of all signals entering the pertinent node via the incoming links.
3. The signal at a node is transmitted to each outgoing link originating from that node, with the transmission being entirely independent of the transfer functions of the outgoing links.

It should be pointed out that there are two different types of links, namely, synaptic links and activation links.

- The behavior of synaptic links is governed by a linear input–output relation. See Figure 2.3, the node signal x_j is multiplied by the synaptic weight w_{kj} to produce the node signal y_k, that is, $y_k = w_{kj}x_j$.
- The behavior of activation links is governed by a nonlinear input–output relation. This is illustrated in Figure 2.4, where $\phi(\cdot)$ is called the nonlinear activation function, that is, $y_k = \phi(x_j)$.

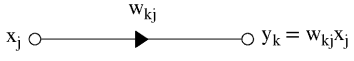

FIGURE 2.3 Illustration of the synaptic link.

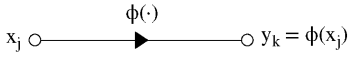

FIGURE 2.4 Illustration of the activation link.

Another expression method that can also be utilized to depict a network is called the architectural graph. Unlike the signal flow graph, the architectural graph possesses the following characteristics [Haykin, 1999]:

1. Source nodes supply input signals to the graph.
2. Each neuron is represented by a signal node called a computation node.
3. The communication links interconnecting the source and computation nodes of the graph carry no weight. They merely provide directions of signal flow in the graph.

Now, we introduce Rosenblatt's neuron [Haykin, 1999, 2009; Rosenblatt, 1958]. Rosenblatt's perceptron occupies a special place in the historical development of neural networks. It was the first algorithmically described neural network, which was built around a nonlinear neuron, namely, the McCulloch–Pitts model. Incidentally, the McCulloch–Pitts model is a neuron stated in recognition of the pioneering work done by McCulloch–Pitts. Rosenblatt's perceptron is an algorithm for learning a binary classifier: a function that maps its input x (a real-valued vector) to an output value f(x) (a single binary value):

$$f(x) = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.1}$$

where w is a vector of real-valued weights, $w \cdot x$ is the dot product (which here computes a weighted sum), and b (a real scalar) is the "bias," a constant term that does not depend on any input value. Figure 2.5 shows the signal flow graph of the Rosenblatt's perceptron.

Here, the harder limiter input (i.e., the induced local field) of the neuron is $w \cdot x + b$.

The value of f(x) (0 or 1) is used to classify x as either a positive or a negative instance, in the case of a binary classification problem. The decision rule for the classification is to assign the point represented by the inputs $x_1, x_2, \ldots, x_n$ to class $\aleph_1$ if the perceptron output y is +1 and to class $\aleph_2$ if it is 0. Note that for the case of two input variables, the decision boundary takes the form of a straight line in a two-dimensional plane (see Figure 2.6). If b is negative, then the weighted combination of inputs must produce a positive value greater than |b| in order to push the classifier

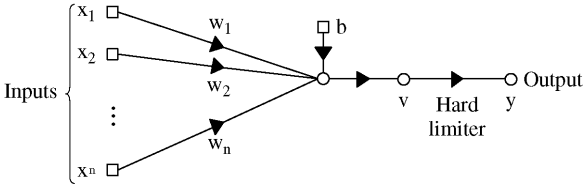

FIGURE 2.5 Signal flow graph of the perceptron.

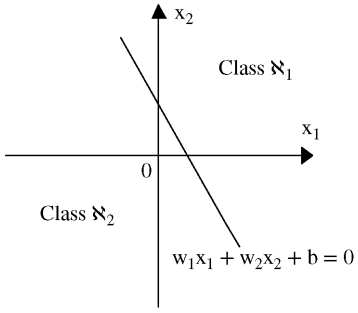

FIGURE 2.6 Illustration of the two-dimensional, two-class pattern classification problem.

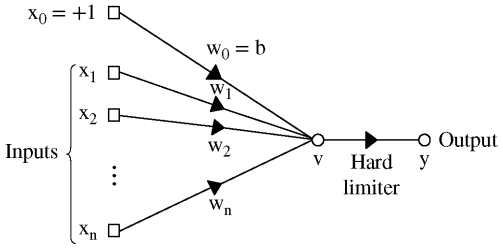

FIGURE 2.7 Equivalent signal flow graph of the perceptron.

neuron over the 0 threshold. Spatially, the bias alters the position (although not the orientation) of the decision boundary.

We now give an equivalent model of the perceptron described in Figure 2.5. Here, the bias b is viewed as a synaptic weight driven by a fixed input equal to +1. Then, the signal flow graph is shown in Figure 2.7.

2.3 PERCEPTRON TRAINING ALGORITHM

Now we consider the performance of the perceptron network and are in a position to introduce the perceptron learning rule. This learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

$$\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_q, t_q\} \tag{2.2}$$

Here p_i is an input to the network and t_i is the corresponding target output. As each input is applied to the network, the network output is compared with the target. The learning rule then adjusts the weights and biases of the network in order to move the network output closer to the target.

