Multilayer Perceptrons

4.1 INTRODUCTION

In this chapter we study multilayer feedforward networks, an important class of neural networks. Typically, the network consists of a set of sensory units (source nodes) that constitute the *input layer*, one or more *hidden layers* of computation nodes, and an *output layer* of computation nodes. The input signal propagates through the network in a forward direction, on a layer-by-layer basis. These neural networks are commonly referred to as *multilayer perceptrons* (MLPs), which represent a generalization of the single-layer perceptron considered in Chapter 3.

Multilayer perceptrons have been applied successfully to solve some difficult and diverse problems by training them in a supervised manner with a highly popular algorithm known as the *error back-propagation algorithm*. This algorithm is based on the *error-correction learning rule*. As such, it may be viewed as a generalization of an equally popular adaptive filtering algorithm: the ubiquitous least-mean-square (LMS) algorithm described in Chapter 3 for the special case of a single linear neuron.

Basically, error back-propagation learning consists of two passes through the different layers of the network: a forward pass and a backward pass. In the *forward pass*, an activity pattern (input vector) is applied to the sensory nodes of the network, and its effect propagates through the network layer by layer. Finally, a set of outputs is produced as the actual response of the network. During the forward pass the synaptic weights of the networks are all *fixed*. During the *backward pass*, on the other hand, the synaptic weights are all *adjusted* in accordance with an error-correction rule. Specifically, the actual response of the network is subtracted from a desired (target) response to produce an *error signal*. This error signal is then propagated backward through the network, against the direction of synaptic connections—hence the name "error back-propagation." The synaptic weights are adjusted to make the actual response of the network move closer to the desired response in a statistical sense. The error back-propagation algorithm is also referred to in the literature as the *back-propagation algorithm*, or simply *back-prop*. Henceforth we will refer to it as the back-propagation algorithm. The learning process performed with the algorithm is called *back-propagation learning*.

A multilayer perceptron has three distinctive characteristics:

1. The model of each neuron in the network includes a *nonlinear activation function*. The important point to emphasize here is that the nonlinearity is *smooth* (i.e., differentiable everywhere), as opposed to the hard-limiting used in Rosenblatt's perceptron. A commonly used form of nonlinearity that satisfies this requirement is a *sigmoidal nonlinearity*[1] defined by the *logistic function:*

$$y_j = \frac{1}{1 + \exp(-v_j)}$$

   where v_j is the induced local field (i.e., the weighted sum of all synaptic inputs plus the bias) of neuron j, and y_j is the output of the neuron. The presence of nonlinearities is important because otherwise the input–output relation of the network could be reduced to that of a single-layer perceptron. Moreover, the use of the logistic function is biologically motivated, since it attempts to account for the refractory phase of real neurons.
2. The network contains one or more layers of *hidden neurons* that are not part of the input or output of the network. These hidden neurons enable the network to learn complex tasks by extracting progressively more meaningful features from the input patterns (vectors).
3. The network exhibits a high degrees of *connectivity*, determined by the synapses of the network. A change in the connectivity of the network requires a change in the population of synaptic connections or their weights.

It is through the combination of these characteristics together with the ability to learn from experience through training that the multilayer perceptron derives it computing power. These same characteristics, however, are also responsible for the deficiencies in our present state of knowledge on the behavior of the network. First, the presence of a distributed form of nonlinearity and the high connectivity of the network make the theoretical analysis of a multilayer perceptron difficult to undertake. Second, the use of hidden neurons makes the learning process harder to visualize. In an implicit sense, the learning process must decide which features of the input pattern should be represented by the hidden neurons. The learning process is therefore made more difficult because the search has to be conducted in a much larger space of possible functions, and a choice has to be made between alternative representations of the input pattern (Hinton, 1989).

The usage of the term "back-propagation" appears to have evolved after 1985, when its use was popularized through the publication of the seminal book entitled *Parallel Distributed Processing* (Rumelhart and McClelland, 1986). For historical notes on the back-propagation algorithm, see Section 1.9.

The development of the back-propagation algorithm represents a landmark in neural networks in that it provides a *computationally efficient* method for the training of multilayer perceptrons. Although we cannot claim that the back-propagation algorithm provides an optimal solution for all solvable problems, it has put to rest the pessimism about learning in multilayer machines that may have been inferred from the book by Minsky and Papert (1969).

Organization of the Chapter

In this chapter, we study basic aspects of the multilayer perceptron as well as back-propagation learning. The chapter is organized in seven parts. In the first part encompassing Sections 4.2 through 4.6, we discuss matters relating to back-propagation learning. We begin with some preliminaries in Section 4.2 to pave the way for the derivation of the back-propagation algorithm. In Section 4.3 we present a detailed derivation of the algorithm, using the chain rule of calculus; we take a traditional approach in the derivation presented here. A summary of the back-propagation algorithm is presented in Section 4.4. In Section 4.5 we illustrate the use of the back-propagation algorithm by solving the XOR problem, an interesting problem that cannot be solved by the single-layer perceptron. In Section 4.6 we present some heuristics or practical guidelines for making the back-propagation algorithm perform better.

The second part, encompassing Sections 4.7 through 4.9, explores the use of multilayer perceptrons for pattern recognition. In Section 4.7 we address the development of a rule for the use of a multilayer perceptron to solve the statistical pattern-recognition problem. In Section 4.8 we use a computer experiment to illustrate the application of back-propagation learning to distinguish between two classes of overlapping two-dimensional Gaussian distributions. The important role of hidden neurons as feature detectors is discussed in Section 4.9.

The third part of the chapter, encompassing Sections 4.10 through 4.12 deals with the error surface. In Section 4.10 we discuss the fundamental role of back-propagation learning in computing partial derivatives of an approximate function. We then discuss computational issues relating to the Hessian matrix of the error surface in Section 4.11.

The fourth part of the chapter deals with various matters relating to the performance of a multilayer perceptron trained with the back-propagation algorithm. In Section 4.12 we discuss the issue of generalization, the very essence of learning. Section 4.13 discusses the approximation of continuous functions by means of multilayer perceptrons. The use of cross-validation as a statistical design tool is discussed in Section 4.14. In Section 4.15 we describe procedures to orderly "prune" a multilayer perceptron while maintaining (and frequently improving) overall performance. Network pruning is desirable when computational complexity is of primary concern.

The fifth part of the chapter completes the study of back-propagation learning. In Section 4.16 we summarize the important advantages and limitations of back-propagation learning. In Section 4.17 we investigate heuristics that provide guidelines for how to accelerate the rate of convergence of back-propagation learning.

In the sixth part of the chapter we take a different viewpoint on learning. With improved learning as the objective, we discuss the issue of supervised learning as a problem in numerical optimization in Section 4.18. In particular, we describe the conjugate-gradient algorithm and quasi-Newton methods for supervised learning.

The last part of the chapter, Section 4.19, deals with the multilayer perceptron itself. There we describe an interesting neural network structure, the *convolutional multilayer perceptron*. This network has been successfully used in the solution of difficult pattern-recognition problems.

The chapter concludes with some general discussion in Section 4.20.

4.2 SOME PRELIMINARIES

Figure 4.1 shows the architectural graph of a multilayer perceptron with two hidden layers and an output layer. To set the stage for a description of the multilayer perceptron in its general form, the network shown here is *fully connected*. This means that a neuron in any layer of the network is connected to all the nodes/neurons in the previous layer. Signal flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.

Figure 4.2 depicts a portion of the multilayer perceptron. Two kinds of signals are identified in this network (Parker, 1987):

1. *Function Signals*. A function signal is an input signal (stimulus) that comes in at the input end of the network, propagates forward (neuron by neuron) through the network, and emerges at the output end of the network as an output signal. We refer to such a signal as a "function signal" for two reasons. First, it is presumed to perform a useful function at the output of the network. Second, at each neuron of the network through which a function signal passes, the signal is

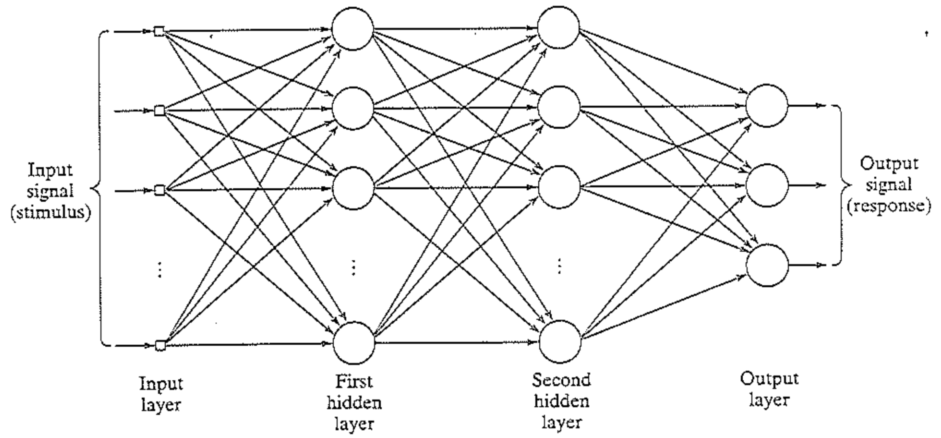


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

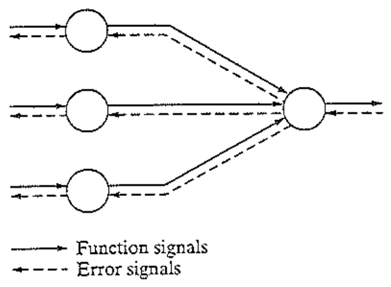


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

calculated as a function of the inputs and associated weights applied to that neuron. The function signal is also referred to as the input signal.

