CHAPTER 3

Multilayer Neural Networks and Backpropagation

Rosenblatt's perceptron that we studied in Chapter 2 is basically a single-layer neural network and it is limited to the classification of linearly separable patterns. In this chapter, in order to overcome the practical limitations of the perceptron, we look to a new neural network structure called a multilayer perceptron.

The following are the basic features of multilayer perceptrons [Haykin, 2009]:

- Each neuron model in the network includes a nonlinear activation function that is differentiable.
- The network contains one or more layers that are hidden from both the input and output modes.
- The network holds a high degree of connectivity, the extent of which is determined by synaptic weights of the network.

The architectural graph of a multilayer perceptron with two hidden layers and an output layer is depicted in Figure 3.1.

A computationally effective method for training the multilayer perceptrons is the backpropagation algorithm [Rumelhart et al., 1986a, 1986b; Werbos, 1974, 1994], which is regarded as a landmark in the development of neural network.

3.1 UNIVERSAL APPROXIMATION THEORY

A multilayer perceptron trained with the backpropagation algorithm can be considered as a practical means for performing a nonlinear input–output mapping of a general nature. Let n_0 denote the number of input (source) nodes of a multilayer perceptron, and let $N = n_l$ denote the number of neurons in the output layer of the network. Then, the input–output relationship of the network defines a mapping from an n_0-dimensional Euclidean input space to an N-dimensional Euclidean output

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

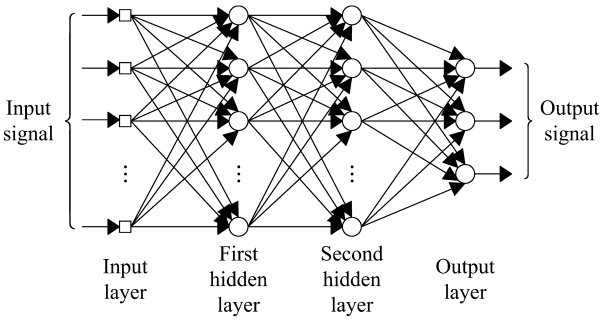

FIGURE 3.1 Architectural graph of a multilayer perceptron with two hidden layers.

space, which is infinitely continuously differentiable when the activation function is likewise continuously differentiable. The universal approximation theorem for a nonlinear input–output mapping is directly applicable to multilayer perceptrons.

Theorem 3.1 [Haykin, 1999] Let $\phi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{n_0} denote the n_0-dimensional unit hypercube $[0, 1]^{n_0}$. The space of continuous functions on I_{n_0} is denoted by $C(I_{n_0})$. Then, given any function $f \in C(I_{n_0})$ and $\varepsilon > 0$, there exist an integer n_1 and sets of real constants μ_i and b_i, and w_{ij}, where $i = 1, 2, \ldots, n_1$ and $j = 1, 2, \ldots, n_0$ such that we may define

$$F(x_1, x_2, \ldots, x_{n_0}) = \sum_{i=1}^{n_1} \mu_i \phi\left(\sum_{j=1}^{n_0} w_{ij} x_j + b_i\right) \tag{3.1}$$

as an approximate realization of the function $f(\cdot)$; that is,

$$\left|F(x_1, x_2, \ldots, x_{n_0}) - f(x_1, x_2, \ldots, x_{n_0})\right| < \varepsilon \tag{3.2}$$

for all $x_1, x_2, \ldots, x_{n_0}$ that lie in the input space.

Here, we notice that the hyperbolic tangent function used as the nonlinearity in a neural model for the construction of a multilayer perceptron is indeed a nonconstant, bounded, and monotone-increasing function. Therefore, it satisfies the conditions imposed on the function $\phi(\cdot)$. In addition, Eq. 3.1 represents the output of a multilayer perceptron described as follows:

- The network has n_0 input nodes and a single hidden layer consisting of n_1 neurons, while the inputs are $x_1, x_2, \ldots, x_{n_0}$.
- Hidden neuron i has synaptic weights $w_{i1}, w_{i2}, \ldots, w_{in_0}$ and bias b_i.
- The network output is a linear combination of the outputs of the hidden neurons, with $\mu_1, \mu_2, \ldots, \mu_{n_1}$ being the synaptic weights of the output layer.

Note that the universal approximation theorem is an existence theorem in the sense that it provides the mathematical justification for the approximation of an arbitrary continuous function as opposed to an exact representation.

3.2 THE BACKPROPAGATION TRAINING ALGORITHM

3.2.1 The Description of the Algorithm

Now, we present the backpropagation algorithm.

Consider Figure 3.2, the neuron j is fed by a set of function signals produced by a layer of neurons to its left. The induced local field $v_j(k)$ produced at the input of the activation function associated with neuron j is

$$v_j(k) = \sum_{i=0}^{n} w_{ji}(k) y_i(k) \tag{3.3}$$

Here, n is the total number of inputs (excluding the bias) applied to neuron j. Note the synaptic weight w_{j0} related to the fixed input $y_0 = +1$ equals the bias b_j applied to neuron j. The function signal $y_j(k)$ appearing at the output of neuron j at iteration k is

$$y_j(k) = \phi_j(v_j(k)) \tag{3.4}$$

Then, the error signal produced at the output of neuron j is defined by

$$e_j(k) = d_j(k) - y_j(k) \tag{3.5}$$

where $d_j(k)$ is the corresponding desired signal. We define the instantaneous error energy of neuron j as

$$E_j(k) = \frac{1}{2} e_j^2(k) \tag{3.6}$$

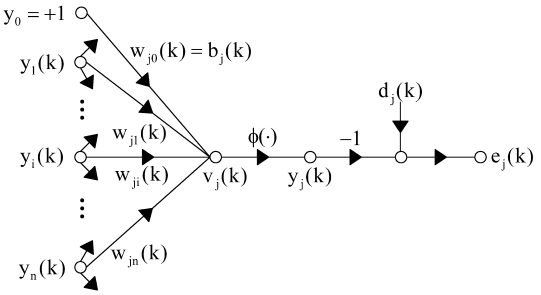

FIGURE 3.2 Signal flow graph highlighting the details of output neuron j.