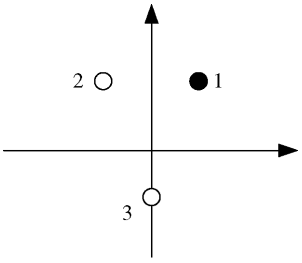

FIGURE 2.8 The test problem.

2.3.1 Test Problem

In our presentation of the perceptron learning rule, we will begin with a simple test problem and will experiment with possible rules to develop some intuition about how the rule should work. The input–target pairs for our test problem are

$$\left\{p_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}, \quad \left\{p_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\}, \quad \left\{p_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}$$

where p_i represents the input and t_i represents the corresponding target output. The problem is displayed graphically in Figure 2.8, where the two input vectors whose target is 0 are represented with a light circle ○, and the vector whose target is 1 is represented with a dark circle ●. This is a very simple problem, and we could almost obtain a solution by inspection. This simplicity will help us gain some intuitive understanding of the basic concepts of the perceptron learning rule.

The network for this problem should have two inputs and one output. To simplify our development of the learning rule, we will begin with a network without a bias. The network will then have just two parameters $w_{1,1}$ and $w_{1,2}$, as shown in Figure 2.9.

By removing the bias, we are left with a network whose decision boundary must pass through the origin. We need to ensure that this network is still able to solve the test problem. There must be an allowable decision boundary that can separate the

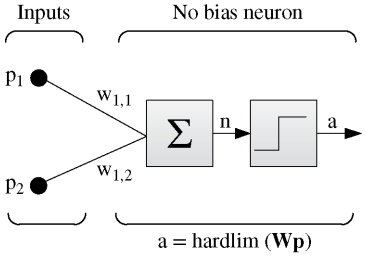

FIGURE 2.9 Test problem network.

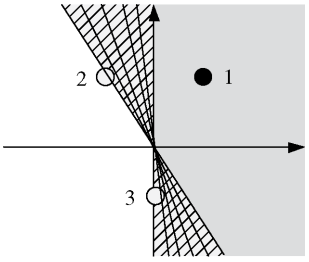

FIGURE 2.10 The boundaries.

vectors p_2 and p_3 from the vector p_1. Figure 2.10 illustrates that there are indeed an infinite number of such boundaries.

Figure 2.11 shows the weight vectors that correspond to the allowable decision boundaries. (Recall that the weight vector is orthogonal to the decision boundary.) We would like a learning rule that will find a weight vector that points to one of these directions. Remember that the length of the weight vector does not matter; only its direction is important.

2.3.2 Constructing Learning Rules

Training begins by assigning some initial values to the network parameters. In this case, we are training a two-input/single-output network without a bias, so we can only initialize its two weights. Here we set the elements of the weight vector ${}_1\mathrm{w}$ to the following randomly generated values:

$$ {}_1\mathrm{w}^{\mathrm{T}} = \begin{bmatrix} 1.0, & -0.8 \end{bmatrix} \tag{2.3} $$

We will now begin presenting the input vectors to the network and find the corresponding outputs, which we will call ϑ. We begin with p_1:

$$ \begin{aligned} \vartheta &= \mathrm{hardlim}({}_1\mathrm{w}^{\mathrm{T}}\mathrm{p}_1) = \mathrm{hardlim}\left([1.0 - 0.8] \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right) \\ \vartheta &= \mathrm{hardlim}(-0.6) = 0 \end{aligned} \tag{2.4} $$

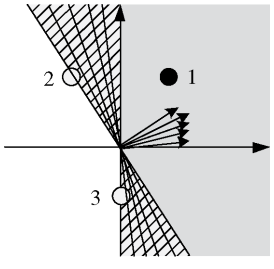

FIGURE 2.11 The weight vectors and boundaries.

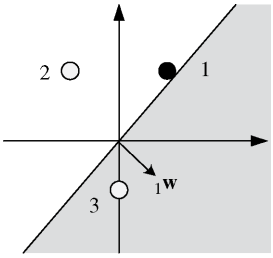

FIGURE 2.12 The classification result for the first input.

The network has not returned the correct value. The network output is 0, while the target response t_1 is 1.

We can see what happened in Figure 2.12. The initial weight vector results in a decision boundary that incorrectly classifies the vector p_1. We need to alter the weight vectors so that it points more toward p_1, so that in the future it has a better chance of classifying it correctly.

One approach would be to set ${}_1w$ equal to p_1. This is simple and would ensure that p_1 was classified properly in the future. Unfortunately, it is easy to construct a problem for which this rule cannot find a solution. Figure 2.13 shows a problem that cannot be solved with the weight vectors pointing directly at either of the two class 1 vectors. If we apply the rule ${}_1w = p$ every time one of these vectors is misclassified, the network's weights will simply oscillate back and forth and will never find a solution.