2. *Error Signals.* An error signal originates at an output neuron of the network, and propagates backward (layer by layer) through the network. We refer to it as an "error signal" because its computation by every neuron of the network involves an error-dependent function in one form or another.

The output neurons (computational nodes) constitute the output layers of the network. The remaining neurons (computational nodes) constitute hidden layers of the network. Thus the hidden units are not part of the output or input of the network—hence their designation as "hidden." The first hidden layer is fed from the input layer made up of sensory units (source nodes); the resulting outputs of the first hidden layer are in turn applied to the next hidden layer; and so on for the rest of the network.

Each hidden or output neuron of a multilayer perceptron is designed to perform two computations:

1. The computation of the function signal appearing at the output of a neuron, which is expressed as a continuous nonlinear function of the input signal and synaptic weights associated with that neuron.
2. The computation of an estimate of the gradient vector (i.e., the gradients of the error surface with respect to the weights connected to the inputs of a neuron), which is needed for the backward pass through the network.

The derivation of the back-propagation algorithm is rather involved. To ease the mathematical burden involved in this derivation, we first present a summary of the notations used in the derivation.

Notation

- The indices i, j, and k refer to different neurons in the network; with signals propagating through the network from left to right, neuron j lies in a layer to the right of neuron i, and neuron k lies in a layer to the right of neuron j when neuron j is a hidden unit.
- In iteration (time step) n, the nth training pattern (example) is presented to the network.
- The symbol $\mathscr{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration n. The average of $\mathscr{E}(n)$ over all values of n (i.e., the entire training set) yields the average error energy $\mathscr{E}_{\text{av}}$.
- The symbol $e_j(n)$ refers to the error signal at the output of neuron j for iteration n.
- The symbol $d_j(n)$ refers to the desired response for neuron j and is used to compute $e_j(n)$.
- The symbol $y_j(n)$ refers to the function signal appearing at the output of neuron j at iteration n.
- The symbol $w_{ji}(n)$ denotes the synaptic weight connecting the output of neuron i to the input of neuron j at iteration n. The correction applied to this weight at iteration n is denoted by $\Delta w_{ji}(n)$.

- The induced local field (i.e., weighted sum of all synaptic inputs plus bias) of neuron j at iteration n is denoted by $v_j(n)$; it constitutes the signal applied to the activation function associated with neuron j.
- The activation function describing the input–output functional relationship of the nonlinearity associated with neuron j is denoted by $\varphi_j(\cdot)$.
- The bias applied to neuron j is denoted by b_j; its effect is represented by a synapse of weight $w_{j0} = b_j$ connected to a fixed input equal to $+1$.
- The ith element of the input vector (pattern) is denoted by $x_i(n)$.
- The kth element of the overall output vector (pattern) is denoted by $o_k(n)$.
- The learning-rate parameter is denoted by η.
- The symbol m_l denotes the size (i.e., number of nodes) in layer l of the multilayer perceptron; $l = 0, 1, \ldots, L$, where L is the "depth" of the network. Thus m_0 denotes the size of the input layer, m_1 denotes the size of the first hidden layer, and m_L denotes the size of the output layer. The notation $m_L = M$ is also used.

4.3 BACK-PROPAGATION ALGORITHM

The error signal at the output of neuron j at iteration n (i.e., presentation of the nth training example) is defined by

$$e_j(n) = d_j(n) - y_j(n), \qquad \text{neuron } j \text{ is an output node} \tag{4.1}$$

We define the instantaneous value of the error energy for neuron j as $\frac{1}{2}e_j^2(n)$. Correspondingly, the instantaneous value $\mathscr{E}(n)$ of the total error energy is obtained by summing $\frac{1}{2}e_j^2(n)$ over *all neurons in the output layer;* these are the only "visible" neurons for which error signals can be calculated directly. We may thus write

$$\mathscr{E}(n) = \frac{1}{2}\sum_{j\in C} e_j^2(n) \tag{4.2}$$

where the set C includes all the neurons in the output layer of the network. Let N denote the total number of patterns (examples) contained in the training set. The *average squared error energy* is obtained by summing $\mathscr{E}(n)$ over all n and then normalizing with respect to the set size N, as shown by

$$\mathscr{E}_{\text{av}} = \frac{1}{N}\sum_{n=1}^{N} \mathscr{E}(n) \tag{4.3}$$

The instantaneous error energy $\mathscr{E}(n)$, and therefore the average error energy $\mathscr{E}_{\text{av}}$, is a function of all the free parameters (i.e., synaptic weights and bias levels) of the network. For a given training set, $\mathscr{E}_{\text{av}}$ represents the *cost function* as a measure of learning performance. The objective of the learning process is to adjust the free parameters of the network to minimize $\mathscr{E}_{\text{av}}$. To do this minimization, we use an approximation similar in rationale to that used for the derivation of the LMS algorithm in Chapter 3. Specifically, we consider a simple method of training in which the weights are updated on a *pattern-by-pattern* basis until one *epoch,* that is, one complete presentation of the entire training set has been dealt with. The adjustments to the weights are made in accordance with the respective errors computed for *each* pattern presented to the network.

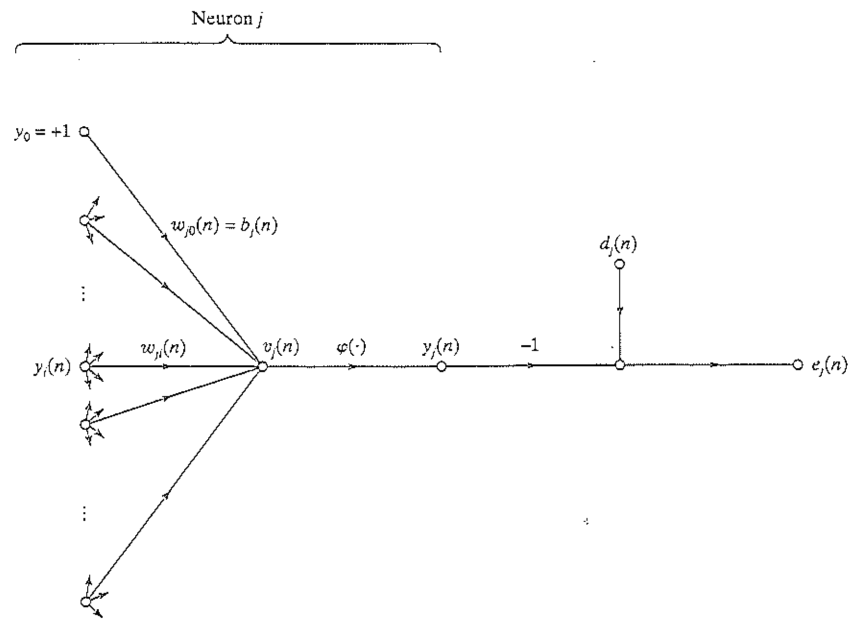


FIGURE 4.3 Signal-flow graph highlighting the details of output neuron j.

The arithmetic average of these individual weight changes over the training set is therefore an *estimate* of the true change that would result from modifying the weights based on minimizing the cost function $\mathscr{E}_{\text{av}}$ over the entire training set. We will address the quality of the estimate later in this section.

Consider then Fig. 4.3, which depicts neuron j being fed by a set of function signals produced by a layer of neurons to its left. The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)y_i(n) \tag{4.4}$$

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

$$y_j(n) = \varphi_j(v_j(n)) \tag{4.5}$$

In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\partial\mathscr{E}(n)/\partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as:

$$\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial\mathscr{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}\frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{4.6}$$

The partial derivative $\partial\mathscr{E}(n)/\partial w_{ji}(n)$ represents a *sensitivity factor*, determining the direction of search in weight space for the synaptic weight w_{ji}.

Differentiating both sides of Eq. (4.2) with respect to $e_j(n)$, we get

$$\frac{\partial\mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \tag{4.7}$$

Differentiating both sides of Eq. (4.1) with respect to $y_j(n)$, we get

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{4.8}$$

Next, differentiating Eq. (4.5) with respect to $v_j(n)$, we get

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{4.9}$$

where the use of prime (on the right-hand side) signifies differentiation with respect to the argument. Finally, differentiating Eq. (4.4) with respect to $w_{ji}(n)$ yields

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{4.10}$$

The use of Eqs. (4.7) to (4.10) in (4.6) yields

$$\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{4.11}$$

The correction $\Delta w_{ji}(n)$ applied to $w_{ji}(n)$ is defined by the *delta rule*:

$$\Delta w_{ji}(n) = -\eta\frac{\partial\mathscr{E}(n)}{\partial w_{ji}(n)} \tag{4.12}$$

where η is the *learning-rate parameter* of the back-propagation algorithm. The use of the minus sign in Eq. (4.12) accounts for *gradient descent* in weight space (i.e., seeking a direction for weight change that reduces the value of $\mathscr{E}(n)$). Accordingly, the use of Eq. (4.11) in (4.12) yields

$$\Delta w_{ji}(n) = \eta\delta_j(n)y_i(n) \tag{4.13}$$

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

$$\begin{aligned}\delta_j(n) &= -\frac{\partial\mathscr{E}(n)}{\partial v_j(n)}\\ &= -\frac{\partial\mathscr{E}(n)}{\partial e_j(n)}\frac{\partial e_j(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)}\\ &= e_j(n)\varphi_j'(v_j(n))\end{aligned} \tag{4.14}$$

The local gradient points to required changes in synaptic weights. According to Eq. (4.14), the local gradient $\delta_j(n)$ for output neuron j is equal to the product of the corresponding error signal $e_j(n)$ for that neuron and the derivative $\varphi_j'(v_j(n))$ of the associated activation function.