By summing the error energy contributions of all the neurons in the output layer, we express the total instantaneous error energy of the whole network as the following form:

$$E(k) = \sum_j E_j(k) = \frac{1}{2}\sum_j e_j^2(k) \tag{3.7}$$

Let

$$\mathscr{E} = \{x(k), d(k)\}_{k=1}^{K} \tag{3.8}$$

be the training sample used to train the network. Basing on the total instantaneous error energy given in Eq. 3.7, we define the error energy averaged over the training sample, or the empirical risk as

$$\overline{E}(k) = \frac{1}{K}\sum_{k=1}^{K} E(k) = \frac{1}{2K}\sum_{k=1}^{K}\sum_j e_j^2(k) \tag{3.9}$$

where K denotes the number of examples that the training sample consists.

The backpropagation algorithm applied a correction $\Delta w_{ji}(k)$ to the synaptic weight $w_{ji}(k)$. Therefore, it is important to compute the partial derivative $\partial E(k)/\partial w_{ji}(k)$. According to the chain rule of calculus, we can express the gradient as follows:

$$\frac{\partial E(k)}{\partial w_{ji}(k)} = \frac{\partial E(k)}{\partial e_j(k)}\frac{\partial e_j(k)}{\partial y_j(k)}\frac{\partial y_j(k)}{\partial v_j(k)}\frac{\partial v_j(k)}{\partial w_{ji}(k)} \tag{3.10}$$

First, considering Eq. 3.6, we can find that

$$\frac{\partial E(k)}{\partial e_j(k)} = e_j(k) \tag{3.11}$$

Then, differentiating both sides of Eq. 3.5 with respect to $y_j(k)$, we obtain

$$\frac{\partial e_j(k)}{\partial y_j(k)} = -1 \tag{3.12}$$

Next, differentiating Eq. 3.4 with respect to $v_j(k)$, we can get

$$\frac{\partial y_j(k)}{\partial v_j(k)} = \phi_j'(v_j(k)) \tag{3.13}$$

At last, from Eq. 3.3, we derive that

$$\frac{\partial v_j(k)}{\partial w_{ji}(k)} = y_i(k) \tag{3.14}$$

Substituting Eqs. 3.11–3.14 to Eq. 3.10 yields

$$\frac{\partial E(k)}{\partial w_{ji}(k)} = -e_j(k)\phi_j'(v_j(k))y_i(k) \tag{3.15}$$

The correction $\Delta w_{ji}(k)$ applied to $w_{ji}(k)$ is defined by the delta rule, that is,

$$\Delta w_{ji}(k) = -\alpha \frac{\partial E(k)}{\partial w_{ji}(k)} \tag{3.16}$$

where α is the learning rate parameter of the backpropagation algorithm. Combining Eqs. 3.15 and 3.16, we have

$$\Delta w_{ji}(k) = \alpha \delta_j(k) y_i(k) \tag{3.17}$$

where the local gradient $\delta_j(k)$ is defined by

$$\begin{aligned} \delta_j(k) &= -\frac{\partial E(k)}{\partial v_j(k)} \\ &= -\frac{\partial E(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \\ &= e_j(k)\phi_j'(v_j(k)) \end{aligned} \tag{3.18}$$

From Eq. 3.18, we observe that the local gradient $\delta_j(k)$ for output neuron j is equal to the product of the corresponding error signal $e_j(k)$ for the neuron and the derivative $\phi_j'(v_j(k))$ of the associated activation function. We have two cases to be considered to compute it according to the location of neuron j, as shown in the following [Haykin, 2009]:

Case 1 Neuron j is an output node. When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. We can use Eq. 3.5 to compute the error signal $e_j(k)$ associated with this neuron. After that, we may compute the local gradient $\delta_j(k)$ straightforwardly by using Eq. 3.18.

Case 2 Neuron j is a hidden node. When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron. Then, the backpropagation algorithm becomes complicated. The error signal for a hidden neuron would have to be determined recursively and working backward in terms of the error signals of all the neurons to which that hidden neuron is directly connected.

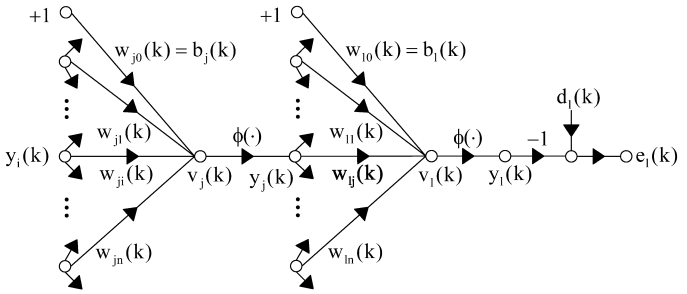

FIGURE 3.3 Signal flow graph highlighting the details of output neuron l connected to the hidden neuron j.

See Figure 3.3, where neuron j is a hidden node of the network. The error signal at the output of neuron l at iteration k is defined by

$$\begin{aligned} e_l(k) &= d_l(k) - y_l(k) \\ &= d_l(k) - \phi_l(v_l(k)) \end{aligned} \tag{3.19}$$

where

$$v_l(k) = \sum_{j=0}^{n} w_{lj}(k) y_j(k) \tag{3.20}$$

Notice in Eq. 3.20, the synaptic weight $w_{l0}(k)$ is equal to the bias $b_l(k)$ applied to neuron l, while the corresponding input is +1. Here, the instantaneous sum of squared errors of the network is

$$E(k) = \frac{1}{2} \sum_{l} e_l^2(k) \tag{3.21}$$

Then, we can obtain

$$\begin{aligned} \frac{\partial E(k)}{\partial y_j(k)} &= \sum_{l} e_l \frac{\partial e_l(k)}{\partial y_j(k)} \\ &= \sum_{l} e_l \frac{\partial e_l(k)}{\partial v_l(k)} \frac{\partial v_l(k)}{\partial y_j(k)} \end{aligned} \tag{3.22}$$

From Eq. 3.19, we have

$$\frac{\partial e_l(k)}{\partial v_l(k)} = -\phi_l'(v_l(k)) \tag{3.23}$$

Besides, according to Eq. 3.20, we can find that

$$\frac{\partial v_l(k)}{\partial y_j(k)} = w_{lj}(k) \tag{3.24}$$

Substituting Eqs. 3.23 and 3.24 to Eq. 3.22, we derive the partial derivative:

$$\begin{aligned} \frac{\partial E(k)}{\partial y_j(k)} &= -\sum_l e_l \phi_l'(v_l(k)) w_{lj}(k) \\ &= -\sum_l \delta_l(k) w_{lj}(k) \end{aligned} \tag{3.25}$$

where $\delta_l(k) = e_l \phi_l'(v_l(k))$. Therefore, considering Eqs. 3.13 and 3.18, we can further obtain

$$\begin{aligned} \delta_j(k) &= -\frac{\partial E(k)}{\partial v_j(k)} \\ &= -\frac{\partial E(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \\ &= \phi_j'(v_j(k)) \sum_l \delta_l(k) w_{lj}(k) \end{aligned} \tag{3.26}$$