Another possibility would be to add p_1 to ${}_1w$. Adding p_1 to ${}_1w$ would make ${}_1w$ point more in the direction of p_1. Repeated presentations of p_1 would cause the direction of ${}_1w$ to asymptotically approach the direction of p_1. This rule can be stated:

$$\text{If } t = 1 \text{ and } \vartheta = 0, \text{ then } {}_1w^{\text{new}} = {}_1w^{\text{old}} + p \tag{2.5}$$

Applying this rule to our test problem results in new values for ${}_1w$:

$${}_1w^{\text{new}} = {}_1w^{\text{old}} + p_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} \tag{2.6}$$

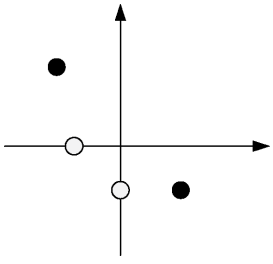

FIGURE 2.13 Another test problem that poses a challenge for setting ${}_1w = p_1$.

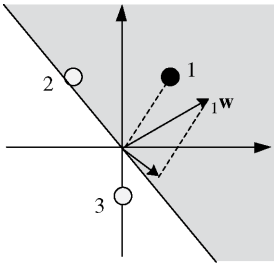

FIGURE 2.14 New values of the weight after adjusting by Eq. 2.6.

This operation is illustrated in Figure 2.14.

We now move on to the next input vector and continue making changes to the weights and cycling through the inputs until they are all classified correctly.

The next input vector is p_2. When it is presented to the network, we find

$$\begin{aligned} \vartheta &= \text{hardlim}({}_1w^T p_2) = \text{hardlim}\left([2.0\ 1.2]\begin{bmatrix}-1\\2\end{bmatrix}\right) \\ &= \text{hardlim}(0.4) = 1 \end{aligned} \tag{2.7}$$

The target t_2 associated with p_2 is 0 and the output ϑ is 1. A class 0 vector was misclassified as a 1.

Since we now find that we'd like to move the weight vector ${}_1w$ away from the input, we can simply change the addition in Eq. 2.5 to subtraction:

$$\text{If } t = 0 \text{ and } \vartheta = 1, \text{ then } {}_1w^{\text{new}} = {}_1w^{\text{old}} - p \tag{2.8}$$

If we apply this to the test problem, we find

$${}_1w^{\text{new}} = {}_1w^{\text{old}} - p_2 = \begin{bmatrix}2.0\\1.2\end{bmatrix} - \begin{bmatrix}-1\\2\end{bmatrix} = \begin{bmatrix}3.0\\-0.8\end{bmatrix} \tag{2.9}$$

which is illustrated in Figure 2.15.

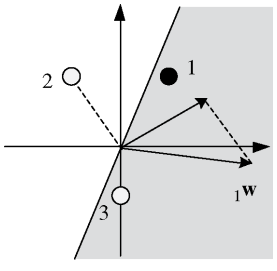

FIGURE 2.15 New values of the weight after adjusting by Eq. 2.9.

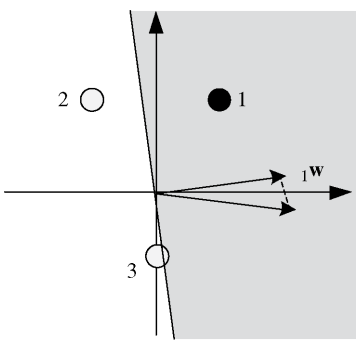

FIGURE 2.16 The classification result for the updated perceptron on all three vectors.

Now we present the third vector p_3:

$$\begin{aligned} \vartheta &= \text{hardlim}({}_1w^T p_3) = \text{hardlim}\left([3.0 - 0.8]\begin{bmatrix} 0 \\ -1 \end{bmatrix}\right) \\ &= \text{hardlim}(0.8) = 1 \end{aligned} \tag{2.10}$$

The current ${}_1w$ results in a decision boundary that misclassifies p_3. This is a situation for which we already have a rule, so ${}_1w$ will be updated again, according to Eq. 2.8:

$${}_1w^{new} = {}_1w^{old} - p_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix} \tag{2.11}$$

Figure 2.16 shows that the perceptron has finally learned to classify the three vectors properly. If we present any of the input vectors to the neuron, it will output the correct class for that input vector.

This brings us to our third and final rule: If it works, don't fix it.

$$\text{If } t = \vartheta, \text{ then } {}_1w^{new} = {}_1w^{old} \tag{2.12}$$

Here are the three rules, which cover all possible combinations of output and target values:

$$\begin{aligned} &\text{If } t = 1 \text{ and } \vartheta = 0, \text{ then } {}_1w^{new} = {}_1w^{old} + p \\ &\text{If } t = 0 \text{ and } \vartheta = 1, \text{ then } {}_1w^{new} = {}_1w^{old} - p \\ &\text{If } t = \vartheta, \text{ then } {}_1w^{new} = {}_1w^{old} \end{aligned} \tag{2.13}$$