From Eqs. (4.13) and (4.14) we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ji}(n)$ is the error signal $e_j(n)$ at the output of neuron j. In this context we may identify two distinct cases, depending on where in the network neuron j is located. In case 1, neuron j is an output node. This case is simple to handle because each output node of the network is supplied with a desired response of its own, making it a straightforward matter to calculate the associated error signal. In case 2, neuron j is a hidden node. Even though hidden neurons are not directly accessible, they share responsibility for any error made at the output of the network. The question, however, is to know how to penalize or reward hidden neurons for their share of the responsibility. This problem is the *credit-assignment problem* considered in Section 2.7. It is solved in an elegant fashion by back-propagating the error signals through the network.

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 may use Eq. (4.1) to compute the error signal $e_j(n)$ associated with this neuron; see Fig. 4.3. Having determined $e_j(n)$, it is a straightforward matter to compute the local gradient $\delta_j(n)$ using Eq. (4.14).

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. Accordingly, the error signal for a hidden neuron would have to be determined recursively in terms of the error signals of all the neurons to which that hidden neuron is directly connected; this is where the development of the back-propagation algorithm gets complicated. Consider the situation depicted in Fig. 4.4, which depicts neuron j as a hidden node of the network. According to Eq. (4.14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron j as

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)}\varphi_j'(v_j(n)), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \tag{4.15}$$

where in the second line we have used Eq. (4.9). To calculate the partial derivative $\partial \mathscr{E}(n)/\partial y_j(n)$, we may proceed as follows. From Fig. 4.4 we see that

$$\mathscr{E}(n) = \frac{1}{2}\sum_{k\in C} e_k^2(n), \qquad \text{neuron } k \text{ is an output node} \tag{4.16}$$

which is Eq. (4.2) with index k used in place of index j. We have done so in order to avoid confusion with the use of index j that refers to a hidden neuron under case 2. Differentiating Eq. (4.16) with respect to the function signal $y_j(n)$, we get

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{4.17}$$

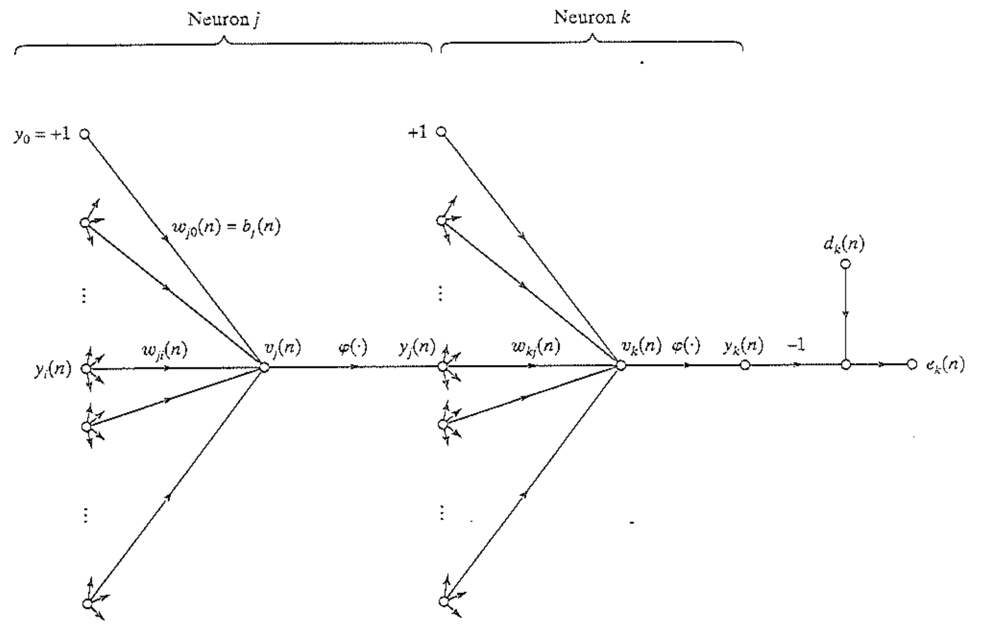


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j.

Next we use the chain rule for the partial derivative $\partial e_k(n)/\partial y_j(n)$, and rewrite Eq. (4.17) in the equivalent form

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k(n)\frac{\partial e_k(n)}{\partial v_k(n)}\frac{\partial v_k(n)}{\partial y_j(n)} \tag{4.18}$$

However, from Fig. 4.4, we note that

$$\begin{aligned}e_k(n) &= d_k(n) - y_k(n) \\ &= d_k(n) - \varphi_k(v_k(n)), \qquad \text{neuron } k \text{ is an output node}\end{aligned} \tag{4.19}$$

Hence

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n)) \tag{4.20}$$

We also note from Fig. 4.4 that for neuron k the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n)y_j(n) \tag{4.21}$$

where m is the total number of inputs (excluding the bias) applied to neuron k. Here again, the synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron k, and the corresponding input is fixed at the value +1. Differentiating Eq. (4.21) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{4.22}$$

By using Eqs. (4.20) and (4.22) in (4.18) we get the desired partial derivative:

$$\begin{aligned}\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n)\varphi_k'(v_k(n))\,w_{kj}(n) \\ &= -\sum_k \delta_k(n)\,w_{kj}(n)\end{aligned} \tag{4.23}$$

where in the second line we have used the definition of the local gradient $\delta_k(n)$ given in Eq. (4.14) with the index k substituted for j.

Finally, using Eq. (4.23) in (4.15), we get the *back-propagation formula* for the local gradient $\delta_j(n)$ as described:

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n) w_{kj}(n), \qquad \text{neuron } j \text{ is hidden} \tag{4.24}$$

Figure 4.5 shows the signal-flow graph representation of Eq. (4.24), assuming that the output layer consists of m_L neurons.

The factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ in Eq. (4.24) depends solely on the activation function associated with hidden neuron j. The remaining factor involved in this computation, namely the summation over k, depends on two sets of terms. The first set of terms, the $\delta_k(n)$, requires knowledge of the error signals $e_k(n)$, for all neurons that lie in the layer to the immediate right of hidden neuron j, and that are directly connected to neuron j: see Fig. 4.4. The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

We now summarize the relations that we have derived for the back-propagation algorithm. First, the correction $\Delta w_{ji}(n)$ applied to the synaptic weight connecting neuron i to neuron j is defined by the delta rule:

$$\begin{pmatrix}\textit{Weight}\\ \textit{correction}\\ \Delta w_{ji}(n)\end{pmatrix} = \begin{pmatrix}\textit{learning-}\\ \textit{rate parameter}\\ \eta\end{pmatrix}\cdot\begin{pmatrix}\textit{local}\\ \textit{gradient}\\ \delta_j(n)\end{pmatrix}\cdot\begin{pmatrix}\textit{input signal}\\ \textit{of neuron } j\\ y_i(n)\end{pmatrix} \tag{4.25}$$

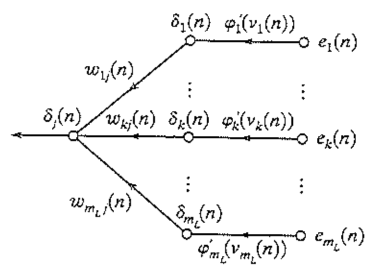


FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

Second, the local gradient $\delta_j(n)$ depends on whether neuron j is an output node or a hidden node:

1. If neuron j is an output node, $\delta_j(n)$ equals the product of the derivative $\varphi_j'(v_j(n))$ and the error signal $e_j(n)$, both of which are associated with neuron j; see Eq.(4.14).
2. If neuron j is a hidden node, $\delta_j(n)$ equals the product of the associated derivative $\varphi_j'(v_j(n))$ and the weighted sum of the δs computed for the neurons in the next hidden or output layer that are connected to neuron j; see Eq. (4.24).

The Two Passes of Computation

In the application of the back-propagation algorithm, two distinct passes of computation are distinguished. The first pass is referred to as the forward pass, and the second is referred to as the backward pass.