Now, we summarize the relations that we have derived for the backpropagation algorithm. According to Eq. 3.17, the correction $\Delta w_{lj}(k)$ applied to the synaptic weight connecting neuron i to neuron j is given by the following delta rule:

$$\Delta w_{ji}(k) = \begin{cases} \alpha e_j(k) \phi_j'(v_j(k)) y_i(k), & \text{if neuron j is an output node} \\ \alpha \phi_j'(v_j(k)) \sum_l \left(e_l \phi_j'(v_l(k)) w_{lj}(k) \right) y_i(k), & \text{if neuron j is a hidden node} \end{cases} \tag{3.27}$$

From Eq. 3.27, we can clearly find that the computation of the correction $\Delta w_{ji}(k)$ requires knowledge of the derivative of the activation function $\phi(\cdot)$. Evidently, we require the function $f(\cdot)$ to be continuous to ensure its derivative exists. Actually, differentiability is the only requirement that an activation function must satisfy. The logistic function and hyperbolic tangent function are two representative instances.

1. *Logistic function.* According to the definition of logistic function given in (2.60), we can express the induced local field of neuron j as

$$\phi_j(v_j(k)) = \frac{1}{1 + \exp(-a v_j(k))} \tag{3.28}$$

where $a > 0$ is an adjustable parameter. By using Eq. 3.4, we can write the derivative $\phi_j'(v_j(k))$ as follows:

$$\begin{aligned}\phi_j'(v_j(k)) &= \frac{a\exp(-av_j(k))}{(1+\exp(-av_j(k)))^2} \\ &= ay_j(k)(1-y_j(k))\end{aligned} \tag{3.29}$$

Let $o_j(k)$ be the jth element of the output vector of the multilayer perceptron. When the neuron j is located in the output layer, we have

$$y_j(k) = o_j(k) \tag{3.30}$$

Then, according to Eqs. 3.18 and 3.29, we obtain the local gradient for neuron j as

$$\begin{aligned}\delta_j(k) &= e_j(k)\phi_j'(v_j(k)) \\ &= a(d_j(k)-o_j(k))o_j(k)(1-o_j(k))\end{aligned} \tag{3.31}$$

When the neuron j is located in the hidden layer, from Eqs. 3.26 and 3.29, the local gradient can be expressed as

$$\begin{aligned}\delta_j(k) &= \phi_j'(v_j(k))\sum_l \delta_l(k)w_{lj}(k) \\ &= ay_j(k)(1-y_j(k))\sum_l \delta_l(k)w_{lj}(k)\end{aligned} \tag{3.32}$$

2. *Hyperbolic tangent function.* Another commonly used sigmoidal nonlinearity is the hyperbolic tangent function, which in general, can be defined as

$$\phi_j(v_j(k)) = a\tan h(bv_j(k)) \tag{3.33}$$

where a and b are positive constants. Considering Eq. 3.4, we obtain the derivative of the hyperbolic tangent function as

$$\begin{aligned}\phi_j'(v_j(k)) &= ab\,\mathrm{sech}^2(bv_j(k)) \\ &= ab(1-\tan h^2(bv_j(k))) \\ &= \frac{b}{a}(a-y_j(k))(a+y_j(k))\end{aligned} \tag{3.34}$$

Then, using Eqs. 3.18 and 3.34, we derive the local gradient as

$$\begin{aligned}\delta_j(k) &= e_j(k)\phi_j'(v_j(k)) \\ &= \frac{b}{a}(d_j(k)-o_j(k))(a-o_j(k))(a+o_j(k))\end{aligned} \tag{3.35}$$

when the neuron j is an output node. Similarly, from Eqs. 3.26 and 3.34, the local gradient is

$$\begin{aligned}\delta_j(k) &= \phi_j'(v_j(k)) \sum_l \delta_l(k) w_{lj}(k) \\ &= \frac{b}{a}(a - y_j(k))(a + y_j(k)) \sum_l \delta_l(k) w_{lj}(k)\end{aligned} \tag{3.36}$$

when the neuron j is a hidden node.

Using Eqs. 3.31 and 3.32 for the logistic function and Eqs. 3.35 and 3.36 for the hyperbolic tangent function, we can compute the local gradient $\delta_j(k)$, and then get the correction $\Delta w_{ji}(k)$ applied to $w_{ji}(k)$.

3.2.2 The Strategy for Improving the Algorithm

In the backpropagation algorithm, the smaller we set the learning rate parameter, the smaller the changes to the synaptic weights in the network will be from one iteration to the next, and the smoother will be the trajectory in the weight space. However, this result is attained at the cost of a slower rate of learning. On the other hand, if we set the learning rate parameter too large in order to speed up the rate of learning, the corresponding large changes in the synaptic weights may assume such a form that the network becomes unstable.

For the purpose of increasing the rate of network learning while avoiding the appearance of instability, a momentum term may be added to the delta rule (Eq. 3.17). This results in a generalized delta rule formulated as

$$\Delta w_{ji}(k) = \beta \Delta w_{ji}(k - 1) + \alpha \delta_j(k) y_i(k) \tag{3.37}$$

where the parameter β is usually positive, called the momentum constant [Haykin, 2009]. This represents a minor modification to the weight update in the back-propagation algorithm.

Next, in order to observe the effect of the sequence of pattern presentations on the synaptic weights due to the momentum constant, we rewrite Eq. 3.37 as a time series with index t. Then, $\Delta w_{ji}(k)$ can be further denoted as a sum of $\delta_j(t) y_i(t)$ with t growing from 0 to k, that is,

$$\Delta w_{ji}(k) = \alpha \sum_{t=0}^{k} \beta^{k-t} \delta_j(t) y_i(t) \tag{3.38}$$

Obviously, the momentum constant must satisfy $0 \leq |\beta| < 1$ to ensure the time series to be convergent. The case $\beta = 0$ reveals that the backpropagation algorithm operates without momentum, namely, Eq. 3.37 becomes Eq. 3.17. Moreover, by making a

comparison between Eq. 3.16 and Eq. 3.17, we can obtain

$$\Delta w_{ji}(k) = -\alpha \sum_{t=0}^{k} \beta^{k-t} \frac{\partial E(t)}{\partial w_{ji}(t)} \tag{3.39}$$

which shows that the current adjustment $\Delta w_{ji}(k)$ represents the sum of an exponentially weighted time series.

