

Multilayer Perceptrons

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 its 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

1. 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.

2. 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.

3. 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.

4. 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.

5. 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.

6. 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.

7. 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

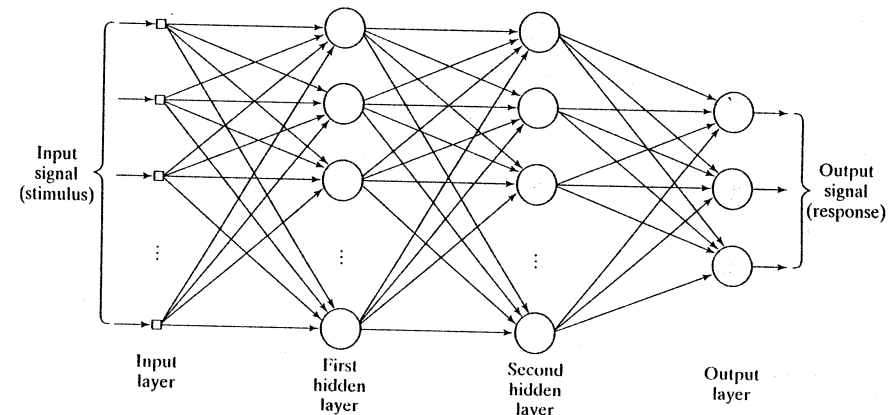


FIGURE 4.1 Architectural graph of a multilayer perceptron with two hidden layers.

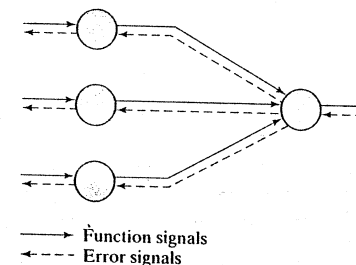


FIGURE 4.2 Illustration of the directions of two basic signal flows in a multilayer perceptron: forward propagation of function signals and back-propagation of error signals.

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 n th training pattern (example) is presented to the network.
- The symbol $\mathcal{E}(n)$ refers to the instantaneous sum of error squares or error energy at iteration n . The average of $\mathcal{E}(n)$ over all values of n (i.e., the entire training set) yields the average error energy $\mathcal{E}_{av}$.
- 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 $\phi_j(\cdot)$.
- 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 i th element of the input vector (pattern) is denoted by $x_i(n)$.
- The k th 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; $l = 0, 1, \dots, L$, 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 n th training example) is defined by

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

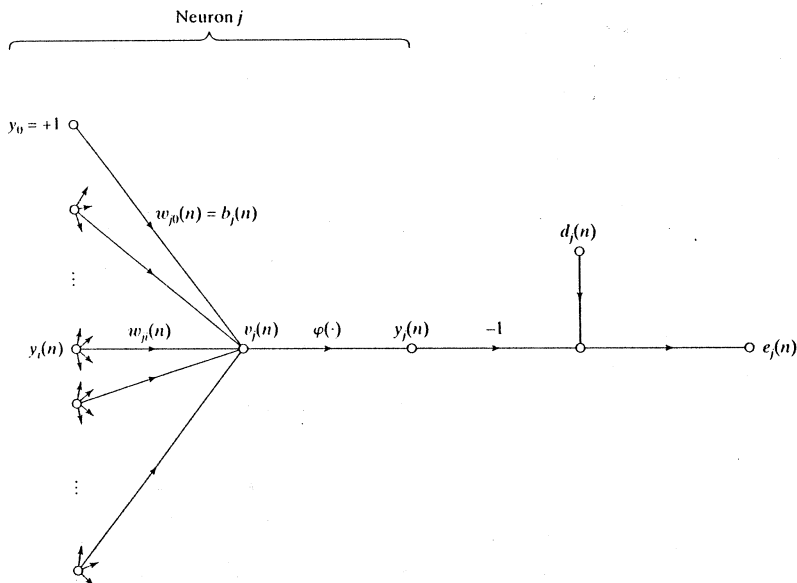
We define the instantaneous value of the error energy for neuron j as $\frac{1}{2}e_j^2(n)$. Correspondingly, the instantaneous value $\mathcal{E}(n)$ of the total error energy is obtained by summing $\frac{1}{2}e_j^2(n)$ over all neurons in the output layer; these are the only "visible" neurons for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

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

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 $\mathcal{E}(n)$ over all n and then normalizing with respect to the set size N , as shown by

$$\mathcal{E}_{av} = \frac{1}{N} \sum_{n=1}^N \mathcal{E}(n) \quad (4.3)$$

The instantaneous error energy $\mathcal{E}(n)$, and therefore the average error energy $\mathcal{E}_{av}$, is a function of all the free parameters (i.e., synaptic weights and bias levels) of the network. For a given training set, $\mathcal{E}_{av}$ 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 $\mathcal{E}_{av}$. 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 $\mathcal{E}_{av}$ 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

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

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

$$y_j(n) = \phi_j(v_j(n)). \quad (4.5)$$

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 $\partial \mathcal{E}(n) / \partial w_{ji}(n)$. According to the *chain rule* of calculus, we may express this gradient as:

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

The partial derivative $\partial \mathcal{E}(n) / \partial w_{ji}(n)$ 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

$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n) \quad (4.7)$$

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

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

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

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \phi'_j(v_j(n)) \quad (4.9)$$

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

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

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

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n) \phi'_j(v_j(n)) y_i(n) \quad (4.11)$$

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

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

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 $\mathcal{E}(n)$). Accordingly, the use of Eq. (4.11) in (4.12) yields

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

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

$$\begin{aligned} \delta_j(n) &= -\frac{\partial \mathcal{E}(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n) \phi'_j(v_j(n)) \end{aligned} \quad (4.14)$$

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 $\phi'_j(v_j(n))$ 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

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \phi'_j(v_j(n)), \quad \text{neuron } j \text{ is hidden.}\end{aligned}\quad (4.15)$$