In the *forward pass* the synaptic weights remain unaltered throughout the network, and the function signals of the network are computed on a neuron-by-neuron basis. The function signal appearing at the output of neuron j is computed as

$$y_j(n) = \varphi(v_j(n)) \tag{4.26}$$

where $v_j(n)$ is the induced local field of neuron j, defined by

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n)\,y_i(n) \tag{4.27}$$

where m is the total number of inputs (excluding the bias) applied to neuron j, and $w_{ji}(n)$ is the synaptic weight connecting neuron i to neuron j, and $y_i(n)$ is the input signal of neuron j or equivalently, the function signal appearing at the output of neuron i. If neuron j is in the first hidden layer of the network, $m = m_0$ and the index i refers to the ith input terminal of the network, for which we write

$$y_i(n) = x_i(n) \tag{4.28}$$

where $x_i(n)$ is the ith element of the input vector (pattern). If, on the other hand, neuron j is in the output layer of the network, $m = m_L$ and the index j refers to the jth output terminal of the network, for which we write

$$y_j(n) = o_j(n) \tag{4.29}$$

where $o_j(n)$ is the jth element of the output vector (pattern). This output is compared with the desired response $d_j(n)$, obtaining the error signal $e_j(n)$ for the jth output neuron. Thus the forward phase of computation begins at the first hidden layer by presenting it with the input vector, and terminates at the output layer by computing the error signal for each neuron of this layer.

The backward pass, on the other hand, starts at the output layer by passing the error signals leftward through the network, layer by layer, and recursively computing the δ (i.e., the local gradient) for each neuron. This recursive process permits the synaptic weights of the network to undergo changes in accordance with the delta rule of Eq.(4.25). For a neuron located in the output layer, the δ is simply equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity. Hence we use

Eq. (4.25) to compute the changes to the weights of all the connections feeding into the output layer. Given the δs for the neurons of the output layer, we next use Eq. (4.24) to compute the δs for all the neurons in the penultimate layer and therefore the changes to the weights of all connections feeding into it. The recursive computation is continued, layer by layer, by propagating the changes to all synaptic weights in the network.

Note that for the presentation of each training example, the input pattern is fixed ("clamped") throughout the round-trip process, encompassing the forward pass followed by the backward pass.

Activation Function

The computation of the δ for each neuron of the multilayer perceptron requires knowledge of the derivative of the activation function $\varphi(\cdot)$ associated with that neuron. For this derivative to exist, we require the function $\varphi(\cdot)$ to be continuous. In basic terms, *differentiability* is the only requirement that an activation function has to satisfy. An example of a continuously differentiable nonlinear activation function commonly used in multilayer perceptrons is *sigmoidal nonlinearity;* two forms are described:

1. *Logistic Function.* This form of sigmoidal nonlinearity in its general form is defined by

$$\varphi_j(v_j(n)) = \frac{1}{1 + \exp(-av_j(n))} \qquad a > 0 \text{ and } -\infty < v_j(n) < \infty \qquad (4.30)$$

where $v_j(n)$ is the induced local field of neuron j. According to this nonlinearity, the amplitude of the output lies inside the range $0 \le y_j \le 1$. Differentiating Eq. (4.30) with respect to $v_j(n)$, we get

$$\varphi_j'(v_j(n)) = \frac{a\exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2} \qquad (4.31)$$

With $y_j(n) = \varphi_j(v_j(n))$, we may eliminate the exponential term $\exp(-av_j(n))$ from Eq. (4.31), and so express the derivative $\varphi_j'(v_j(n))$ as

$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)] \qquad (4.32)$$

For a neuron j located in the output layer, $y_j(n) = o_j(n)$. Hence, we may express the local gradient for neuron j as

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= a[d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \qquad \text{neuron } j \text{ is an output node}\end{aligned} \qquad (4.33)$$

where $o_j(n)$ is the function signal at the output of neuron j, and $d_j(n)$ is the desired response for it. On the other hand, for an arbitrary hidden neuron j, we may express the local gradient as

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) \\ &= ay_j(n)[1 - y_j(n)]\sum_k \delta_k(n)w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \qquad (4.34)$$

Note from Eq. (4.32) that the derivative $\varphi_j'(v_j(n))$ attains its maximum value at $y_j(n) = 0.5$, and its minimum value (zero) at $y_j(n) = 0$, or $y_j(n) = 1.0$. Since the amount of change in a synaptic weight of the network is proportional to the derivative $\varphi_j'(v_j(n))$, it follows that for a sigmoid activation function the synaptic weights are changed the most for those neurons in the network where the function signals are in their midrange. According to Rumelhart et al. (1986a), it is this feature of back-propagation learning that contributes to its stability as a learning algorithm.

2. *Hyperbolic tangent function.* Another commonly used form of sigmoidal nonlinearity is the hyperbolic tangent function, which in its most general form is defined by

$$\varphi_j(v_j(n)) = a\tanh(bv_j(n)), \qquad (a,b) > 0 \qquad (4.35)$$

where a and b are constants. In reality, the hyperbolic tangent function is just the logistic function rescaled and biased. Its derivative with respect to $v_j(n)$ is given by

$$\begin{aligned}\varphi_j'(v_j(n)) &= ab\,\mathrm{sech}^2(bv_j(n)) \\ &= ab\left(1 - \tanh^2(bv_j(n))\right) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)]\end{aligned} \qquad (4.36)$$

For a neuron j located in the output layer, the local gradient is

$$\begin{aligned}\delta_j(n) &= e_j(n)\varphi_j'(v_j(n)) \\ &= \frac{b}{a}[d_j(n) - o_j(n)][a - o_j(n)]\,[a + o_j(n)]\end{aligned} \qquad (4.37)$$

For a neuron j in a hidden layer, we have

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)]\sum_k \delta_k(n)w_{kj}(n), \qquad \text{neuron } j \text{ is hidden}\end{aligned} \qquad (4.38)$$

By using Eqs. (4.33) and (4.34) for the logistic function and Eqs. (4.37) and (4.38) for the hyperbolic tangent function, we may calculate the local gradient δ_j without requiring explicit knowledge of the activation function.

Rate of Learning

The back-propagation algorithm provides an "approximation" to the trajectory in weight space computed by the method of steepest descent. The smaller we make the learning-rate parameter η, 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 weight space. This improvement, however, is attained at the cost of a slower rate of learning. If, on the other hand, we make the learning-rate parameter η too large in order to speed up the rate of learning, the resulting large changes in the synaptic weights assume such a form that the network may become unstable (i.e., oscillatory). A simple method of increasing the rate of learning yet avoiding the danger of instability

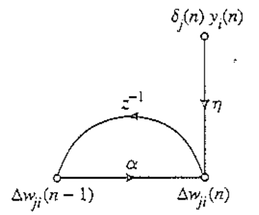


FIGURE 4.6 Signal-flow graph illustrating the effect of momentum constant α.

is to modify the delta rule of Eq.(4.13) by including a momentum term,[2] as shown by (Rumelhart et al. 1986a)

$$\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n) \tag{4.39}$$

where α is usually a positive number called the *momentum constant*. It controls the feedback loop acting around $\Delta w_{ji}(n)$, as illustrated in Fig. 4.6 where z^{-1} is the unit-delay operator. Equation (4.39) is called the *generalized delta rule*[3]; it includes the delta rule of Eq. (4.13) as a special case (i.e., $\alpha = 0$).

In order to see the effect of the sequence of pattern presentations on the synaptic weights due to the momentum constant α, we rewrite Eq. (4.39) as a time series with index t. The index t goes from the initial time 0 to the current time n. Equation (4.39) may be viewed as a first-order difference equation in the weight correction $\Delta w_{ji}(n)$. Solving this equation for $\Delta w_{ji}(n)$ we have

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t) \tag{4.40}$$

which represents a time series of length $n + 1$. From Eqs. (4.11) and (4.14) we note the product $\delta_j(n) y_i(n)$ is equal to $-\partial \mathscr{E}(n)/\partial w_{ji}(n)$. Accordingly, we may rewrite Eq. (4.40) in the equivalent form

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \mathscr{E}(t)}{\partial w_{ji}(t)} \tag{4.41}$$

Based on this relation, we may make the following insightful observations (Watrous, 1987; Jacobs, 1988):

1. The current adjustment $\Delta w_{ji}(n)$ represents the sum of an exponentially weighted time series. For the time series to be *convergent*, the momentum constant must be restricted to the range $0 \leq |\alpha| < 1$. When α is zero, the back-propagation algorithm operates without momentum. Also the momentum constant α can be positive or negative, although it is unlikely that a negative α would be used in practice.

2. When the partial derivative $\partial \mathscr{E}(t)/\partial w_{ji}(t)$ has the same algebraic sign on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ grows in magnitude, and so the weight $w_{ji}(n)$ is adjusted by a large amount. The inclusion of momentum in the back-propagation algorithm tends to *accelerate descent* in steady downhill directions.

3. When the partial derivative $\partial \mathscr{E}(t)/\partial w_{ji}(t)$ has opposite signs on consecutive iterations, the exponentially weighted sum $\Delta w_{ji}(n)$ shrinks in magnitude, so the weight $w_{ji}(n)$ is adjusted by a small amount. The inclusion of momentum in the back-propagation algorithm has a *stabilizing effect* in directions that oscillate in sign.

The incorporation of momentum in the back-propagation algorithm represents a minor modification to the weight update, yet it may have some beneficial effects on the learning behavior of the algorithm. The momentum term may also have the benefit of preventing the learning process from terminating in a shallow local minimum on the error surface.

In deriving the back-propagation algorithm, it was assumed that the learning-rate parameter is a constant denoted by η. In reality, however, it should be defined as η_{ji}; that is, the learning-rate parameter should be *connection-dependent*. Indeed, many interesting things can be done by making the learning-rate parameter different for different parts of the network. We provide more detail on this issue in subsequent sections.