2.3.3 Unified Learning Rule

The three rules in Eq. 2.13 can be rewritten as a single expression. First, we will define a new variable, the perceptron error e:

$$e = t - \vartheta \tag{2.14}$$

We can now rewrite the three rules of Eq. 2.13 as follows:

$$\begin{aligned} &\text{If } e = 1, \text{ then } {}_1w^{new} = {}_1w^{old} + p \\ &\text{If } e = -1, \text{ then } {}_1w^{new} = {}_1w^{old} - p \\ &\text{If } e = 0, \text{ then } {}_1w^{new} = {}_1w^{old} \end{aligned} \tag{2.15}$$

Looking carefully at the first two rules in Eq. 2.15, we can see that the sign of p is the same as the sign on the error e. Furthermore, the absence of p in the third rule corresponds to an e of 0. Thus, we can unify the three rules into a single expression:

$${}_1w^{new} = {}_1w^{old} + ep = {}_1w^{old} + (t - \vartheta)p \tag{2.16}$$

This rule can be extended to train the bias by noting that a bias is simply a weight whose input is always 1. We can thus replace the input p in Eq. 2.16 with the input to the bias, which is 1. The result is the perceptron rule for a bias:

$$b^{new} = b^{old} + e \tag{2.17}$$

2.3.4 Training Multiple-Neuron Perceptrons

The perceptron rule, as given by Eqs. 2.16 and 2.17, updates the weight vector of a single-neuron perceptron. We can generalize this rule for the multiple-neuron perceptron of Figure 2.17 as follows:

To update the ith row of the weight matrix, use

$${}_iw^{new} = {}_iw^{old} + e_ip \tag{2.18}$$

To update the ith element of the bias vector, use

$$b_i^{new} = b_i^{old} + e_i \tag{2.19}$$

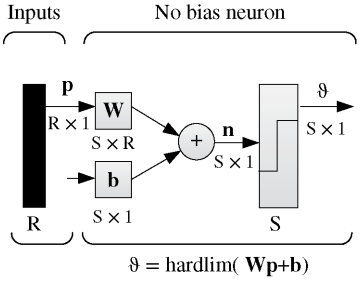

FIGURE 2.17 Test problem multiple-neuron network.

Perceptron rule: This rule can be written conveniently in matrix notation:

$$W^{new} = W^{old} + ep^T \tag{2.20}$$

and

$$b^{new} = b^{old} + e \tag{2.21}$$

To test the perceptron learning rule, consider the apple/orange recognition problem.

2.3.4.1 Problem Statement A producer dealer has a warehouse that stores a variety of fruits and vegetables. When fruit is brought to the warehouse, various types of fruit may be mixed together. The dealer wants a machine that will sort the fruit according to the type. There is a conveyer belt on which the fruit is loaded. This conveyer passes through a set of sensors that measure three properties of the fruit: *shape, texture*, and *weight*. These sensors are somewhat primitive. The shape sensor will output a 1 if the fruit is approximately round and a -1 if it is more elliptical. The texture sensor will output a 1 if the surface of the fruit is smooth and a -1 if it is rough. The weight sensor will output a 1 if the fruit is more than 1 lb and a -1 if it is less than 1 lb.

The three sensor outputs will then be input to a neural network. The purpose of the network is to decide which kind of fruit is on the conveyer, so that the fruit can be directed to the correct storage bin. To make the problem even simpler, let's assume that there are only two kinds of fruit on the conveyer: apples and oranges.

As each fruit passes through the sensors, it can be represented by a three-dimensional vector. The first element of the vector represents shape, the second element represents texture, and the third element represents weight:

$$p = \begin{bmatrix} \text{shape} \\ \text{texture} \\ \text{weight} \end{bmatrix} \tag{2.22}$$

Therefore, a prototype orange would be represented by

$$p_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} \tag{2.23}$$

and a prototype apple would be represented by

$$p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} \tag{2.24}$$

The neural network will receive one three-dimensional input vector for each fruit on the conveyer and must make a decision as to whether the fruit is an orange (p_1) or an apple (p_2).

For the apple and orange problem, the input/output prototype vectors will be

$$\left\{ p_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = [0] \right\}, \quad \left\{ p_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = [1] \right\} \tag{2.25}$$

(Note that we are using 0 as the target output for the orange pattern p_1 instead of -1, as was used in previous statements. This is because we are using the *hardlim* transfer function.)

Typically the weights and biases are initialized to small random numbers. Suppose that here we start with the initial weight matrix and bias:

$$W = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix}, \quad b = 0.5 \tag{2.26}$$

The first step is to apply the first input vector p_1 to the network:

$$\begin{aligned} \vartheta &= \text{hardlim}(Wp_1 + b) = \text{hardlim}\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 \right) \\ \vartheta &= \text{hardlim}(2.5) = 1 \end{aligned} \tag{2.27}$$

Then we calculate the error:

$$e = t_1 - \vartheta = 0 - 1 = -1 \tag{2.28}$$

The weight update is

$$\begin{aligned} W^{\text{new}} &= W^{\text{old}} + ep^{\text{T}} = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} \end{aligned} \tag{2.29}$$

The bias update is

$$b^{\text{new}} = b^{\text{old}} + e = 0.5 + (-1) = -0.5 \tag{2.30}$$

This completes the first iteration.