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

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

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

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \quad (4.17)$$

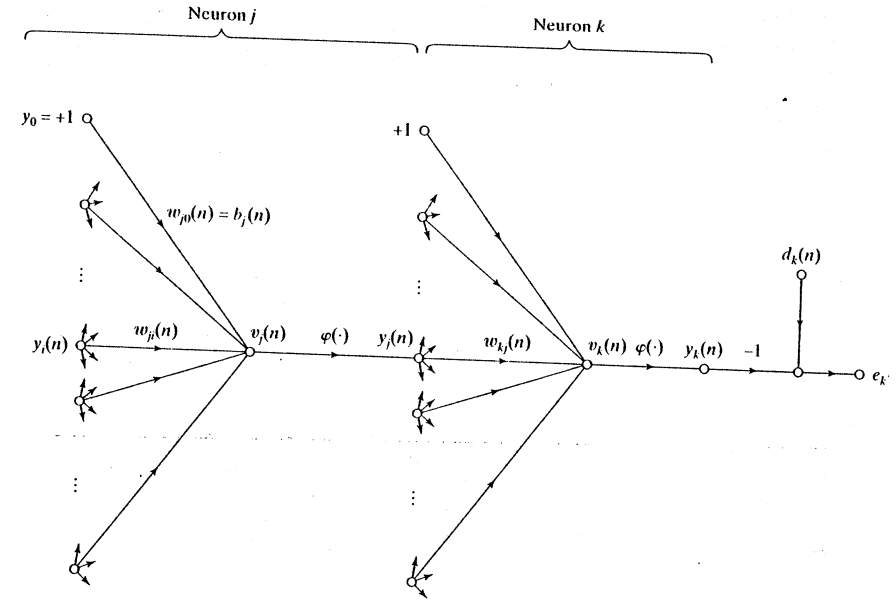


FIGURE 4.4 Signal-flow graph highlighting the details of output neuron k connected to hidden neuron j .

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

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

However, from Fig. 4.4, we note that

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

Hence

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\phi'_k(v_k(n)) \quad (4.20)$$

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

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

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

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

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

$$\begin{aligned} \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} &= - \sum_k e_k(n) \phi'_k(v_k(n)) w_{kj}(n) \\ &= - \sum_k \delta_k(n) w_{kj}(n) \end{aligned} \quad (4.23)$$

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:

$$\delta_j(n) = \phi'_j(v_j(n)) \sum_k \delta_k(n) w_{kj}(n), \quad \text{neuron } j \text{ is hidden} \quad (4.24)$$

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 $\phi'_j(v_j(n))$ 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:

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

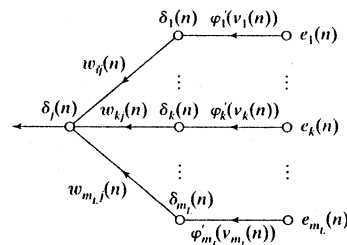


FIGURE 4.5 Signal-flow graph of a part of the adjoint system pertaining to back-propagation of error signals.

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 $\phi'_j(v_j(n))$ 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 $\phi'_j(v_j(n))$ and the weighted sum of the δ s 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

$$y_j(n) = \phi(v_j(n)) \quad (4.26)$$

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

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

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 i th input terminal of the network, for which we write

$$y_i(n) = x_i(n) \quad (4.28)$$

where $x_i(n)$ is the i th 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 j th output terminal of the network, for which we write

$$y_j(n) = o_j(n) \quad (4.29)$$

where $o_j(n)$ is the j th 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 j th 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 δ s for the neurons of the output layer, we next use Eq. (4.24) to compute the δ s 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 $\phi(\cdot)$ associated with that neuron. For this derivative to exist, we require the function $\phi(\cdot)$ 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

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

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 $0 \leq y_j \leq 1$. Differentiating Eq. (4.30) with respect to $v_j(n)$, we get

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

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

$$\phi'_j(v_j(n)) = ay_j(n)[1 - y_j(n)] \quad (4.32)$$

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

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

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

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

Note from Eq. (4.32) that the derivative $\phi'_j(v_j(n))$ 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 $\phi'_j(v_j(n))$, 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

$$\phi_j(v_j(n)) = a \tanh(bv_j(n)), \quad (a, b) > 0 \quad (4.35)$$

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

$$\begin{aligned} \phi'_j(v_j(n)) &= ab \operatorname{sech}^2(bv_j(n)) \\ &= ab(1 - \tanh^2(bv_j(n))) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \end{aligned} \quad (4.36)$$

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

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

For a neuron j in a hidden layer, we have

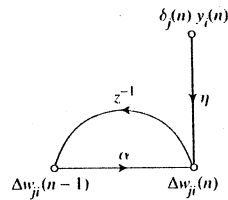
$$\begin{aligned} \delta_j(n) &= \phi'_j(v_j(n)) \sum_k \delta_k(n)w_{kj}(n) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n)w_{kj}(n), \quad \text{neuron } j \text{ is hidden} \end{aligned} \quad (4.38)$$

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)

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

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

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

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

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

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 $\partial \mathcal{E}(t) / \partial w_{ji}(t)$ 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 $\partial \mathcal{E}(t) / \partial w_{ji}(t)$ 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 $(\mathbf{x}(1), \mathbf{d}(1))$, ..., $(\mathbf{x}(N), \mathbf{d}(N))$. 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:

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

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 j th 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

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

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 surface⁴. 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 $\mathcal{E}_{av}(\mathbf{w})$ 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

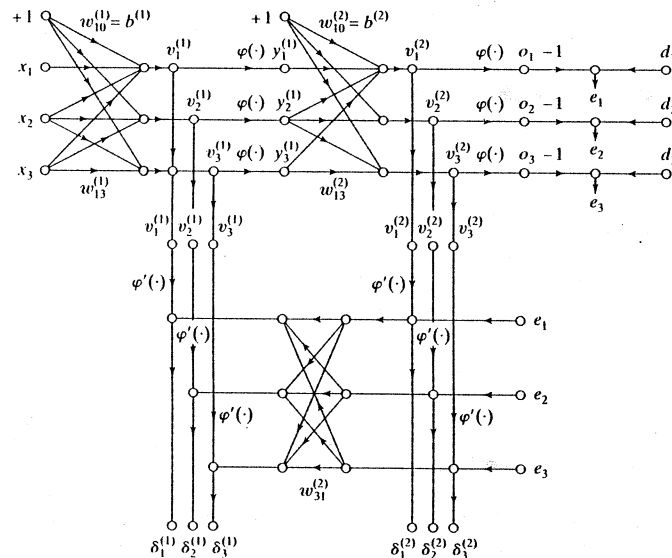


FIGURE 4.7 Signal-flow graphical summary of back-propagation learning. Top part of the graph: forward pass. Bottom part of the graph: backward pass.