It is also noteworthy that in the application of the back-propagation algorithm we may choose all the synaptic weights in the network to be adjustable, or we may constrain any number of weights in the network to remain fixed during the adaptation process. In the latter case, the error signals are back-propagated through the network in the usual manner; however, the fixed synaptic weights are left unaltered. This can be done simply by making the learning-rate parameter η_{ji} for synaptic weight w_{ji} equal to zero.

Sequential and Batch Modes of Training

In a practical application of the back-propagation algorithm, learning results from the many presentations of a prescribed set of training examples to the multilayer perceptron. As mentioned earlier, one complete presentation of the entire training set during the learning process is called an *epoch*. The learning process is maintained on an epoch-by-epoch basis until the synaptic weights and bias levels of the network stabilize and the average squared error over the entire training set converges to some minimum value. It is good practice to *randomize the order of presentation of training examples* from one epoch to the next. This randomization tends to make the search in weight space stochastic over the learning cycles, thus avoiding the possibility of limit cycles in the evolution of the synaptic weight vectors; limit cycles are discussed in Chapter 14.

For a given training set, back-propagation learning may thus proceed in one of two basic ways:

1. *Sequential Mode.* The *sequential mode* of back-propagation learning is also referred to as *on-line, pattern, or stochastic mode*. In this mode of operation weight updating is performed after the presentation of each training example; this is the very mode of operation for which the derivation of the back-propagation algorithm presented applies. To be specific, consider an epoch consisting of N training examples (patterns) arranged in the order $(\mathbf{x}(1), \mathbf{d}(1)), \ldots, (\mathbf{x}(N), \mathbf{d}(N))$. The first example pair $(\mathbf{x}(1), \mathbf{d}(1))$ in the epoch is presented to the network, and the sequence of forward and backward computations described previously is performed, resulting in certain adjustments to the synaptic weights and bias levels of the network. Then the second example pair $(\mathbf{x}(2), \mathbf{d}(2))$ in the epoch is presented, and the sequence of forward and backward computations is repeated, resulting in further adjustments to the synaptic weights and

bias levels. This process is continued until the last example pair $(\mathbf{x}(N), \mathbf{d}(N))$ in the epoch is accounted for.

2. *Batch Mode.* In the *batch mode* of back-propagation learning, weight updating is performed *after* the presentation of *all* the training examples that constitute an epoch. For a particular epoch, we define the cost function as the average squared error of Eqs. (4.2) and (4.3), reproduced here in the composite form:

$$\mathscr{E}_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in C} e_j^2(n) \tag{4.42}$$

where the error signal $e_j(n)$ pertains to output neuron j for training example n and which is defined by Eq. (4.1). The error $e_j(n)$ equals the difference between $d_j(n)$ and $y_j(n)$, which represents the jth element of the desired response vector $\mathbf{d}(n)$ and the corresponding value of the network output, respectively. In Eq. (4.42) the inner summation with respect to j is performed over all the neurons in the output layer of the network, whereas the outer summation with respect to n is performed over the entire training set in the epoch at hand. For a learning-rate parameter η, the adjustment applied to synaptic weight w_{ji}, connecting neuron i to neuron j, is defined by the delta rule

$$\begin{aligned} \Delta w_{ji} &= -\eta \frac{\partial \mathscr{E}_{av}}{\partial w_{ji}} \\ &= -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}} \end{aligned} \tag{4.43}$$

To calculate the partial derivative $\partial e_j(n)/\partial w_{ji}$ we proceed in the same way as before. According to Eq. (4.43), in the batch mode the weight adjustment Δw_{ji} is made only after the entire training set has been presented to the network.

From an "on-line" operational point of view, the sequential mode of training is preferred over the batch mode because it requires *less* local storage for each synaptic connection. Moreover, given that the patterns are presented to the network in a random manner, the use of pattern-by-pattern updating of weights makes the search in weight space *stochastic* in nature. This in turn makes it less likely for the back-propagation algorithm to be trapped in a local minimum.

In the same way, the stochastic nature of the sequential mode makes it difficult to establish theoretical conditions for convergence of the algorithm. In contrast, the use of batch mode of training provides an accurate estimate of the gradient vector; convergence to a local minimum is thereby guaranteed under simple conditions. Also, the composition of the batch mode makes it easier to parallelize than the sequential mode.

When the training data are *redundant* (i.e., the data set contains several copies of exactly the same pattern), we find that unlike the batch mode, the sequential mode is able to take advantage of this redundancy because the examples are presented one at a time. This is particularly so when the data set is large and highly redundant.

In summary, despite the fact that the sequential mode of back-propagation learning has several disadvantages, it is highly popular (particularly for solving pattern-classification problems) for two important practical reasons:

- The algorithm is simple to implement.
- It provides effective solutions to large and difficult problems.

Stopping Criteria

In general, the back-propagation algorithm cannot be shown to converge, and there are no well-defined criteria for stopping its operation. Rather, there are some reasonable criteria, each with its own practical merit, which may be used to terminate the weight adjustments. To formulate such a criterion, it is logical to think in terms of the unique properties of a *local* or *global minimum* of the error surface[4]. Let the weight vector $\mathbf{w}^*$ denote a minimum, be it local or global. A necessary condition for $\mathbf{w}^*$ to be a minimum is that the gradient vector $\mathbf{g}(\mathbf{w})$ (i.e., first-order partial derivative) of the error surface with respect to the weight vector $\mathbf{w}$ be zero at $\mathbf{w} = \mathbf{w}^*$. Accordingly, we may formulate a sensible convergence criterion for back-propagation learning as follows (Kramer and Sangiovanni-Vincentelli, 1989):

The back-propagation algorithm is considered to have converged when the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.

The drawback of this convergence criterion is that, for successful trials, learning times may be long. Also, it requires the computation of the gradient vector $\mathbf{g}(\mathbf{w})$.

Another unique property of a minimum that we can use is the fact that the cost function or error measure $\mathscr{E}_{av}(\mathbf{w})$ is stationary at the point $\mathbf{w} = \mathbf{w}^*$. We may therefore suggest a different criterion of convergence:

The back-propagation algorithm is considered to have converged when the absolute rate of change in the average squared error per epoch is sufficiently small.

The rate of change in the average squared error is typically considered to be small enough if it lies in the range of 0.1 to 1 percent per epoch. Sometimes a value as small as 0.01 percent per epoch is used. Unfortunately, this criterion may result in a premature termination of the learning process.

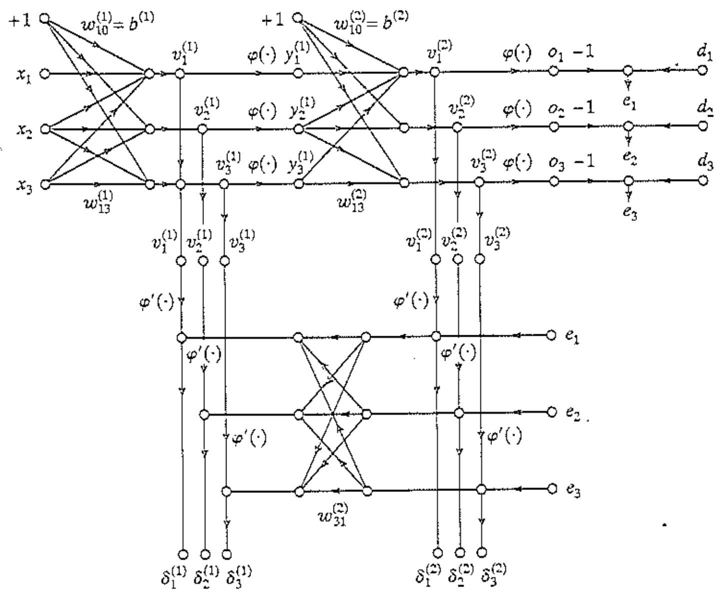
There is another useful and theoretically supported criterion for convergence. After each learning iteration, the network is tested for its generalization performance. The learning process is stopped when the generalization performance is adequate, or when it is apparent that the generalization performance has peaked; see Section 4.14 for more details.

4.4 SUMMARY OF THE BACK-PROPAGATION ALGORITHM

Figure 4.1 presents the architectural layout of a multilayer perceptron. The corresponding signal-flow graph for back-propagation learning, incorporating both the forward and backward phases of the computations involved in the learning process, is presented in Fig. 4.7 for the case of $L = 2$ and $m_0 = m_1 = m_2 = 3$. The top part of the signal-flow graph accounts for the forward pass. The lower part of the signal-flow graph accounts for the backward pass, which is referred to as a *sensitivity graph* for computing the local gradients in the back-propagation algorithm (Narendra and Parthasarathy, 1990).