The inclusion of a momentum term has a great impact on the backpropagation algorithm in terms of finding a proper equilibrium between the learning speed and stability during the training process of network. When the partial derivative $\partial E(t)/\partial w_{ji}(t)$ has the same algebraic sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(k)$ grows in magnitude, and therefore the weight $w_{ji}(k)$ is adjusted by a large amount. In this case, the inclusion of momentum in backpropagation algorithm tends to accelerate adjustment in steady directions. Conversely, when the partial derivative $\partial E(t)/\partial w_{ji}(t)$ has opposite sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(k)$ shrinks in magnitude, and therefore the weight $w_{ji}(k)$ is adjusted by a small amount. Here, the inclusion of momentum in backpropagation algorithm has a stabilizing effect.

3.2.3 The Design Procedure of the Algorithm

For implementing the backpropagation algorithm online, the sequential updating approach of network weights is performed here [Haykin, 1999]. The design procedure of backpropagation algorithm via the training sample $\{x(k), d(k)\}_{k=1}^{K}$ is described as follows. Notice $x(k)$ is the input vector applied to the input layer and $d(k)$ is the desired response vector presented to the output layer.

1. *Initialization.* Start with a reasonable network configuration. Set the synaptic weights and threshold levels of the network to small random numbers that are uniformly distributed.
2. *Presentation of training samples.* Present the network with an epoch of training examples. For each example in the sample, perform the forward and backward computations, as described in steps 3 and 4.
3. *Forward computation.* For a training example denoted by $(x(k), d(k))$, compute the induced local fields and function signals of the network by proceeding forward through the network, layer-by-layer. The induced local field $v_j^{(h)}$ for neuron j in layer h is

$$v_j^{(h)}(k) = \sum_{i=0}^{n} w_{ji}^{(h)}(k) y_i^{(h-1)}(k) \tag{3.40}$$

where $y_i^{(h-1)}(k)$ is the output signal of neuron i at iteration k, and $w_{ji}^{(h)}(k)$ is the synaptic weight of neuron j in layer h that is fed from neuron i in layer $h-1$.

For $i = 0$, we have $y_0^{(h-1)}(k) = +1$ and $w_{j0}^{(h)}(k) = b_j^{(h)}(k)$, where $b_j^{(h)}(k)$ is the bias applied to neuron j in layer h. Then, the output signal of neuron j in layer h is

$$y_j^{(h)}(k) = \phi_j(v_j^{(h)}(k)) \tag{3.41}$$

If neuron j is in the first hidden layer (i.e., $h = 1$), set

$$y_j^{(0)}(k) = x_j(k) \tag{3.42}$$

If neuron j is in the output layer (i.e., $h = H$, where H is referred to as the depth of the network), set

$$y_j^{(H)}(k) = o_j(k) \tag{3.43}$$

Then, the error signal can be obtained by

$$e_j^{(H)}(k) = d_j(k) - o_j(k) \tag{3.44}$$

4. *Backward computation.* Compute the local gradients of the network according to

$$\delta_j^{(h)}(k) = \begin{cases} e_j^{(H)}(k)\phi_j'(v_j^{(H)}(k)), & \text{if neuron j is located in output layer H} \\ \phi_j'(v_j^{(h)}(k))\sum_l \delta_l^{(h+1)}(k)w_{lj}^{(h+1)}(k), & \text{if neuron j is located in hidden layer h} \end{cases} \tag{3.45}$$

Here, $\phi_j'(v_j^{(h)}(k))$ denotes the differentiation with respect to the argument. Update the synaptic weights of the network in layer h in accordance with the generalized delta rule:

$$w_{ji}^{(h)}(k+1) = w_{ji}^{(h)}(k) + \beta\left(w_{ji}^{(h)}(k) - w_{ji}^{(h)}(k-1)\right) + \alpha\delta_j^{(h)}(k)y_i^{(h-1)}(k) \tag{3.45}$$

where α is the learning rate parameter and β is the momentum constant.

5. *Iteration.* Let $k = k + 1$. Iterate the forward and backward computations in steps 3 and 4 by presenting new epochs of training examples to the network until $\overline{E}(k)$ satisfies the prespecified requirement. The order of presentation of training examples should be randomized from epoch-to-epoch.

3.3 BATCH LEARNING AND ONLINE LEARNING

Now, we present two different learning methods, batch learning and online learning, on the basis of how the supervised learning of the multilayer perceptron is actually performed.

3.3.1 Batch Learning

In batch learning, adjustments to the synaptic weights of the multilayer perceptron are performed after presenting all the K examples in the training sample $\mathcal{E}$ that constitute one epoch of training. That is to say, the cost function for batch learning is defined by the average error energy $\overline{E}$. Adjustments to the synaptic weights of the multilayer perceptron are made on an epoch-by-epoch basis. Then, one realization of the learning curve is obtained by plotting $\overline{E}$ versus the number of epochs. Note that for each epoch of training, the examples in the training sample $\mathcal{E}$ are randomly shuffled. Therefore, the learning curve is computed by ensemble averaging a large enough number of such realizations, where each realization is performed for a different set of initial conditions chosen at random.

The advantages of batch learning are as follows when the gradient descent method is used to implement the training process:

- It can give an accurate estimation of the gradient vector, that is, the derivative of the cost function $\overline{E}$ with respect to the weight vector w, thereby guaranteeing, under simple conditions, convergence of the steepest descent method to a local minimum.
- It ensures the parallelization of the learning process.

Nevertheless, from a practical perspective, batch learning is rather demanding in terms of storage requirements.

Besides, in a statistical context, batch learning may be viewed as a form of statistical inference. Therefore, it is well suited for solving nonlinear regression problems.

3.3.2 Online Learning

In online learning, adjustments to the synaptic weights of the multilayer perceptron are performed on the example-by-example basis. Thus, the cost function to be minimized is the total instantaneous error energy E(k).

Consider an epoch of K training examples arranged in the order $\{x(1), d(1)\}$, $\{x(2), d(2)\}$, ..., $\{x(K), d(K)\}$. The first example pair $\{x(1), d(1)\}$ in the epoch is presented to the network, and the weight adjustments are performed using the gradient descent method. Then, the second example $\{x(2), d(2)\}$ in the epoch is presented to the network, which leads to further adjustments to weights in the network. This procedure is continued until the last example $\{x(K), d(K)\}$ is considered. Unfortunately, such a procedure works against the parallelization of online learning.