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

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

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

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

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 j th 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

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

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

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

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

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:

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

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:

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

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

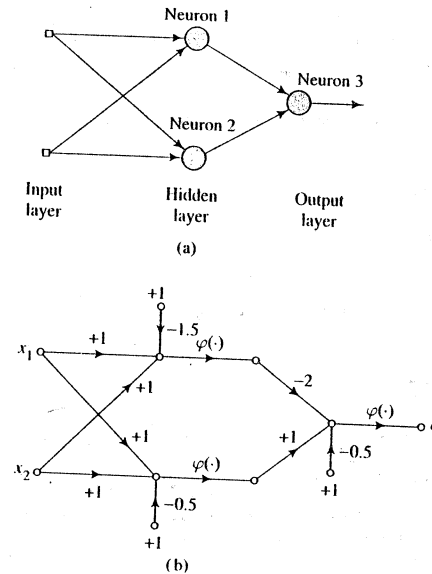


FIGURE 4.8 (a) Architectural graph of network for solving the XOR problem. (b) Signal-flow graph of the network.

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.

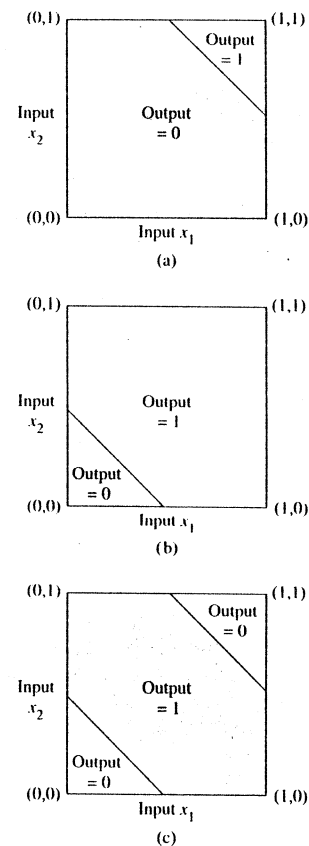


FIGURE 4.9 (a) Decision boundary constructed by hidden neuron 1 of the network in Fig. 4.8. (b) Decision boundary constructed by hidden neuron 2 of the network. (c) Decision boundaries constructed by the complete network.

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

Section 4.6 Heuristics for Making the Back-propagation Algorithm Perform Better 179

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

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

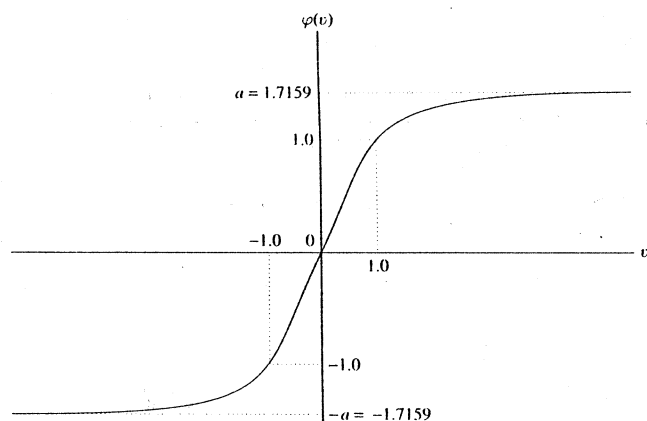
as depicted in Fig. 4.10a. This condition is not satisfied by the standard logistic function depicted in Fig. 4.10b.

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

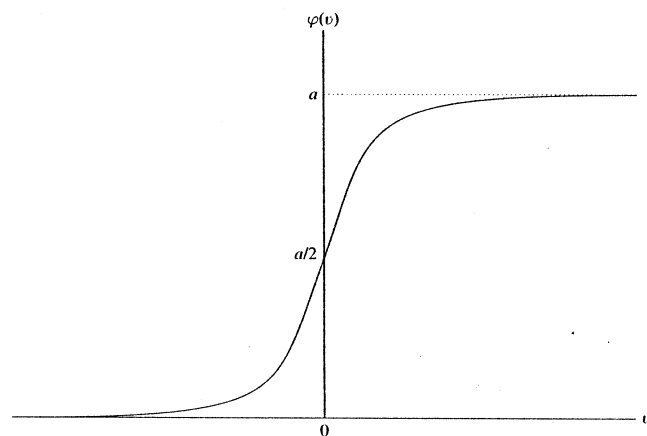
$$\varphi(v) = a \tanh(bv)$$

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

$$a = 1.7159$$



(a)



(b)

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.

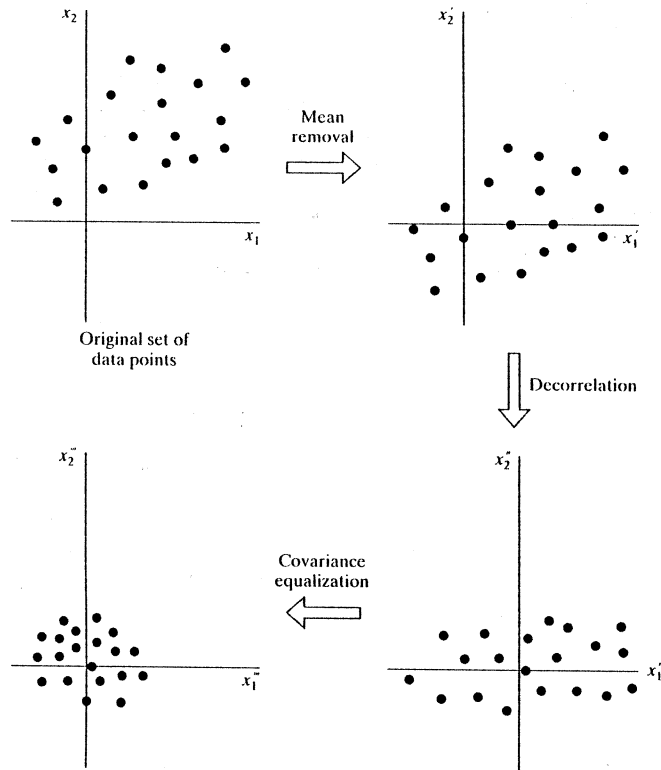


FIGURE 4.11 Illustrating the operation of mean removal, decorrelation, and covariance equalization for a two-dimensional input space.

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

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

and

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

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

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

and variance

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

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

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

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)

$$\sigma_w = m^{-1/2} \quad (4.49)$$

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.

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 j th prototype (i.e., unique sample) of an m -dimensional random vector $\mathbf{x}$ to be classified by a multilayer perceptron. The k th of M possible classes to which $\mathbf{x}$ can belong is denoted by $\mathcal{C}_k$. Let $y_{k,j}$ be the k th output of the network produced in response to the prototype $\mathbf{x}_j$, as shown by

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

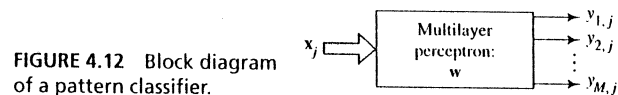


FIGURE 4.12 Block diagram of a pattern classifier.