Earlier we mentioned that the sequential updating of weights is the preferred method for on-line implementation of the back-propagation algorithm. For this mode of operation, the algorithm cycles through the training sample $\{(\mathbf{x}(n), \mathbf{d}(n))\}_{n=1}^{N}$ as follows:

1. *Initialization.* Assuming that no prior information is available, pick the synaptic weights and thresholds from a uniform distribution whose mean is zero and whose



+1
$w_{10}^{(1)} = b^{(1)}$
x_1
x_2
x_3
$w_{13}^{(1)}$
$v_1^{(1)}$
$v_2^{(1)}$
$v_3^{(1)}$
$\varphi(\cdot)$ $y_1^{(1)}$
$\varphi(\cdot)$ $y_2^{(1)}$
$\varphi(\cdot)$ $y_3^{(1)}$
+1
$w_{10}^{(2)} = b^{(2)}$
$w_{13}^{(2)}$
$v_1^{(2)}$
$v_2^{(2)}$
$v_3^{(2)}$
$\varphi(\cdot)$ o_1 -1
$\varphi(\cdot)$ o_2 -1
$\varphi(\cdot)$ o_3 -1
e_1
e_2
e_3
d_1
d_2
d_3
$\varphi'(\cdot)$
$w_{31}^{(2)}$
$\delta_1^{(1)}$ $\delta_2^{(1)}$ $\delta_3^{(1)}$
$\delta_1^{(2)}$ $\delta_2^{(2)}$ $\delta_3^{(2)}$

FIGURE 4.7 Signal-flow graphical summary of back-propagation learning. Top part of the graph: forward pass. Bottom part of the graph: backward pass.

variance is chosen to make the standard deviation of the induced local fields of the neurons lie at the transition between the linear and saturated parts of the sigmoid activation function.

2. *Presentations of Training Examples.* Present the network with an epoch of training examples. For each example in the set, ordered in some fashion, perform the sequence of forward and backward computations described under points 3 and 4, respectively.

3. *Forward Computation.* Let a training example in the epoch be denoted by $(\mathbf{x}(n), \mathbf{d}(n))$, with the input vector $\mathbf{x}(n)$ applied to the input layer of sensory nodes and the desired response vector $\mathbf{d}(n)$ presented to the output layer of computation nodes. Compute the induced local fields and function signals of the network by proceeding forward through the network, layer by layer. The induced local field $v_j^{(l)}(n)$ for neuron j in layer l is

$$v_j^{(l)}(n) = \sum_{i=0}^{m_0} w_{ji}^{(l)}(n) y_i^{(l-1)}(n) \qquad (4.44)$$

where $y_i^{(l-1)}(n)$ is the output (function) signal of neuron i in the previous layer $l - 1$ at iteration n and $w_{ji}^{(l)}(n)$ is the synaptic weight of neuron j in layer l that is fed from neuron i in layer $l - 1$. For $i = 0$, we have $y_0^{(l-1)}(n) = +1$ and $w_{j0}^{(l)}(n) = b_j^{(l)}(n)$ is the bias

applied to neuron j in layer l. Assuming the use of a sigmoid function, the output signal of neuron j in layer l is

$$y_j^{(l)} = \varphi_j(v_j(n))$$

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

$$y_j^{(0)}(n) = x_j(n)$$

where $x_j(n)$ is the jth element of the input vector $\mathbf{x}(n)$. If neuron j is in the output layer (i.e., $l = L$, where L is referred to as the *depth* of the network), set

$$y_j^{(L)} = o_j(n)$$

Compute the error signal

$$e_j(n) = d_j(n) - o_j(n) \qquad (4.45)$$

where $d_j(n)$ is the jth element of the desired response vector $\mathbf{d}(n)$.

4. *Backward Computation.* Compute the δs (i.e., local gradients) of the network, defined by

$$\delta_j^{(l)}(n) = \begin{cases} e_j^{(L)}(n)\, \varphi_j'(v_j^{(L)}(n)) & \text{for neuron } j \text{ in output layer } L \\ \varphi_j'(v_j^{(l)}(n)) \sum_k \delta_k^{(l+1)}(n) w_{kj}^{(l+1)}(n) & \text{for neuron } j \text{ in hidden layer } l \end{cases} \qquad (4.46)$$

where the prime in $\varphi_j'(\cdot)$ denotes differentiation with respect to the argument. Adjust the synaptic weights of the network in layer l according to the generalized delta rule:

$$w_{ji}^{(l)}(n+1) = w_{ji}^{(l)}(n) + \alpha[w_{ji}^{(l)}(n-1)] + \eta \delta_j^{(l)}(n) y_i^{(l-1)}(n) \qquad (4.47)$$

where η is the learning-rate parameter and α is the momentum constant.

5. *Iteration.* Iterate the forward and backward computations under points 3 and 4 by presenting new epochs of training examples to the network until the stopping criterion is met.

Notes: The order of presentation of training examples should be randomized from epoch to epoch. The momentum and learning-rate parameter are typically adjusted (and usually decreased) as the number of training iterations increases. Justification for these points will be presented later.

4.5 XOR PROBLEM

In the elementary (single-layer) perceptron there are no hidden neurons. Consequently, it cannot classify input patterns that are not linearly separable. However, nonlinearly separable patterns are of common occurrence. For example, this situation arises in the *Exclusive OR (XOR) problem*, which may be viewed as a special case of a more general problem, namely that of classifying points in the *unit hypercube*. Each point in the hypercube is either in class 0 or class 1. However, in the special case of the XOR problem, we need consider only the four corners of the *unit square* that correspond to the input patterns (0,0), (0,1), (1,1), and (1,0). The first and third input patterns are in class 0, as shown by

$$0 \oplus 0 = 0$$

and

$$1 \oplus 1 = 0$$

where $\oplus$ denotes the Exclusive OR Boolean function operator. The input patterns (0,0) and (1,1) are at opposite corners of the unit square, yet they produce the identical output 0. On the other hand, the input patterns (0,1) and (1,0) are also at opposite corners of the square, but they are in class 1, as shown by

$$0 \oplus 1 = 1$$

and

$$1 \oplus 0 = 1$$

We first recognize 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; 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. With the input patterns (0,0) and (1,1) located on opposite corners of the unit square, and likewise for the other two input patterns (0,1) and (1,0), it is clear that we cannot construct a straight line for a decision boundary so that (0,0) and (0,1) lie in one decision region, and (0,1) and (1,0) lie in the other decision region. In other words, an elementary perceptron cannot solve the XOR problem.

We may solve the XOR problem by using a single hidden layer with two neurons, as in Fig. 4.8a. (Touretzky and Pomerleau, 1989). The signal-flow graph of the network is shown in Fig. 4.8b. The following assumptions are made here:

- Each neuron is represented by a McCulloch–Pitts model, which uses a threshold function for its activation function.
- Bits 0 and 1 are represented by the levels 0 and +1, respectively.

The top neuron, labeled 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 Fig. 4.9a. The bottom neuron, labeled 2 in the hidden layer, is characterized as:

$$w_{21} = w_{22} = +1$$

$$b_2 = -\frac{1}{2}$$

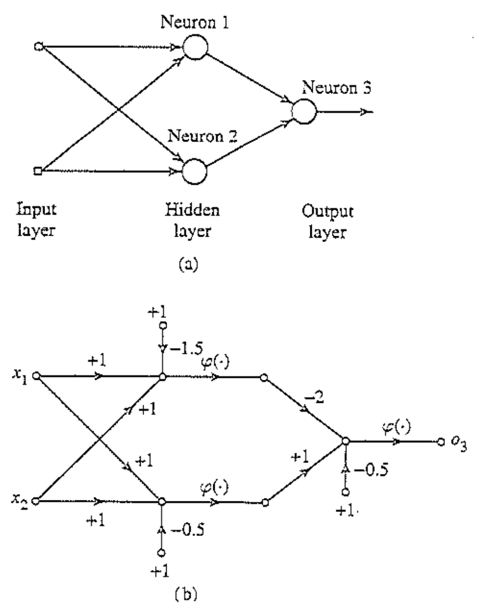


FIGURE 4.8 (a) Architectural graph of network for solving the XOR problem. (b) Signal-flow graph of the network.

The orientation and position of the decision boundary constructed by this second hidden neuron are as shown in Fig. 4.9b.

The output neuron, labeled 3 in Fig. 4.8a, is characterized as:

$$w_{31} = -2$$

$$w_{32} = +1$$

$$b_3 = -\frac{1}{2}$$

The function of the output neuron is to construct a linear combination of the decision boundaries formed by the two hidden neurons. The result of this computation is shown in Fig. 4.9c. The bottom hidden neuron has an excitatory (positive) connection to the output neuron, whereas the top hidden neuron has a stronger inhibitory (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 hidden neuron is on, which occurs when the input pattern is (0,1) or (1,0), the output neuron is switched on due to the excitatory effect of the positive weight connected to the bottom hidden neuron. Thus the network of Fig. 4.8a does indeed solve the XOR problem.

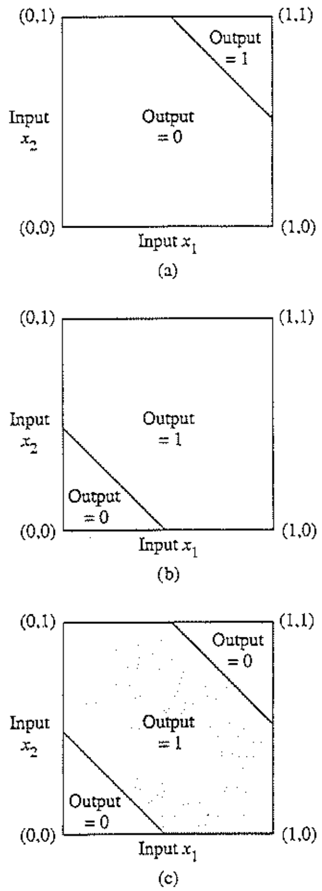


FIGURE 4.9 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8. (b) Decision boundary constructed by hidden neuron 2 of the network. (c) Decision boundaries constructed by the complete network.