The second iteration of the perceptron rule is

$$\begin{aligned} \vartheta &= \text{hardlim}(Wp_2 + b) = \text{hardlim}\left(\begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5) \right) \\ &= \text{hardlim}(-1.5) = 0 \end{aligned} \tag{2.31}$$

$$e = t_2 - \vartheta = 1 - 0 = 1 \tag{2.32}$$

$$W^{new} = W^{old} + ep^T = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} + 1\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} \tag{2.33}$$

$$b^{new} = b^{old} + e = -0.5 + 1 = 0.5 \tag{2.34}$$

The third iteration begins again with the first input vector:

$$\begin{aligned} \vartheta &= \text{hardlim}(Wp_1 + b) = \text{hardlim}\left(\begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 \right) \\ &= \text{hardlim}(0.5) = 1 \end{aligned} \tag{2.35}$$

$$e = t_1 - \vartheta = 0 - 1 = -1 \tag{2.36}$$

$$\begin{aligned} W^{new} &= W^{old} + ep^T = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 2 & 0.5 \end{bmatrix} \end{aligned} \tag{2.37}$$

$$b^{new} = b^{old} + e = 0.5 - 1 = -0.5 \tag{2.38}$$

If you continue with more iterations, you will find that both input vectors will be classified correctly. The algorithm has converged to a solution, whose boundary correctly classifies the two input vectors.

We can now summarize the perceptron training algorithm. First, we define some variables:

- $y = f(x)$ denotes the *output* from the perceptron for an input vector x.
- b is the *bias* term.
- $D = \{(x(1), d(1)), \ldots, (x(s), d(s))\}$ is the *training set* of s samples, where
- x(k) is the n-dimensional input vector.
- d(k) is the desired output value of the perceptron for that input.

We show the values of the features as follows:

- $x_i(k)$ is the value of the ith feature of the kth training *input vector*.
- $x_0(k) = 1$.

To represent the weights:

- w_i is the ith value in the *weight vector*, to be multiplied by the value of the ith input feature.
- Because$x_0(k) = 1$, w_0 is effectively a learned bias that we use instead of the bias constant b.

To show the time dependence of w, we use the following:

- $w_i(k)$ is the weight i at time k.
- α is the *learning rate*, where $0 < \alpha \leq 1$.

Too high a learning rate makes the perceptron periodically oscillate around the solution unless additional steps are taken.

Then, the training algorithm goes iteratively in accordance with the following procedure [Haykin, 1999, 2009]:

1. Initialize the weights and the threshold. Weights may be initialized to 0 or to a small random value. In the example below, we use 0.
2. For each example k in our training set D, perform the following steps over the input x(k) and desired output d(k):
3. Calculate the actual output:

$$y(k) = f[w(t) \cdot x(k)] = f[w_0(k) + w_1(k)x_1(k) + w_2(k)x_2(k) + \cdots + w_n(k)x_n(k)] \tag{2.39}$$

4. Update the weights:

$$w_i(k+1) = w_i(k) + \alpha(d_i(k) - y_i(k))x_i(k) \tag{2.40}$$

for all features $0 \leq i \leq n$.

In the following section, the proof of the perceptron convergence algorithm in the case of $\alpha = 1$ and the fixed increment convergence theorem will be discussed.

2.4 THE PERCEPTRON CONVERGENCE THEOREM

Given the initial condition $w(0) = 0$, we now present the proof of the perceptron convergence algorithm [Haykin, 1999, 2009].

Let $\mathfrak{A}_1$ be the subspace of training vectors that belong to class $\aleph_1$, while $\mathfrak{A}_2$ be the subspace of training vectors that belong to class $\aleph_2$.

Suppose the perceptron incorrectly classifies the training vectors x(1), x(2), ... For example, $w^T(k)x(k) < 0$ for $k = 1, 2, \ldots$, but x(1), x(2), ... belongs to the

subset $\mathfrak{A}_1$. Considering $\alpha(k) = 1$, the second equation in (2.40) can be written as

$$w(k+1) = w(k) + x(k), \quad \text{for } x(k) \text{ belongs to class } \aleph_1 \tag{2.41}$$

Expanding (2.41), we can further obtain

$$w(k+1) = \sum_{j=1}^{k} x(j) \tag{2.42}$$

Because $\aleph_1$ and $\aleph_2$ are assumed to be linearly separable, there exists a solution w^0 such that $w^{0T}x(k) > 0$ for the training vectors $x(1), x(2), \ldots, x(k)$ belonging to the subset $\mathfrak{A}_1$. Define

$$\theta = \min_{x(k) \in \mathfrak{A}_1} w^{0T}x(k) \tag{2.43}$$

Then we can derive that

$$w^{0T}w(k+1) = \sum_{j=1}^{k} w^{0T}x(j) \geq k\theta \tag{2.44}$$

Using the Cauchy–Schwarz inequality, we can obtain

$$\left\|w^0\right\|^2 \|w(k+1)\|^2 \geq \left(w^{0T}w(k+1)\right)^2 \geq k^2\theta^2 \tag{2.45}$$

where $\|\cdot\|$ denotes the Euclidean norm of the enclosed argument vector. Therefore, we have

$$\|w(k+1)\|^2 \geq \frac{k^2\theta^2}{\|w^0\|^2} \tag{2.46}$$

On the other hand, we can rewrite (2.41) as

$$w(l+1) = w(l) + x(l), \quad \text{for } l = 1, 2, \ldots, k \text{ and } x(l) \in \mathfrak{A}_1 \tag{2.47}$$