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

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

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:

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

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:

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

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 incidentally correspond to the upper and lower bounds on the network outputs when using the logistic function), written as:

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

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

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

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):

- The *law of large numbers* is invoked to show that as the size of the training set, N , approaches infinity, the weight vector $\mathbf{w}$ that minimizes cost functional R of Eq. (4.53) approaches the optimum weight vector $\mathbf{w}^*$ that minimizes the expectation of the random quantity $\frac{1}{2} \|\mathbf{d} - \mathbf{F}(\mathbf{w}, \mathbf{x})\|^2$, where $\mathbf{d}$ is the desired response vector and $\mathbf{F}(\mathbf{w}, \mathbf{x})$ is the approximation realized by a multilayer perceptron with weight vector $\mathbf{w}$ and vector $\mathbf{x}$ as input (White, 1989a). The function $\mathbf{F}(\mathbf{w}, \mathbf{x})$, showing explicit dependence on the weight vector $\mathbf{w}$, is the same as $\mathbf{F}(\mathbf{x})$ used previously.
- The optimum weight vector $\mathbf{w}^*$ has the property that the corresponding vector of actual network outputs, $\mathbf{F}(\mathbf{w}^*, \mathbf{x})$, is a mean-squared, error-minimizing approximation to the conditional expectation of the desired response vector, given the input vector $\mathbf{x}$ (White, 1989a). This issue is discussed in Chapter 2.
- For a 1 of M pattern classification problem, the k th element of the desired response vector equals one if the input vector $\mathbf{x}$ belongs to class $\mathcal{C}_k$ and zero otherwise. Hence, the conditional expectation of the desired response vector, given $\mathbf{x}$, equals the *a posteriori* class probability $P(\mathcal{C}_k|\mathbf{x})$, $k = 1, 2, \dots, M$ (Richard and Lippmann, 1991).

It follows therefore that a multilayer perceptron classifier (using the logistic function for nonlinearity) does indeed approximate the *a posteriori* class probabilities, provided that the size of the training set is large enough and that the back-propagation learning process does not get stuck at a local minimum. We may now answer the question we posed earlier. Specifically, we may say that an appropriate output decision rule is the (approximate) Bayes rule generated by the *a posteriori* probability estimates:

Classify the random vector $\mathbf{x}$ as belonging to class $\mathcal{C}_k$ if

$$F_k(\mathbf{x}) > F_j(\mathbf{x}) \quad \text{for all } j \neq k \quad (4.55)$$

where $F_k(\mathbf{x})$ and $F_j(\mathbf{x})$ are elements of the vector-valued mapping function

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} F_1(\mathbf{x}) \\ F_2(\mathbf{x}) \\ \vdots \\ F_M(\mathbf{x}) \end{bmatrix}$$

A unique largest output value exists with probability 1 when the underlying posterior class distributions are distinct. (Here it is assumed that infinite-precision arithmetic is used; ties are possible with finite precision.) This decision rule has the advantage of rendering *unambiguous* decisions over the common *ad hoc* rule of selecting class membership based on the concept of output "firing." That is, the vector $\mathbf{x}$ is assigned membership in a particular class if the corresponding output value is greater than some fixed threshold (usually 0.5 for the logistic form of activation function), which can lead to multiple class assignments.

In Section 4.6 we pointed out that the binary target values $[0,1]$, corresponding to the logistic function of Eq. (4.30), are perturbed by a small amount ϵ as a practical measure, to avoid the saturation of synaptic weights (due to finite numerical precision) during training of the network. As a result of this perturbation, the target values are now nonbinary, and the asymptotic approximations $F_k(\mathbf{x})$ are no longer exactly the *a posteriori* probabilities $P(\mathcal{C}_k|\mathbf{x})$ of the M classes of interest (Hampshire and

Pearlmutter, 1990). Instead, $P(\mathcal{C}_k|\mathbf{x})$ is linearly mapped to the closed interval $[\epsilon, 1 - \epsilon]$, such that $P(\mathcal{C}_k|\mathbf{x}) = 0$ is mapped to an output of ϵ and $P(\mathcal{C}_k|\mathbf{x}) = 1$ is mapped to an output of $1 - \epsilon$. Because this linear mapping preserves relative ordering, it does *not* affect the result of applying the output decision rule of Eq. (4.55).

It is also interesting that when a decision boundary is formed by thresholding the outputs of a multilayer perceptron against some fixed values, the overall shape and orientation of the decision boundary may be explained heuristically (for the case of a single hidden layer) in terms of the number of hidden neurons and the ratios of synaptic weights connected to them (Lui, 1990). Such an analysis, however, is not applicable to a decision boundary formed in accordance with the output decision rule of Eq. (4.55). A more appropriate approach is to consider the hidden neurons as *nonlinear feature detectors* that attempt to map classes from the original input space $\mathbb{R}^m$, where the classes may not be linearly separable, into the space of hidden-layer activations, where it is more likely for them to be linearly separable.

4.8 COMPUTER EXPERIMENT

In this section we use a computer experiment to illustrate the learning behavior of a multilayer perceptron used as a pattern classifier. The objective of the experiment is to distinguish between two classes of "overlapping," two-dimensional, Gaussian-distributed patterns labeled 1 and 2. Let $\mathcal{C}_1$ and $\mathcal{C}_2$ denote the set of events for which a random vector $\mathbf{x}$ belongs to patterns 1 and 2, respectively. We may then express the conditional probability density functions for the two classes:

$$\text{Class } \mathcal{C}_1: \quad f_{\mathbf{x}}(\mathbf{x}|\mathcal{C}_1) = \frac{1}{2\pi\sigma_1^2} \exp\left(-\frac{1}{2\sigma_1^2} \|\mathbf{x} - \boldsymbol{\mu}_1\|^2\right) \quad (4.56)$$

where

$$\boldsymbol{\mu}_1 = \text{mean vector} = [0,0]^T$$

$$\sigma_1^2 = \text{variance} = 1$$

$$\text{Class } \mathcal{C}_2: \quad f_{\mathbf{x}}(\mathbf{x}|\mathcal{C}_2) = \frac{1}{2\pi\sigma_2^2} \exp\left(-\frac{1}{2\sigma_2^2} \|\mathbf{x} - \boldsymbol{\mu}_2\|^2\right) \quad (4.57)$$

where

$$\boldsymbol{\mu}_2 = [2,0]^T$$

$$\sigma_2^2 = 4$$

The two classes are assumed to be equiprobable; that is,

$$p_1 = p_2 = \frac{1}{2}$$

Figure 4.13a shows three-dimensional plots of the two Gaussian distributions defined by Eqs. (4.56) and (4.57). The input vector is $\mathbf{x} = [x_1, x_2]^T$, and the dimensionality of the