4.6 HEURISTICS FOR MAKING THE BACK-PROPAGATION ALGORITHM PERFORM BETTER

It is often said that the design of a neural network using the back-propagation algorithm is more of an art than a science in the sense that many of the numerous factors involved in the design are the results of one's own personal experience. There is some truth in this statement. Nevertheless, there are methods that will significantly improve the back-propagation algorithm's performance, as described here.

1. *Sequential versus batch update.* As mentioned previously, the sequential mode of back-propagation learning (involving pattern-by-pattern updating) is computationally faster than the batch mode. This is especially true when the training data set is

large and highly redundant. (Highly redundant data pose computational problems for the estimation of the Jacobian required for the batch update.)

2. *Maximizing information content.* As a general rule, every training example presented to the back-propagation algorithm should be chosen on the basis that its information content is the largest possible for the task at hand (LeCun, 1993). Two ways of achieving this aim are:

- The use of an example that results in the largest training error.
- The use of an example that is radically different from all those previously used.

These two heuristics are motivated by a desire to search more of the weight space.

In pattern-classification tasks using sequential back-propagation learning, a simple technique that is commonly used is to randomize (i.e., shuffle) the order in which the examples are presented to the multilayer perceptron from one epoch to the next. Ideally, the randomization ensures that the successive examples in an epoch presented to the network rarely belong to the same class.

For a more refined technique, we may use an *emphasizing scheme*, which involves more difficult patterns than easy ones being presented to the network (LeCun, 1993). Whether a particular pattern is easy or difficult can be identified by examining the error it produces, compared to previous iterations of the algorithm. However, there are two problems with the use of an emphasizing scheme that should be carefully examined:

- The distribution of examples within an epoch presented to the network is distorted.
- The presence of an outlier or a mislabeled example can have a catastrophic consequence on the performance of the algorithm; learning such outliers compromises the generalization ability of the network over more probable regions of the input space.

3. *Activation function.* A multilayer perceptron trained with the back-propagation algorithm may, in general, learn faster (in terms of the number of training iterations required) when the sigmoid activation function built into the neuron model of the network is antisymmetric than when it is nonsymmetric; see Section 4.11 for details. We say that an activation function $\varphi(v)$ is *antisymmetric* (i.e., odd function of its argument) if

$$\varphi(-v) = -\varphi(v)$$

as depicted in Fig. 4.10a. This condition is not satisfied by the standard logistic function depicted in Fig. 4.10b.

A popular example of an antisymmetric activation function is a sigmoidal nonlinearity in the form of a *hyperbolic tangent*, defined by

$$\varphi(v) = a \tanh(bv)$$

where a and b are constants. Suitable values for the constants a and b are (LeCun, 1989, 1993)

$$a = 1.7159$$

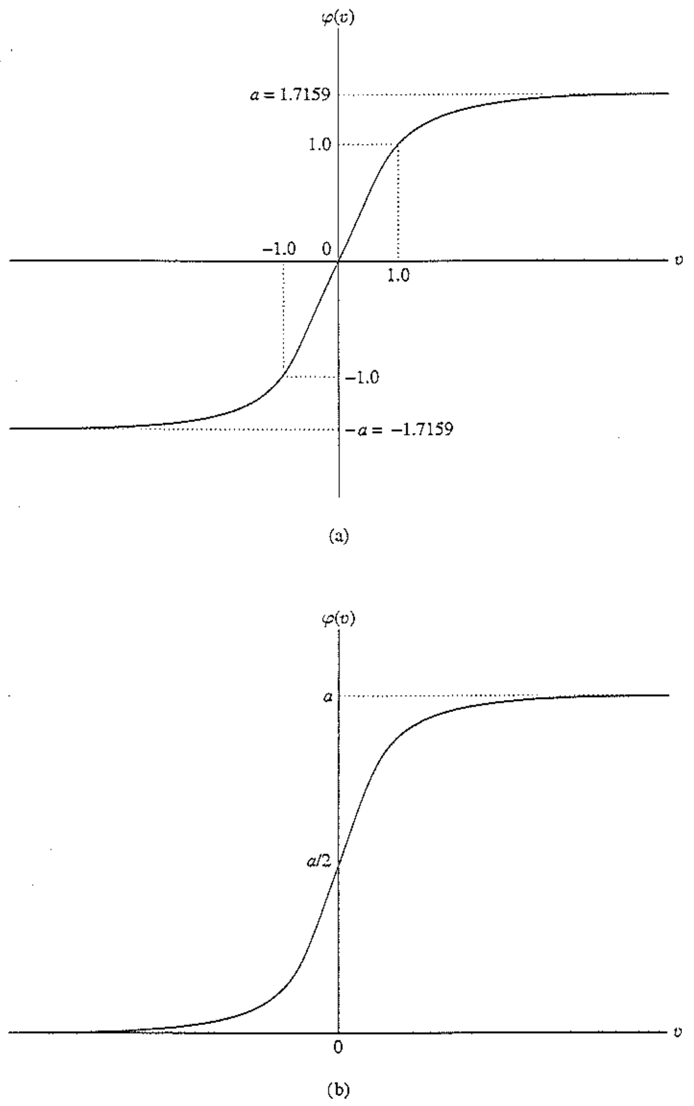


FIGURE 4.10 Antisymmetric activation function. (b) Nonsymmetric activation function.

and

$$b = \frac{2}{3}$$

The hyperbolic tangent function so defined has the following useful properties:

- $\varphi(1) = 1$ and $\varphi(-1) = -1$
- At the origin the slope (i.e., effective gain) of the activation function is close to unity, as shown by

$$\begin{aligned}\varphi(0) &= ab \\ &= 1.7159 \times 2/3 \\ &= 1.1424\end{aligned}$$

- The second derivative of $\varphi(v)$ attains its maximum value at $v = 1$.

4. *Target values*. It is important that the target values (desired response) be chosen within the range of the sigmoid activation function. More specifically, the desired response d_j for neuron j in the output layer of the multilayer perceptron should be *offset* by some amount ϵ away from the limiting value of the sigmoid activation function, depending on whether the limiting value is positive or negative. Otherwise the back-propagation algorithm tends to drive the free parameters of the network to infinity, and thereby slow down the learning process by driving the hidden neurons into saturation. To be specific, consider the antisymmetric activation function of Fig. 4.10a. For the limiting value $+a$, we set

$$d_j = a - \epsilon$$

and for the limiting value of $-a$, we set

$$d_j = -a + \epsilon$$

where ϵ is an appropriate positive constant. For the choice of $a = 1.7159$ referred to earlier, we may set $\epsilon = 0.7159$, in which case the target value (desired response) d_j can be conveniently chosen as ± 1, as indicated in Fig. 4.10a.

5. *Normalizing the inputs*. Each input variable should be *preprocessed* so that its mean value, averaged over the entire training set, is close to zero, or else it is small compared to its standard deviation (LeCun, 1993). To appreciate the practical significance of this rule, consider the extreme case where the input variables are consistently positive. In this situation, the synaptic weights of a neuron in the first hidden layer can only increase together or decrease together. Accordingly, if the weight vector of that neuron is to change direction, it can only do so by zigzagging its way through the error surface, which is typically slow and should therefore be avoided.

In order to accelerate the back-propagation learning process, the normalization of the inputs should also include two other measures (LeCun, 1993):

- The input variables contained in the training set should be *uncorrelated*; this can be done by using principal components analysis, as detailed in Chapter 8.
- The decorrelated input variables should be scaled so that their *covariances are approximately equal*, thereby ensuring that the different synaptic weights in the network learn at approximately the same speed.

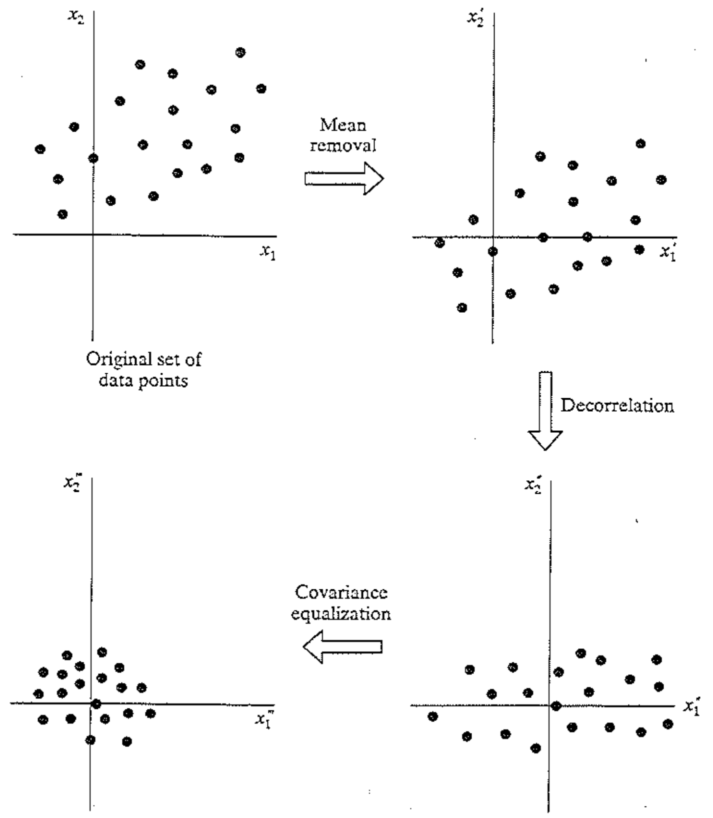


FIGURE 4.11 Illustrating the operation of mean removal, decorrelation, and covariance equalization for a two-dimensional input space.