Then, we can obtain

$$\|w(l+1)\|^2 = \|w(l)\|^2 + \|x(l)\|^2 + 2w^T(l)x(l) \tag{2.48}$$

According to the supposition $w^T(k)x(k) < 0$, we can further derive that

$$\|w(l+1)\|^2 \leq \|w(l)\|^2 + \|x(l)\|^2 \tag{2.49}$$

which is equivalent to

$$\|w(l+1)\|^2 - \|w(l)\|^2 \leq \|x(l)\|^2 \tag{2.50}$$

Define

$$\delta = \min_{x(l) \in \mathfrak{A}_1} \|x(l)\|^2 \tag{2.51}$$

Then, in accordance with Eq. 2.50, we can obtain

$$\|w(l+1)\|^2 \leq \sum_{l=1}^{k} \|x(l)\|^2 \leq k\delta \tag{2.52}$$

By making a comparison between Eqs. 2.46 and 2.52, we can find that the two equations hold simultaneously only if

$$k \leq k_{max} = \delta \frac{\left\| w^0 \right\|^2}{\theta^2} \tag{2.53}$$

Otherwise, the two equations conflict with each other. As a consequence, we have proved that for $\alpha(k) = 1$ for all k and $w(0) = 0$, and given that a solution vector w^0 exists, the rule for updating the synaptic weights of the perceptron must terminate after at most k_{max} iterations. Incidentally, the value of w^0 or k_{max} is not unique, which can be seen from Eqs. 2.43, 2.51, and 2.53.

Now, we expound the fixed increment convergence theorem for the perceptron as follows [Rosenblatt, 1962].

Theorem 2.1 Let the subsets of training vectors $\mathfrak{A}_1$ and $\mathfrak{A}_2$ be linearly separable. Let the inputs presented to the perceptron originate from these two subsets. The perceptron converges after some k_0 iterations, in the sense that

$$w(k_0) = w(k_0 + 1) = w(k_0 + 2) = \cdots$$

is a solution vector for $k_0 \leq k_{max}$.

2.5 COMPUTER EXPERIMENT USING PERCEPTRONS

Table 2.1 describes two classes of patterns in the two-dimensional plane.

Consider the single-layer perceptron illustrated in Figure 2.18. The corresponding input–output mapping is defined as

$$y = \phi(v) = \begin{cases} 1, & \text{if } w_1 x_1 + w_2 x_2 + b \geq 0 \\ 0, & \text{if } w_1 x_1 + w_2 x_2 + b < 0 \end{cases} \tag{2.54}$$

TABLE 2.1 Pattern Classification

x_1	x_2	d	Class
2	2	0	$\aleph_1$
1	−2	1	$\aleph_2$
−2	2	0	$\aleph_1$
−1	0	1	$\aleph_2$

Now, we describe the iterative process of the training algorithm as follows, with the purpose of classifying the patterns:

1. Set $w(0) = 0$.
2. Compute

$$y(0) = \phi\left(\begin{bmatrix}0 & 0 & 0\end{bmatrix}\begin{bmatrix}1 & 2 & 2\end{bmatrix}^T\right) = \phi(0) = 1$$

Since the actual response is not equal to the desired response, we update the weight and bias as

$$w(1) = \begin{bmatrix}0 & 0 & 0\end{bmatrix}^T + (0-1)\begin{bmatrix}1 & 2 & 2\end{bmatrix}^T = \begin{bmatrix}-1 & -2 & -2\end{bmatrix}^T$$

3. Compute

$$y(1) = \phi\left(\begin{bmatrix}-1 & -2 & -2\end{bmatrix}\begin{bmatrix}1 & 1 & -2\end{bmatrix}^T\right) = \phi(1) = 1$$

The actual response is equal to the desired response, so we do not need to update the weight and bias.

4. Compute

$$y(2) = \phi\left(\begin{bmatrix}-1 & -2 & -2\end{bmatrix}\begin{bmatrix}1 & -2 & 2\end{bmatrix}^T\right) = \phi(-1) = 0$$

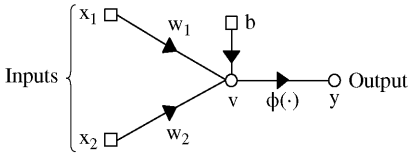

FIGURE 2.18 Structure of the single-layer perceptron.

Here, the actual response is equal to the desired response. Therefore, we do not need to update the weight and bias.