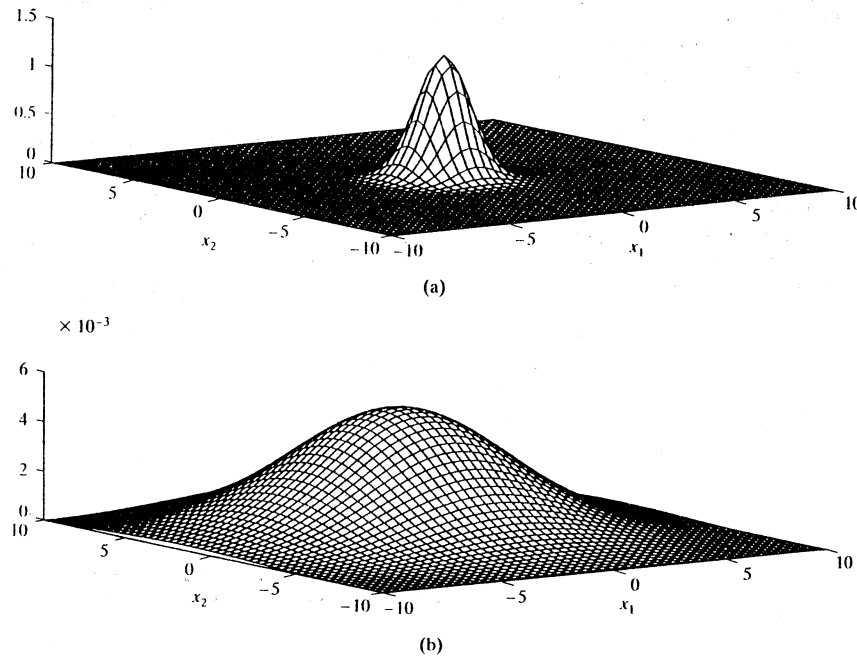


FIGURE 4.13 (a) Probability density function $f_x(x|C_1)$; (b) Probability density function $f_x(x|C_2)$.

input space is $m_0 = 2$. Figure 4.14 shows individual scatter diagrams for classes C_1 and C_2 and the joint scatter diagram representing the superposition of scatter plots of 500 points taken from each of the two processes. This latter diagram clearly shows that the two distributions overlap each other significantly, indicating that there is inevitably a significant probability of misclassification.

Bayesian Decision Boundary

The Bayes criterion for optimum classification is discussed in Chapter 3. Assuming that for a two-class problem, (1) classes C_1 and C_2 are equiprobable, (2) the costs for correct classifications are zero, and (3) the costs for misclassifications are equal, we find that the optimum decision boundary is found by applying the likelihood ratio test:

$$\Lambda(\mathbf{x}) \underset{C_1}{\overset{C_2}{\geq}} \xi \quad (4.58)$$

where $\Lambda(\mathbf{x})$ is the *likelihood ratio*, defined by

$$\Lambda(\mathbf{x}) = \frac{f_x(\mathbf{x}|C_1)}{f_x(\mathbf{x}|C_2)} \quad (4.59)$$

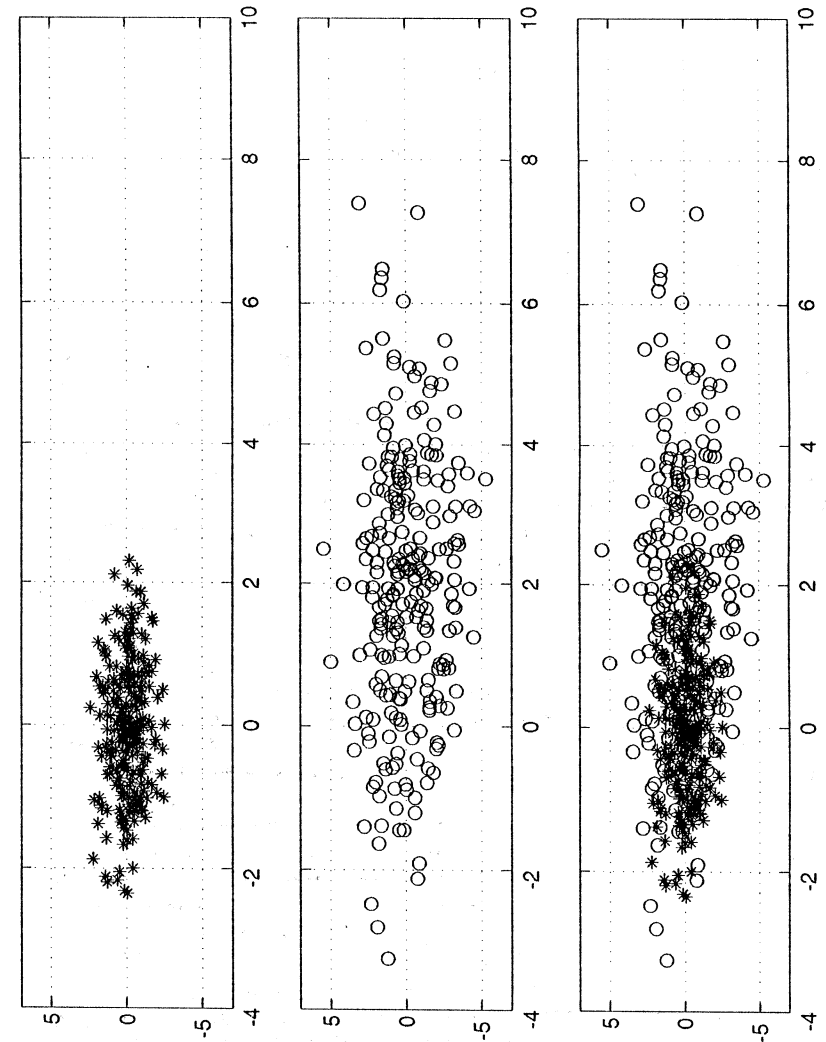


FIGURE 4.14 (a) Scatter plot of class C_1 ; (b) Scatter plot of class C_2 ; (c) Combined scatter plot of both classes.

and ξ is the *threshold of the test*, defined by

$$\xi = \frac{p_2}{p_1} = 1 \quad (4.60)$$

For the example being considered, we have

$$\Lambda(\mathbf{x}) = \frac{\sigma_2^2}{\sigma_1^2} \exp\left(-\frac{1}{2\sigma_1^2} \|\mathbf{x} - \boldsymbol{\mu}_1\|^2 + \frac{1}{2\sigma_2^2} \|\mathbf{x} - \boldsymbol{\mu}_2\|^2\right)$$