For a given set of initial conditions, a single realization of the learning curve is obtained by plotting the final value E(k) versus the number of epochs used in the training session. The training examples are randomly shuffled after each epoch. As with batch learning, the learning curve for online learning is computed by ensemble averaging such realizations over a large enough number of initial conditions chosen at

random. Naturally, for a given network structure, the learning curve obtained under online learning will be quite different from that under batch learning.

Given that the training examples are presented to the network in a random manner, the use of online learning makes the search in the multidimensional weight space stochastic in nature. Note that it is for this reason that the method of online learning is sometimes referred to as a stochastic method. This stochasticity has the desirable effect of making it less likely for the learning process to be trapped in a local minimum, which is a definite advantage of online learning over batch learning. Another advantage of online learning is the fact that it requires much less storage than batch learning.

Moreover, when the training data are redundant (i.e., the training sample $\mathscr{E}$ contains several copies of the same example), we find that, unlike batch learning, online learning is able to take advantage of this redundancy because the examples are presented one at a time.

Another useful property of online learning is its ability to track small changes in the training data, particularly when the environment responsible for generating the data is nonstationary.

To summarize, despite the disadvantage of online learning, it is highly popular for solving pattern classification problems due to the following two important practical reasons [Haykin, 2009]:

- Online learning is simple to implement.
- It provides effective solutions to large-scale and difficult pattern classification problems.

Accordingly, we can find that much of the material presented in the chapter is devoted to online learning.

3.4 CROSS-VALIDATION AND GENERALIZATION

3.4.1 Cross-Validation

From the above sections, we know that the essence of backpropagation learning is to encode an input–output mapping (represented by a set of labeled examples) into the synaptic weights and thresholds of a multilayer perceptron. It is hoped that the network becomes well trained so that it learns enough about the past to generalize to the future. From such a perspective, the learning process amounts to a choice of network parameterization for a given set of data. In other words, we may view the network selection problem as choosing, within a set of candidate model structures (parameterizations), the "best" one according to a certain criterion. Here, cross-validation, which is a standard tool in statistics, provides an appealing guiding principle.

First, the available data set is randomly partitioned into a training sample and a test set. The training sample is further partitioned into two disjoint subsets:

- an estimation subset, used to select the model;
- a validation subset, used to test or validate the model.

The motivation is to validate the model on a data set different from the one used for parameter estimation. In this way, we may use the training sample to assess the performance of various candidate models and then choose the "best" one. However, there is a distinct possibility that the model with the best-performing parameter values may end up overfitting the validation subset. For the purpose of guarding against this possibility, the generalization performance of the selected model is measured on the test set, which is different from the validation subset [Haykin, 2009].

Cross-validation is appealing particularly when we have to design a large neural network with good generalization as the goal in different ways.

1. *Network complexity.* The problem of choosing network complexity measured in terms of the number of hidden neurons used in a multilayer perceptron can be interpreted as that of choosing the size of the parameter set used to model the data set. Measured in terms of the ability of the network to generalize, there is obviously a limit on the size of the network. This follows from the basic observation that it may not be an optimal strategy to train the network to perfection on a given data set, because of the ill-posedness of any finite set of data representing a target function, a condition that is true for both "noisy" and "clean" data. Rather, it would be better to train the network in order to produce the "best" generalization. To do so, we may use cross-validation. Specifically, the training data set is partitioned into training and test subsets, in which case "overtraining" will show up as poorer performance on the cross-validation set.
2. *Size of training set.* Another direction in which cross-validation can be used is to decide when the training of a network on the training set should be actually stopped. In this case, the error performance of the network on generalization is exploited to determine the size of the data set used in training. The idea of cross-validation used here is illustrated in Figure 3.4, where two curves are shown for the mean-squared error in generalization, plotted versus the number of epochs used in training. In Figure 3.4, one curve relates to the use of few adjustable parameters (i.e., underfitting), while the other relates to the use of many parameters (i.e., overfitting). In addition, we can find that the error performance on generalization exhibits a minimum point, and the minimum mean-squared error for overfitting is smaller and better defined than that for underfitting. Therefore, we may obtain good generalization even if the neural network designed has too many parameters, provided that training of the network on the training set is stopped at a number of epochs corresponding to the minimum point of the error performance curve on cross-validation.
3. *Size of learning rate parameter.* Cross-validation may also be employed to adjust the size of the learning rate parameter of a multilayer perceptron, with backpropagation learning used as a pattern classifier. In particular, the network is first trained on the subtraining set, and then the cross-validation set is used to

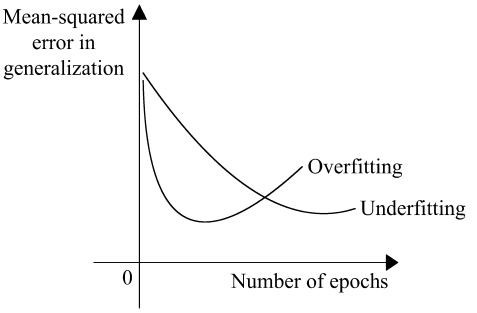

FIGURE 3.4 Illustrating the idea of cross-validation.

validate the training after each epoch. When the classification performance of the network on the cross-validation set fails to improve by a certain amount, the size of the learning rate parameter is reduced. After each succeeding epoch, the learning rate parameter is further reduced, until once again there is no further improvement in classification performance on the cross-validation set. The training of the network is halted as soon as that point is reached.

3.4.2 Generalization

During the process of backpropagation learning, we typically start with a training sample and use the backpropagation algorithm to compute the synaptic weights of a multilayer perceptron by loading (encoding) as many of the training examples as possible into the network. It is hoped that the designed neural network will generalize well. A network is said to generalize well when the input–output mapping computed by the network is correct (or nearly so) for test data never used in creating or training the network. Note that the term "generalization" is borrowed from psychology. Here, it is assumed that the test data are drawn from the same population used to generate the training data [Haykin, 2009].

Training a neural network may be considered as a "curve fitting" problem. Besides, the network itself may be viewed simply as a nonlinear input–output mapping. In this sense, we can regard generalization as the effect of a good nonlinear interpolation of the input data. The network performs useful interpolation primarily because multilayer perceptrons with continuous activation functions lead to output functions that are also continuous.

Figure 3.5 describes how generalization may occur in a hypothetical network. The nonlinear input–output mapping represented by the curve depicted in the figure is computed by the network as a result of learning the points labeled as "training data." The generalization point is seen as the result of interpolation performed by the network.