Figure 4.11 illustrates the results of three normalization steps: mean removal, decorrelation, and covariance equalization, applied in that order.

6. *Initialization.* A good choice for the initial values of the synaptic weights and thresholds of the network can be of tremendous help in a successful network design. The key question is: What is a good choice?

When the synaptic weights are assigned large initial values, it is highly likely that the neurons in the network will be driven into saturation. If this happens, the local gradients in the back-propagation algorithm assume small values, which in turn will cause the learning process to slow down. However, if the synaptic weights are assigned small initial values, the back-propagation algorithm may operate on a very flat area around the origin of the error surface; this is particularly true in the case of antisymmetric activation functions such as the hyperbolic tangent function. Unfortunately, the origin is a *saddle point*, which refers to a stationary point where the curvature of the error surface

across the saddle is negative and the curvature along the saddle is positive. For these reasons the use of both large and small values for initializing the synaptic weights should be avoided. The proper choice of initialization lies somewhere between these two extreme cases.

To be specific, consider a multilayer perceptron using the hyperbolic tangent function for its activation functions. Let the bias applied to each neuron in the network be set to zero. We may then express the induced local field of neuron j as

$$v_j = \sum_{i=1}^{m} w_{ji} y_i$$

Let it be assumed that the inputs applied to each neuron in the network have zero mean and unit variance, as shown by

$$\mu_y = E[y_i] = 0 \qquad \text{for all } i$$

and

$$\sigma_y^2 = E[(y_i - \mu_i)^2] = E[y_i^2] = 1 \qquad \text{for all } i$$

Let it be further assumed that the inputs are uncorrelated, as shown by

$$E[y_i y_k] = \begin{cases} 1 & \text{for } k = i \\ 0 & \text{for } k \neq i \end{cases}$$

and that the synaptic weights are drawn from a uniformly distributed set of numbers with zero mean

$$\mu_w = E[w_{ji}] = 0 \qquad \text{for all } (j, i) \text{ pairs}$$

and variance

$$\sigma_w^2 = E[(w_{ji} - \mu_w)^2] = E[w_{ji}^2] \qquad \text{for all } (j, i) \text{ pairs}$$

Accordingly, we may express the mean and variance of the induced local field v_j as

$$\mu_v = E[v_j] = E\left[\sum_{i=1}^{m} w_{ji} y_i\right] = \sum_{i=1}^{m} E[w_{ji}]E[y_i] = 0$$

and

$$\begin{aligned} \sigma_v^2 &= E[(v_j - \mu_v)^2] = E[v_j^2] \\ &= E\left[\sum_{i=1}^{m}\sum_{k=1}^{m} w_{ji} w_{jk} y_i y_k\right] \\ &= \sum_{i=1}^{m}\sum_{k=1}^{m} E[w_{ji} w_{jk}] E[y_i y_k] \\ &= \sum_{i=1}^{m} E[w_{ji}^2] \\ &= m\sigma_w^2 \end{aligned} \tag{4.48}$$

where m is the number of synaptic connections of a neuron.

In light of this result, we may now describe a good strategy for initializing the synaptic weights so that the standard deviation of the induced local field of a neuron lies in the transition area between the linear and saturated parts of its sigmoid activation function. For example, for the case of a hyperbolic tangent function with its parameters a and b as specified previously, this objective is satisfied by setting $\sigma_v = 1$ in Eq. (4.48), in which case we obtain (LeCun, 1993)

$$\sigma_w = m^{-1/2} \tag{4.49}$$

Thus it is desirable for the uniform distribution, from which the synaptic weights are selected, to have a mean of zero and a variance equal to the reciprocal of the number of synaptic connections of a neuron.

7. *Learning from hints.* Learning from a set of training examples deals with an unknown input–output mapping function $f(\cdot)$. In effect, the learning process exploits the information contained in the examples about the function $f(\cdot)$ to *infer* an approximate implementation of it. The process of learning from examples may be generalized to *include learning from hints,* which is achieved by allowing prior information that we may have about the function $f(\cdot)$ to be included in the learning process (Abu-Mostafa, 1995). Such information may include invariance properties, symmetries, or any other knowledge about the function $f(\cdot)$ that may be used to accelerate the search for its approximate realization, and more importantly, to improve the quality of the final estimate. The use of Eq. (4.49) is an example of how this is achieved.

8. *Learning rates.* All neurons in the multilayer perceptron should ideally learn at the same rate. The last layers usually have larger local gradients than the layers at the front end of the network. Hence, the learning-rate parameter η should be assigned a smaller value in the last layers than in the front layers. Neurons with many inputs should have a smaller learning-rate parameter than neurons with few inputs so as to maintain a similar learning time for all neurons in the network. In LeCun (1993), it is suggested that for a given neuron, the learning rate should be inversely proportional to the square root of synaptic connections made to that neuron. We discuss learning rates more fully in Section 4.17.

4.7 OUTPUT REPRESENTATION AND DECISION RULE

In theory, for an M-class *classification problem* in which the union of the M distinct classes forms the entire input space, we need a total of M outputs to represent all possible classification decisions, as depicted in Fig. 4.12. In this figure the vector $\mathbf{x}_j$ denotes the jth *prototype* (i.e., unique sample) of an m-dimensional random vector $\mathbf{x}$ to be classified by a multilayer perceptron. The kth of M possible classes to which $\mathbf{x}$ can belong is denoted by $\mathscr{C}_k$. Let y_{kj} be the kth output of the network produced in response to the prototype $\mathbf{x}_j$, as shown by

$$y_{k,j} = F_k(\mathbf{x}_j), \qquad k = 1, 2, \dots, M \tag{4.50}$$

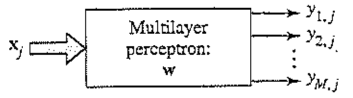


FIGURE 4.12 Block diagram of a pattern classifier.

where the function $F_k(\cdot)$ defines the mapping learned by the network from the input to the kth output. For convenience of presentation, let

$$\begin{aligned} \mathbf{y}_j &= [y_{1,j}, y_{2,j}, \dots, y_{M,j}]^T \\ &= [F_1(\mathbf{x}_j), F_2(\mathbf{x}_j), \dots, F_M(\mathbf{x}_j)]^T \\ &= \mathbf{F}(\mathbf{x}_j) \end{aligned} \tag{4.51}$$

where $\mathbf{F}(\cdot)$ is a vector-valued function. A basic question we wish to address in this section is:

After a multilayer perceptron is trained, what should the optimum decision rule be for classifying the M outputs of the network?

Clearly, any reasonable output decision rule should be based on knowledge of the vector-valued function:

$$\mathbf{F}: \mathbb{R}^m \ni \mathbf{x} \rightarrow \mathbf{y} \in \mathbb{R}^M \tag{4.52}$$

In general, all that is certain about the vector-valued function $\mathbf{F}(\cdot)$ is that it is a continuous function that minimizes the *empirical risk functional:*

$$R = \frac{1}{2N} \sum_{j=1}^{N} \|\mathbf{d}_j - \mathbf{F}(\mathbf{x}_j)\|^2 \tag{4.53}$$

where $\mathbf{d}_j$ is the desired (target) output pattern for the prototype $\mathbf{x}_j$, $\|\cdot\|$ is the Euclidean norm of the enclosed vector, and N is the total number of examples presented to the network in training. The essence of the criterion of Eq.(4.53) is the same as the cost function of Eq. (4.3). The vector-valued function $\mathbf{F}(\cdot)$ is strongly dependent on the choice of examples $(\mathbf{x}_j, \mathbf{d}_j)$ used to train the network, so that different values of $(\mathbf{x}_j, \mathbf{d}_j)$ will indeed lead to different vector-valued function $\mathbf{F}(\cdot)$. Note that the terminology $(\mathbf{x}_j, \mathbf{d}_j)$ used here is the same as that of $(\mathbf{x}(j), \mathbf{d}(j))$ used previously.

Suppose now that the network is trained with *binary* target values (that incidently correspond to the upper and lower bounds on the network outputs when using the logistic function), written as:

$$d_{kj} = \begin{cases} 1 & \text{when the prototype } \mathbf{x}_j \text{ belongs to class } \mathscr{C}_k \\ 0 & \text{when the prototype } \mathbf{x}_j \text{ does not belong to class } \mathscr{C}_k \end{cases} \tag{4.54}$$

Based on this notation, class $\mathscr{C}_k$ is represented by the M-dimensional target vector

$$\begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \begin{matrix} \\ \\ \leftarrow k\text{th element} \\ \\ \\ \end{matrix}$$

It is tempting to suppose that a multilayer perceptron classifier trained with the back-propagation algorithm on a finite set of independently and identically distributed (i.i.d.) examples may lead to an asymptotic approximation of the underlying *a posteriori* class probabilities. This property may be justified on the following grounds (White, 1989a; Richard and Lippmann, 1991):