The optimum (Bayesian) decision boundary is therefore defined by

$$\frac{\sigma_2^2}{\sigma_1^2} \exp\left(-\frac{1}{2\sigma_1^2} \|\mathbf{x} - \boldsymbol{\mu}_1\|^2 + \frac{1}{2\sigma_2^2} \|\mathbf{x} - \boldsymbol{\mu}_2\|^2\right) = 1$$

or equivalently,

$$\frac{1}{\sigma_2^2} \|\mathbf{x} - \boldsymbol{\mu}_2\|^2 - \frac{1}{\sigma_1^2} \|\mathbf{x} - \boldsymbol{\mu}_1\|^2 = 4 \log\left(\frac{\sigma_1}{\sigma_2}\right) \quad (4.61)$$

Using straightforward manipulations, we may redefine the optimum decision boundary of Eq. (4.61) simply as

$$\|\mathbf{x} - \mathbf{x}_c\|^2 = r^2 \quad (4.62)$$

where

$$\mathbf{x}_c = \frac{\sigma_2^2 \boldsymbol{\mu}_1 - \sigma_1^2 \boldsymbol{\mu}_2}{\sigma_2^2 - \sigma_1^2} \quad (4.63)$$

and

$$r^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_2^2 - \sigma_1^2} \left[\frac{\|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|^2}{\sigma_2^2 - \sigma_1^2} + 4 \log\left(\frac{\sigma_2}{\sigma_1}\right) \right] \quad (4.64)$$

Equation (4.62) represents a circle with center $\mathbf{x}_c$ and radius r . Let Ω_1 define the region lying inside this circle. The Bayesian classification rule for the problem at hand may be stated as follows:

Classify the observation vector $\mathbf{x}$ as belonging to class $\mathcal{C}_1$ if the likelihood ratio $\Lambda(\mathbf{x})$ is greater than the threshold ξ , and to class $\mathcal{C}_2$ otherwise.

For the particular parameters of this experiment, we have a circular decision boundary whose center is located at

$$\mathbf{x}_c = \begin{bmatrix} -2/3 \\ 0 \end{bmatrix}$$

and whose radius is

$$r \approx 2.34$$

Let c denote the set of correct classification outcomes, and e the set of erroneous classification outcomes. The *probability of error* (misclassification), P_e , of a classifier operating according to the Bayesian decision rule is

$$P_e = p_1 P(e|\mathcal{C}_1) + p_2 P(e|\mathcal{C}_2) \quad (4.65)$$

where $P(e|\mathcal{C}_1)$ is the conditional probability of error given that the classifier input vector was drawn from the distribution of class $\mathcal{C}_1$, and similarly for $P(e|\mathcal{C}_2)$; p_1 and p_2 are the *a priori* probabilities of classes $\mathcal{C}_1$ and $\mathcal{C}_2$, respectively. For our problem we may numerically evaluate the probability integrals to obtain

$$P(e|\mathcal{C}_1) \approx 0.1056$$

and

$$P(e|\mathcal{C}_2) \approx 0.2642$$

With $p_1 = p_2 = 1/2$, the probability of misclassification is therefore

$$P_e \approx 0.1849$$

Equivalently, the *probability of correct classification* is

$$P_c = 1 - P_e \\ \approx 0.8151$$

Experimental Determination of Optimal Multilayer Perceptron

Table 4.1 lists the variable parameters of a multilayer perceptron (MLP) that involves a single layer of hidden neurons, and that is trained with the back-propagation algorithm operating in the sequential mode. Since the ultimate objective of a pattern classifier is to achieve an acceptable rate of correct classification, this criterion is used to judge when the variable parameters of the MLP (used as a pattern classifier) are optimal.

Optimal Number of Hidden Neurons. Reflecting practical approaches to the problem of determining the optimal number of hidden neurons, m_1 , the criterion used is the smallest number of hidden neurons that yields a performance "close" to the Bayesian classifier—usually within 1 percent. Thus the experimental study begins with two hidden neurons as the starting point for the simulation results summarized in

TABLE 4.1 Variable Parameters of Multilayer Perceptron

Parameter	Symbol	Typical Range
Number of hidden neurons	m_1	(2, ∞)
Learning-rate parameter	η	(0, 1)
Momentum constant	α	(0, 1)

TABLE 4.2 Simulation Results for Two Hidden Neurons^a

Run Number	Training Set Size	Number of Epochs	Mean-Square Error	Probability of Correct Classification, P_c
1	500	320	0.2375	80.36%
2	2000	80	0.2341	80.33%
3	8000	20	0.2244	80.47%

^aLearning rate parameter $\eta = 0.1$ and momentum $\alpha = 0$.

Table 4.2. Since the purpose of the first set of simulations is merely to ascertain the sufficiency of two hidden neurons or otherwise, the learning-rate parameter η and momentum constant α are arbitrarily set to some nominal values. For each simulation run, a training set of examples, randomly generated from the Gaussian distributions for classes $\mathcal{C}_1$ and $\mathcal{C}_2$ with equal probability, is repeatedly cycled through the network, with each training cycle representing an *epoch*. The number of epochs is chosen so that the total number of training examples used for each run is constant. By so doing, any potential effects arising from variations of the training set sizes are averaged out.

In Table 4.2 and subsequent tables, the *mean-square error* is computed precisely as the error functional defined in Eq. (4.53). We emphasize that the mean-square error is included in these tables only as a matter of record, since a *small mean-square error does not necessarily imply good generalization* (i.e., good performance with data not seen before).

After convergence of a network trained with a total number of N patterns, the probability of correct classification can in theory be calculated as follows:

$$P(c, N) = p_1 P(c, N | \mathcal{C}_1) + p_2 P(c, N | \mathcal{C}_2) \quad (4.66)$$

where $p_1 = p_2 = 1/2$, and

$$P(c, N | \mathcal{C}_1) = \int_{\Omega_1(N)} f_{\mathbf{x}}(\mathbf{x} | \mathcal{C}_1) d\mathbf{x} \quad (4.67)$$

$$P(c, N | \mathcal{C}_2) = 1 - \int_{\Omega_1(N)} f_{\mathbf{x}}(\mathbf{x} | \mathcal{C}_2) d\mathbf{x} \quad (4.68)$$

and $\Omega_1(N)$ is the region in decision space over which the multilayer perceptron (trained with N patterns) classifies the vector $\mathbf{x}$ (representing a realization of the random vector $\mathbf{X}$) as belonging to class $\mathcal{C}_1$. This region is usually found experimentally by evaluating the mapping function learned by the network and then applying the output decision rule of Eq. (4.55). Unfortunately, the numerical evaluation of $P(c, N | \mathcal{C}_1)$ and $P(c, N | \mathcal{C}_2)$ is problematic because closed-form expressions describing the decision boundary $\Omega_1(N)$ cannot easily be found.