A neural network that is designed to generalize well will produce a correct input–output mapping even when the input is slightly different from the examples used to

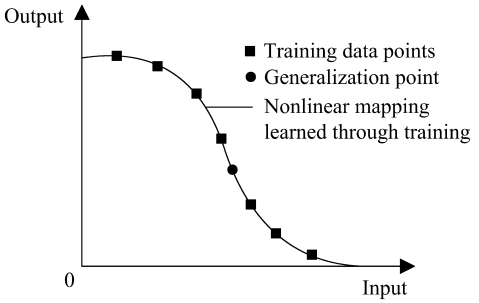

FIGURE 3.5 Properly fitted nonlinear mapping with good generalization.

train the network. However, if a neural network learns too many input–output examples, the network may end up memorizing the training data. It may do so by finding a feature that is present in the training data, but not true of the underlying function that is to be modeled. Such a phenomenon is referred to as overfitting or overtraining. When the network is overtrained, it loses the ability to generalize between similar input–output patterns.

Ordinarily, loading data into a multilayer perceptron in this way requires the use of more hidden neurons than are actually necessary, with the result that undesired contributions to the input space due to noise are stored in synaptic weights of the network. For the same data as depicted in Figure 3.5, an example of how poor generalization due to memorization in a neural network may occur is illustrated in Figure 3.6. "Memorization" is essentially a "lookup table," which implies that input–output mapping computed by the neural network is not smooth. In input–output mapping, on the contrary, it is important to select the "simplest" function in the absence of any prior knowledge [Poggio and Girosi, 1990a, 1990b]. The simplest function signifies the smoothest function that approximates the mapping for a given error criterion, because such a choice generally demands the fewest computational resources. In addition, smoothness is also natural in many applications, depending on

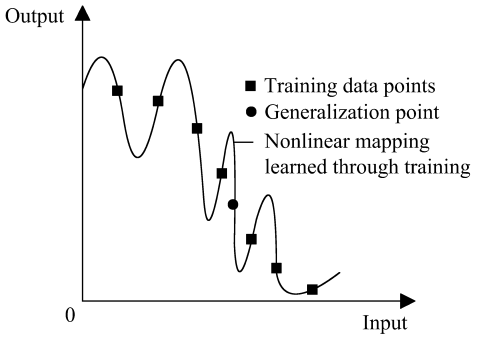

FIGURE 3.6 Overfitted nonlinear mapping with poor generalization.

the scale of the phenomenon being studied. Therefore, it is meaningful to seek a smooth nonlinear mapping for ill-posed input–output relationships, so that the network is able to classify novel patterns correctly with respect to the training patterns.

Overall, generalization is influenced by the following three factors:

- the size of the training sample and how representative the training sample is of the environment of interest;
- the architecture of the neural network; and
- the physical complexity of the problem at hand.

Clearly, we have no control over the third factor. Thus, we can view the issue of generalization from the following two perspectives [Haykin, 2009]:

- When the architecture of the network is fixed, the issue to be considered is that of determining the size of the training sample needed for a good generalization to occur.
- When the size of the training sample is fixed, the issue of interest is that of determining the best architecture of network for achieving good generalization.

In practice, it seems that all we really need for a good generalization is to have the size of the training sample K satisfy the condition

$$K = O\left(\frac{W}{\varepsilon}\right) \tag{3.47}$$

where W is the total number of free parameters in the network, including synaptic weights and biases, ε denotes the fraction of classification errors permitted on test data, and $O(\cdot)$ denotes the order of quantity enclosed within.

3.4.3 Convolutional Neural Networks

In this chapter, we know that the basic idea of backpropagation is that gradients can be computed efficiently by propagation from the output to the input. Needless to say, backpropagation is by far the most widely used neural network learning algorithm, and probably the most widely used learning algorithm of any form. In this part, we provide an extended introduction of the convolutional neural networks [LeCun *et al.*, 1998].

The ability of multilayer networks trained with gradient descent to learn complex, high-dimensional, nonlinear mappings from large collections of examples makes them obvious candidates for image recognition tasks. In the traditional model of pattern recognition, a hand-designed feature extractor gathers relevant information from the input and eliminates irrelevant variabilities. A trainable classifier then categorizes the resulting feature vectors into classes. In this scheme, standard, fully connected multilayer networks can be used as classifiers. A potentially more

interesting scheme is to rely as much as possible on learning in the feature extractor itself. In the case of character recognition, a network could be fed with almost raw inputs (e.g., size-normalized images). While this can be done with an ordinary fully connected feedforward network with some success for tasks such as character recognition, there are problems shown as follows.

For one thing, typical images are large, often with several hundred variables (pixels). A fully connected first layer with, for example, 100 hidden units in the first layer would already contain several tens of thousands of weights. Such a large number of parameters increase the capacity of the system and therefore it requires a larger training set. In addition, the memory requirement to store so many weights may rule out certain hardware implementations. But the main deficiency of unstructured nets for image or speech applications is that they have no built-in invariance with respect to translations or local distortions of the inputs. Before being sent to the fixed-size input layer of a neural network, character images, or other two- or one-dimensional signals, must be approximately size normalized and centered in the input field. Unfortunately, no such preprocessing can be perfect: Handwriting is often normalized at the word level, which can cause size, slant, and position variations for individual characters. This, combined with variability in writing style, will cause variations in the position of distinctive features in input objects. In principle, a fully connected network of sufficient size could learn to produce outputs that are invariant with respect to such variations. However, learning such a task would probably result in multiple units with similar weight patterns positioned at various locations in the input so as to detect distinctive features wherever they appear on the input. Learning these weight configurations requires a very large number of training instances to cover the space of possible variations. In convolutional neural networks, as described below, shift invariance is automatically obtained by forcing the replication of weight configurations across space.

For another thing, a deficiency of fully connected architectures is that the topology of the input is entirely ignored. The input variables can be presented in any (fixed) order without affecting the outcome of the training. On the contrary, images (or time–frequency representations of speech) have a strong two-dimensional local structure: Variables (or pixels) that are spatially or temporally nearby are highly correlated. Local correlations are the reasons for the well-known advantages of extracting and combining local features before recognizing spatial or temporal objects, because configurations of neighboring variables can be classified into a small number of categories (e.g., edges, corners). Convolutional neural networks force the extraction of local features by restricting the receptive fields of hidden units to be local.