5. Compute

$$y(3) = \phi\left(\begin{bmatrix} -1 & -2 & -2 \end{bmatrix}\begin{bmatrix} 1 & -1 & 0 \end{bmatrix}^T\right) = \phi(1) = 1$$

Since the actual response is also equal to the desired response, we keep the weight and bias in their present values.

6. Compute

$$y(4) = \phi\left(\begin{bmatrix} -1 & -2 & -2 \end{bmatrix}\begin{bmatrix} 1 & 2 & 2 \end{bmatrix}^T\right) = \phi(-9) = 0$$

We see that the actual response is equal to the desired response. Thus, we do not need to update the weight and bias. Besides, we can further observe that the weight and bias here can make the actual response equal to the desired response for all input patterns. Accordingly, no update is required.

By considering

$$-2x_1 - 2x_2 - 1 = 0 \tag{2.55}$$

we can obtain a line

$$x_2 = -x_1 - \frac{1}{2} \tag{2.56}$$

to classify the inputs patterns in Table 2.1, as shown in Figure 2.19.

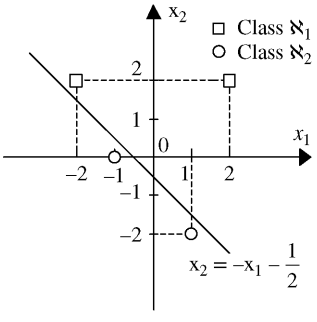

FIGURE 2.19 Illustration of the computer experiment that creates a line to separate the two classes.

2.6 ACTIVATION FUNCTIONS

In biologically inspired neural networks, the activation function is usually an abstraction representing the rate of action potential firing in the cell. In its simplest form, this function is binary, that is, the neuron is either firing or not.

The activation function defines the output of a neuron in terms of the induced local field [Haykin, 2009]. In this part, we identify two basic types of activation functions: a threshold function and a sigmoid function.

2.6.1 Threshold Function

The threshold function is defined as

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq 0; \\ 0, & \text{if } v < 0 \end{cases} \tag{2.57}$$

In engineering, this form of a threshold function is commonly referred to as a Heaviside function. See Figure 2.20.

The output of neuron m employing such a threshold function is expressed as

$$y_m = \begin{cases} 1, & \text{if } v_m \geq 0 \\ 0, & \text{if } v_m < 0 \end{cases} \tag{2.58}$$

where

$$v_m = \sum_{i=1}^{n} w_{mi} x_i + b_m \tag{2.59}$$

Notice in this model, the output of a neuron takes on the value of 1 if the induced local field of that neuron is nonnegative, and 0 otherwise. Such a neuron is referred to as the McCulloch–Pitts model.

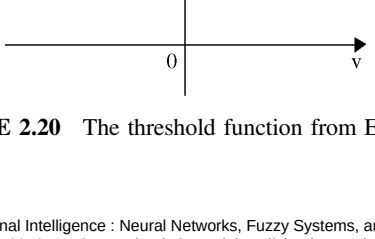

FIGURE 2.20 The threshold function from Eq. 2.57.

2.6.2 Sigmoid Function

The sigmoid function is the most common form of activation function used in the construction of neural networks. It is a strictly increasing function, which holds an excellent balance between linear and nonlinear behavior. The logistic function, a typical example of such function, is defined by

$$\phi(v) = \frac{1}{1 + e^{-av}} \tag{2.60}$$

where a is the slope parameter of the sigmoid function. It is depicted in Figure 2.21. Note that a threshold function assumes the value of 0 or −1, while a sigmoid function assumes a continuous range of values from 0 to 1. Another important property of the sigmoid function is that it is differentiable, whereas the threshold function is not.

In (2.57) and (2.60), the activation functions range from 0 to +1. However, it is sometimes desirable to have the activation function range from −1 to 1. In this case, the threshold function can be defined as the signum function, which is formulated as

$$\phi(v) = \begin{cases} 1, & \text{if } v > 0 \\ 0, & \text{if } v = 0 \\ -1, & \text{if } v < 0 \end{cases} \tag{2.61}$$

and described in Figure 2.22.

The corresponding form of a sigmoid function is hyperbolic tangent function defined as

$$\phi(v) = \tanh(v) = \frac{e^{v} - e^{-v}}{e^{v} + e^{-v}} \tag{2.62}$$

and plotted in Figure 2.23. It allows the activation function to assume negative values and can bring practical benefits.

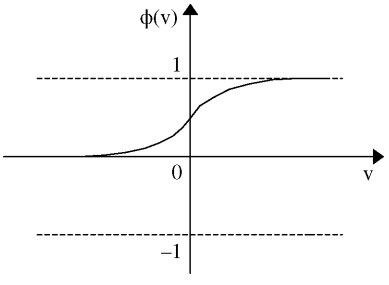

FIGURE 2.21 The logistic function from Eq. 2.60.