Accordingly, we resort to the use of an experimental approach that involves testing the trained multilayer perceptron against another independent set of examples that are again drawn randomly from the distributions for classes $\mathcal{C}_1$ and $\mathcal{C}_2$ with equal

probability. Let A be a random variable that counts the number of patterns out of the N test patterns that are classified correctly. Then the ratio

$$p_N = \frac{A}{N}$$

is a random variable that provides the maximum-likelihood unbiased estimate of the actual classification performance p of the network. Assuming that p is constant over the N input-output pairs, we may apply the *Chernoff bound* (Devroye, 1991) to the estimator p_N of p , obtaining

$$P(|p_N - p| > \epsilon) < 2 \exp(-2\epsilon^2 N) = \delta$$

Application of the Chernoff bound yields $N \approx 26,500$ for $\epsilon = 0.01$, and $\delta = 0.01$ (i.e., 99 percent certainty that the estimate p has the given tolerance). We thus picked a test set of size $N = 32,000$. The last column of Table 4.2 presents the probability of correct classification estimated for this test set size, with each result being the average of 10 independent trials of the experiment.

The classification performance presented in Table 4.2 for a multilayer perceptron using two hidden neurons is already reasonably close to the Bayesian performance $P_c = 81.51$ percent. On this basis we may conclude that for the pattern-classification problem described here the use of two hidden neurons is adequate. To emphasize this conclusion, in Table 4.3 we present the results of simulations repeated for the case of four hidden neurons, with all other parameters held constant. Although the mean-square error in Table 4.3 for four hidden neurons is slightly lower than that in Table 4.2 for two hidden neurons, the average rate of correct classification does not show improvement; in fact, it is slightly worse. For the remainder of the computer experiment described here, the number of hidden neurons is held at two.

Optimal Learning and Momentum Constants. For the "optimal" values of the learning-rate parameter η and momentum constant α , we may use any one of three definitions:

1. The η and α that on average yield convergence to a local minimum in the error surface of the network with the least number of epochs.
2. The η and α that, for either the worst-case or on average, yield convergence to the global minimum in the error surface with the least number of epochs.

TABLE 4.3 Simulation Results for Multilayer Perceptron Using Four Hidden Neurons^a

Run Number	Training Set Size	Number of Epochs	Mean-Square Error	Probability of Correct Classification
1	500	320	0.2199	80.80%
2	2000	80	0.2108	80.81%
3	8000	20	0.2142	80.19%

^aLearning rate parameter $\eta = 0.1$ and momentum constant $\alpha = 0$.

3. The η and α that on average yield convergence to the network configuration that has the best generalization over the entire input space, with the least number of epochs.

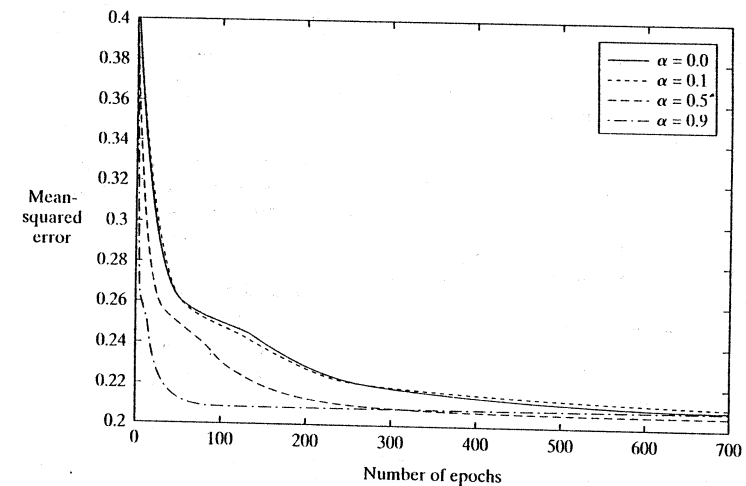
The terms “average” and “worst-case” used here refer to the distribution of the training input–output pairs. Definition 3 is the ideal in practice; however, it is difficult to apply since minimizing the mean-square error is usually the mathematical criterion for optimality during network training and, as stated previously, a lower mean-square error over a training set does not necessarily imply good generalization. From a research point of view, definition 2 is more interesting than definition 1. For example, in Luo (1991), rigorous results are presented for the optimal adaptation of the learning-rate parameter η such that the smallest number of epochs is needed for the multilayer perceptron to approximate the globally optimum synaptic weight matrix to a desired accuracy, albeit for the special case of linear neurons. In general, however, heuristic and experimental procedures dominate the optimal selection of η and α when using definition 1. For the experiment described here, we therefore consider optimality in the sense of definition 1.

Using a multilayer perceptron with two hidden neurons, combinations of learning-rate parameter $\eta \in \{0.01, 0.1, 0.5, 0.9\}$ and momentum constant $\alpha \in \{0.0, 0.1, 0.5, 0.9\}$ are simulated to observe their effect on network convergence. Each combination is trained with the same set of initial random weights and the same set of 500 examples so that the results of the experiment may be compared directly. The learning process was continued for 700 epochs, after which it was terminated; this length of training was considered to be adequate for the back-propagation algorithm to reach a local minimum on the error surface. The ensemble-averaged learning curves so computed are plotted in Figs. 4.15a–4.15d, which are individually grouped by η .

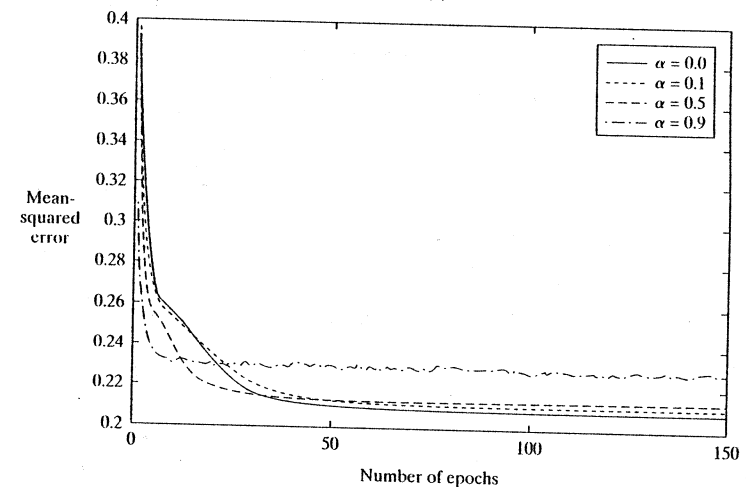
The experimental learning curves shown here suggest the following trends:

- While, in general, a small learning-rate parameter η results in slower convergence, it can locate “deeper” local minima in the error surface than a larger η . This finding is intuitively satisfying, since a smaller η implies that the search for a minimum should cover more of the error surface than would be the case for a larger η .
- For $\eta \rightarrow 0$, the use of $\alpha \rightarrow 1$ produces increasing speed of convergence. On the other hand, for $\eta \rightarrow 1$, the use of $\alpha \rightarrow 0$ is required to ensure learning stability.
- The use of the constants $\eta = \{0.5, 0.9\}$ and $\alpha = 0.9$ causes oscillations in the mean-squared error during learning and a higher value for the final mean-square error at convergence, both of which are undesirable effects.