Convolutional neural networks provide an efficient method to constrain the complexity of feedforward neural networks by weight sharing and restriction to local connections. Convolutional networks combine three architectural ideas to ensure some degree of shift, scale, and distortion invariance: (i) local receptive fields; (ii) shared weights (or weight replication); and (iii) spatial or temporal subsampling. The kernel of the convolution is the set of connection weights used by the units in the feature map. An interesting property of convolutional layers is that if the input image is shifted, the feature map output will be shifted by the same

amount, but it will be left unchanged otherwise. This property is at the basis of the robustness of convolutional networks to shifts and distortions of the input. Besides, since all the weights are learned with backpropagation, convolutional networks can be seen as synthesizing their own feature extractor. The weight sharing technique has the interesting side effect of reducing the number of free parameters, thereby reducing the "capacity" of the machine and reducing the gap between the test error and training error.

The field of deep machine learning focuses on computational models for information representation that exhibit similar characteristics to that of the neocortex. Actually, the structure of convolutional neural networks is well established in the current deep learning field and shows great promise for future work. Convolutional neural networks are the first truly successful deep learning approach where many layers of a hierarchy are successfully trained in a robust manner. A convolutional neural network is a choice of topology or architecture that leverages spatial relationships to reduce the number of parameters that must be learned and thus improves upon general feedforward backpropagation training. Convolutional neural networks were proposed as a deep learning framework that was motivated by minimal data preprocessing requirements. In convolutional neural networks, small portions of the image (dubbed a local receptive field) are treated as inputs to the lowest layer of the hierarchical structure. Information generally propagates through the different layers of the network whereby at each layer digital filtering is applied in order to obtain salient features of the data observed. The method provides a level of invariance to shift, scale, and rotation as the local receptive field allows the neuron or processing unit access to elementary features such as oriented edges or corners. The advancements made with respect to developing deep machine learning systems will undoubtedly shape the future of machine learning and artificial intelligence systems in general.

3.5 COMPUTER EXPERIMENT USING BACKPROPAGATION

In this section, we reconsider the exclusive-OR (XOR) problem [Haykin, 2009]. From Chapter 2, we know that the Rosenblatt's single-layer perceptron has no hidden neuron. As a result, it cannot classify input patterns that are not linearly separable. A typical example is the XOR problem.

The XOR problem can be viewed as a special case of the problem of classifying points in the unit hypercube. Each point in the hypercube is in either class 0 or class 1. In a more special case of the XOR problem, we need to consider only the four corners of a unit square that correspond to the input patterns $(0, 0)$, $(0, 1)$, $(1, 0)$, and $(1, 1)$.

Denote $\oplus$ as the exclusive-OR Boolean function operator. Since

$$0 \oplus 0 = 0 \tag{3.48}$$

$$1 \oplus 1 = 0 \tag{3.49}$$

TABLE 3.1 Pattern Classification

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

and

$$0 \oplus 1 = 1 \tag{3.50}$$

$$1 \oplus 0 = 1 \tag{3.51}$$

the input patterns $(0, 0)$ and $(1, 1)$ are in class 0, while $(0, 1)$ and $(1, 0)$ are in class 1. See Table 3.1. Obviously, the input patterns $(0, 0)$ and $(1, 1)$ are at opposite corners of the unit square, but they produce the identical output. This is the same for the input patterns $(0, 1)$ and $(1, 0)$.

We know that the use of a single neuron with two inputs results in a straight line for a decision boundary in the input space. For all points on one side of this line, the neuron outputs 1, while for all points on the other side of the line, it outputs 0. The position and orientation of the line in the input space are determined by the synaptic weights of the neuron connected to the input nodes and the bias applied to the neuron. As the input patterns $(0, 0)$ and $(1, 1)$ are located on opposite corners of the unit square, and likewise for the other two input patterns $(0, 1)$ and $(1, 0)$, we cannot construct a straight line for a decision boundary so that $(0, 0)$ and $(1, 1)$ lie in one decision region and $(0, 1)$ and $(1, 0)$ lie in the other decision region. This implies that the single-layer perceptron cannot solve the XOR problem.

However, we can solve the XOR problem by using a network with a single hidden layer with two neurons, as plotted in Figure 3.7. The corresponding signal flow graph of the network is shown in Figure 3.8. Notice that the following two assumptions are required.

- Each neuron is represented by a McCulloch–Pitts model, which utilizes the threshold function as activation function.

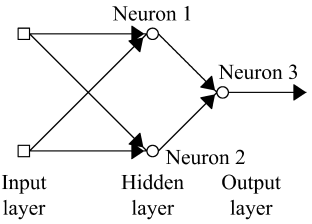

FIGURE 3.7 Architectural graph of network for solving the XOR problem.

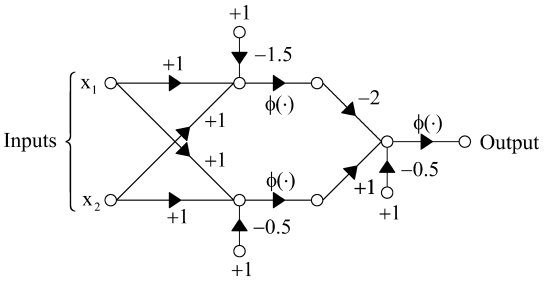

FIGURE 3.8 Signal flow graph of network for solving the XOR problem.

- Bits 0 and 1 are represented by the levels 0 and +1, respectively.

In Figure 3.7, the top neuron, labeled as "Neuron 1" in the hidden layer, is characterized as

$$w_{11} = w_{12} = +1$$
$$b_1 = -\frac{3}{2}$$

The slope of the decision boundary constructed by this hidden neuron is equal to -1 and positioned as in Figure 3.9. The bottom neuron, labeled as "Neuron 2" in the hidden layer, is characterized as

$$w_{21} = w_{22} = +1$$
$$b_2 = -\frac{1}{2}$$

The orientation and position of the decision boundary constructed by this hidden neuron are shown in Figure 3.10. The output neuron, labeled as "Neuron 3," is characterized as

$$w_{31} = -2$$
$$w_{32} = +1$$
$$b_3 = -\frac{1}{2}$$

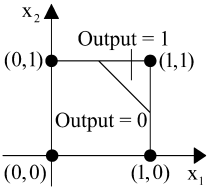

FIGURE 3.9 Decision boundary constructed by hidden neuron 1 of the network in Figure 3.7.

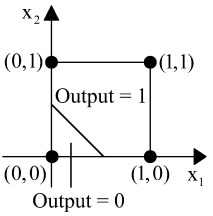

FIGURE 3.10 Decision boundary constructed by hidden neuron 2 of the network in Figure 3.7.