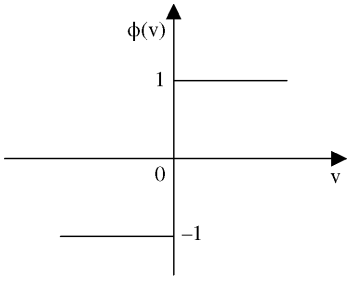

FIGURE 2.22 The signum function.

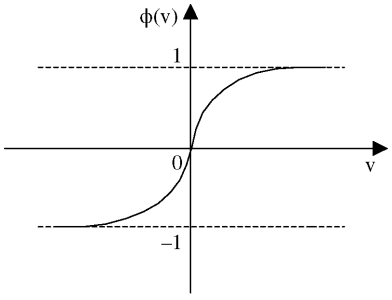

FIGURE 2.23 The hyperbolic tangent function.

EXERCISES

2.1. Suppose the inputs applied to neuron j, which is built around the McCulloch–Pitts model, are 10, −20, 4, −2. The synaptic weights of connecting to the neuron are 0.8, 0.2, −1, −0.9. Given that the externally applied bias is 0, compute the induced local field and the output of neuron j, respectively.

2.2. Study how the graph of the logistic function is affected by varying the value of slope parameter a in Eq. 2.60. Calculate the slope of the logistic function at origin. Then point out what the logistic function will be when the slope parameter approaches infinity.

2.3. The limiting values of the algebraic sigmoid function

$$\phi(v) = \frac{v}{\sqrt{1+v^2}}$$

are −1 and +1. Show that the derivative of $\phi(v)$ with respect to v is given by

$$\frac{d\phi}{dv} = \frac{\phi^3(v)}{v^3}$$

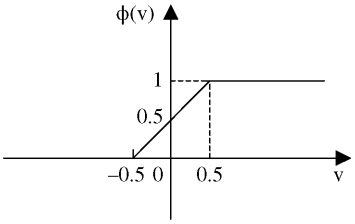

FIGURE 2.24 The piecewise-linear function of Exercise 2.4.

Calculate the value of the derivative at the origin.

2.4. Another activation function

$$\phi(v) = \begin{cases} 0, & \text{if } v \le -\dfrac{1}{2} \\ v, & \text{if } -\dfrac{1}{2} < v < \dfrac{1}{2} \\ 1, & \text{if } v \ge \dfrac{1}{2} \end{cases}$$

is depicted in Figure 2.24 and is called the piecewise-linear function. Here, the amplification factor inside the linear region of operation is assumed to be unity. Based on this expression, consider the activation functions presented in Figures 2.25 and 2.26 and, and study the following problems, respectively.

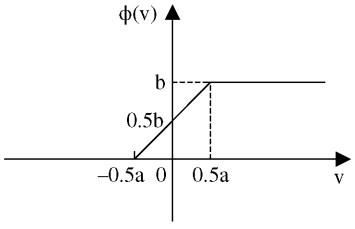

FIGURE 2.25 The activation function of Exercise 2.4 (case 1).

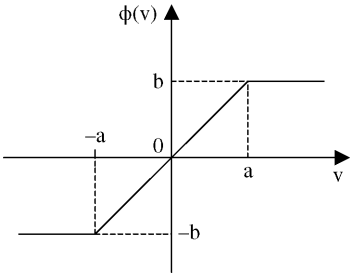

FIGURE 2.26 The activation function of Exercise 2.4 (case 2).

TABLE 2.2 Pattern Classification of Exercise 2.6

x_1	x_2	d	Class
0	2	1	$\aleph_1$
1	0	1	$\aleph_1$
0	−2	0	$\aleph_2$
2	0	0	$\aleph_2$

1. Formulate $\phi(v)$ as a function of v.
2. Show what will happen to $\phi(v)$ if a is allowed to approach zero.

2.5. Suppose that in the signal flow graph of the perceptron shown in Figure 2.7, the hard limiter is prescribed as the sigmoidal nonlinearity:

$$\phi(v) = \tanh\left(\frac{v}{2}\right)$$

where v is the induced local field. The classification decisions made by the perceptron are defined as

➢ observation vector x that belongs to class $\aleph_1$ if the output $y > \xi$, where ξ is a threshold; otherwise, x belongs to class $\aleph_2$.

Show that the decision boundary so constructed is a hyperplane.

2.6. Table 2.2 describes the two classes of patterns in the two-dimensional plane. Classify the two classes by using the single-layer perceptron. Given the detailed iterative process of the training algorithm, draw the separating line on the x_1, x_2 plane.

2.7. Show the single-layer perceptron can classify the patterns described in Table 2.3 successfully. However, a basic limitation of the perceptron is

TABLE 2.3 Pattern Classification of Exercise 2.7

x_1	x_2	d
0	0	0
0	1	0
1	0	0
1	1	1

that it cannot implement the EXCLUSIVE OR function. Explain the reason for this limitation.

2.8. Consider two one-dimensional, Gaussian-distributed classes $\aleph_1$ and $\aleph_2$ that have a common variance equal to 1. Their mean values are $\rho_1 = -10$, $\rho_2 = 10$, respectively. These two classes are essentially linearly separable. Design a classifier that separates these two classes.