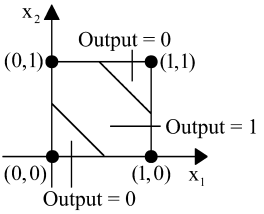

FIGURE 3.11 Decision boundaries constructed by the complete network in Figure 3.7.

The function of the output neuron is to construct a linear combination of the decision boundaries formed by the two hidden neurons. The computation result is shown in Figure 3.11. See Figure 3.8, we find that the bottom hidden neuron has a positive connection to the output neuron, whereas the top hidden neuron has a negative connection to the output neuron. When both hidden neurons are off, which occurs when the input pattern is $(0, 0)$, the output neuron remains off. When both hidden neurons are on, which occurs when the input pattern is $(1, 1)$, the output neuron is switched off again because the inhibitory effect of the larger negative weight connected to the top hidden neuron overpowers the excitatory effect of the positive weight connected to the bottom hidden neuron. When the top hidden neuron is off and the bottom neuron is on, which occurs when the input patterns is $(0, 1)$ or $(1, 0)$, the output neuron is switched on because of the excitatory effect of the positive weight connected to the bottom hidden neuron. Accordingly, the network described in Figure 3.7 does indeed solve the XOR problem.

EXERCISES

3.1. Consider the functions

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-(t^2/2)} dt$$

and

$$\phi(x) = \frac{2}{\pi}\tan^{-1}(x)$$

Explain why both the functions fit the requirement of a sigmoid function. What is the difference between the two functions?

3.2. Consider a two-layer network containing no hidden neurons. Assume that the network has q inputs and a single output neuron. Let x_i denote the ith input signal and define the corresponding output as

$$y = \phi\left(\sum_{i=0}^{q} w_i x_i\right)$$

where w_i is a threshold and

$$\phi(v) = \frac{1}{1 + \exp(-v)}$$

Show that this network implements a linear decision boundary that consists of a hyperplane in the input space $\mathfrak{R}^q$. Illustrate your conclusion when $q = 2$.

3.3. The network shown in Figure 3.12 has been trained to classify correctly a set of two-dimensional, two-class patterns. Identify the function performed by the classifier, assuming initially that the neurons have function

$$\phi(v) = \frac{1}{1 + \exp(-av)}$$

Draw the resulting separating lines between the two classes on the x_1, x_2 plane.

3.4. Figure 3.13 shows a neural network involving a single hidden neuron for solving the XOR problem. It can be viewed as an alternative to the network considered in Section 3.5. Show that the network of Figure 3.13 solves the XOR problem by constructing decision regions and a truth table for the network.

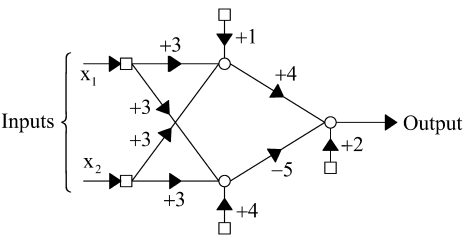

FIGURE 3.12 The network of Exercise 3.3.

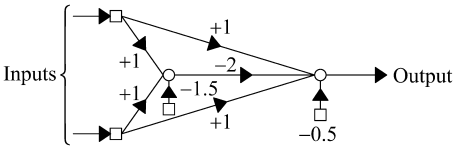

FIGURE 3.13 The network of Exercise 3.4.

3.5. Use the backpropagation algorithm for computing a set of synaptic weights and bias levels for a neural network structured as in Figure 3.7 to solve the XOR problem. Assume the use of a logistic function for the nonlinearity.

3.6. Consider the network shown in Figure 3.14, with the initial weight and basis are chosen as $w_1 = 1,\ b_1 = 1, w_2 = -2,\ b_2 = 1$.

The activation function $\phi(\cdot)$ is set the same as Exercise 3.2. Assume the input and the desired responses of the network are $x = 1$, $d = 1$, respectively.

1. Calculate the total instantaneous error energyE.
2. Compute $\partial E/\partial w_1$ based on the result of 1.
3. Recompute $\partial E/\partial w_1$ by using the backpropagation algorithm and compare the result with (2).

3.7. Linearly nonseparable patterns as shown in Figure 3.15 have to be classified into two categories by using a layered network. Construct the separating planes in the pattern space and draw patterns in the image space. Calculate all weights and

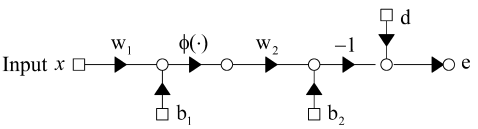

FIGURE 3.14 The network of Exercise 3.6.

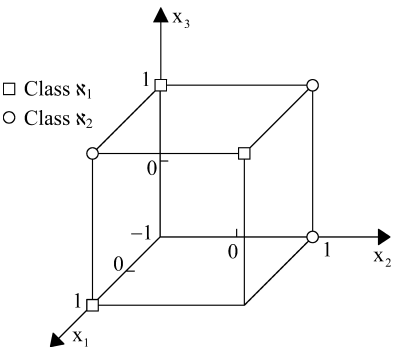

FIGURE 3.15 Patterns for layered network classification for Exercise 3.7.

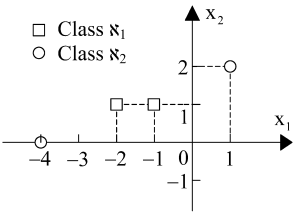

FIGURE 3.16 Pattern classification of Exercise 3.9.

threshold values of related units. Use the minimum number of threshold units to perform the classification.

3.8. Investigate the use of backpropagation learning algorithm employing a sigmoidal nonlinearity to achieve one-to-one mappings, as described below:

1. $f(x) = \frac{1}{x}, \quad 1 \leq x \leq 100$

2. $f(x) = \lg x, \quad 1 \leq x \leq 10$

3. $f(x) = \exp(-x), \quad 1 \leq x \leq 10,$

4. $f(x) = \sin x, \quad 0 \leq x \leq \frac{\pi}{2}$

For each mapping, do the following things.

1. Set up two sets of data, one for network training and the other for testing.

2. Use the training data set to compute the synaptic weights of the network, assuming it has a single hidden layer.

3. Evaluate the computation accuracy of the network by using the test data.

Use a single hidden layer, but with a variable number of hidden neurons. Investigate how the network performance is affected by varying the size of the hidden layer.

3.9. Classify the two classes of input patterns depicted in Figure 3.16 by using backpropagation training algorithm.