In Fig. 4.16 we show plots of the “best” learning curves selected from each group of the learning curves plotted in Fig. 4.16, so as to determine an “overall” best learning curve, “best” being defined in the sense of point 1 described previously. From Fig. 4.16, it appears that the optimal learning-rate parameter η_{opt} is about 0.1 and the optimal momentum constant α_{opt} is about 0.5. Thus, Table 4.4 summarizes the “optimal” values of network parameters used in the remainder of the experiment. The fact that the final mean-square error of each curve in Fig. 4.16 does not vary significantly over the range of η and α suggests a “well-behaved” (i.e., relatively smooth) error surface for the problem.



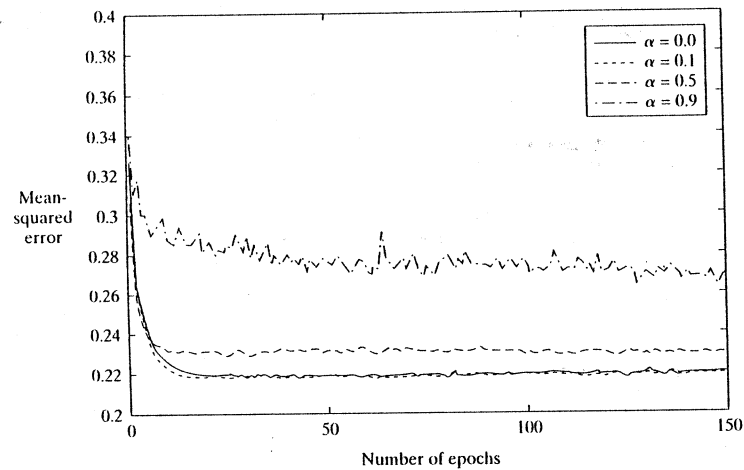
(a)



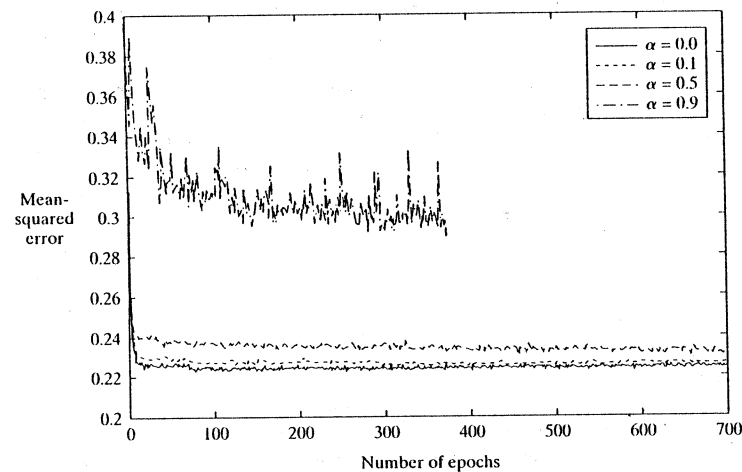
(b)

FIGURE 4.15 Ensemble-averaged learning curves for varying momentum α , and the following values of learning-rate parameters: (a) $\eta = 0.01$, (b) $\eta = 0.1$ (c) $\eta = 0.5$, and (d) $\eta = 0.9$.

(continued on p. 196)



(c)



(d)

FIGURE 4.15 (continued from p. 195)

Evaluation of Optimal Network Design. Given the “optimized” multilayer perceptron having the parameters summarized in Table 4.4, the final network is evaluated to determine its decision boundary, ensemble-averaged learning curve, and probability of correct classification. With finite-size training sets, the network function learned with the optimal parameters is “stochastic” in nature. Accordingly, these performance measures are ensemble-averaged over 20 independently trained networks. Each training set consists

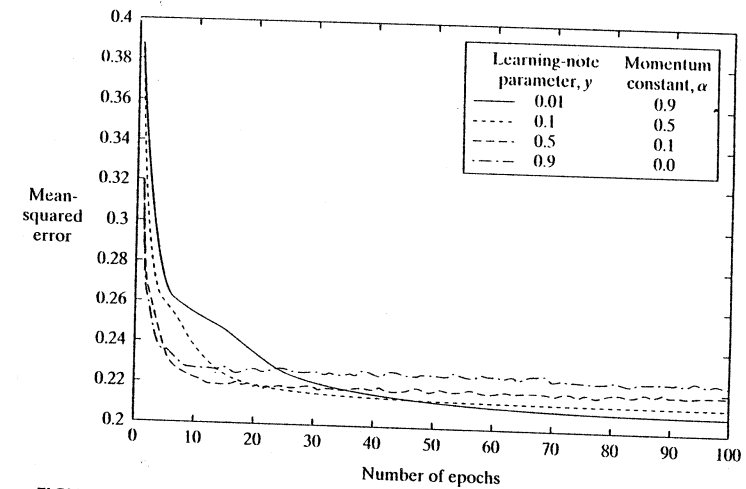


FIGURE 4.16 Best learning curves selected from the four parts of Fig. 4.15.

TABLE 4.4 Configuration of Optimized Multilayer Perceptron

Parameter	Symbol	Value
Optimum number of hidden neurons	m_{opt}	2
Optimum learning-rate parameter	η_{opt}	0.1
Optimum momentum constant	α_{opt}	0.5

of 1000 examples, drawn from the distributions for classes $\mathcal{C}_1$ and $\mathcal{C}_2$ with equal probability, and which are presented to the network in random order. As before, the training was continued for 700 epochs. For the experimental determination of the probabilities of correct classification, the same test set of 32,000 examples used previously is used again.

Figure 4.17a shows three of the “best” decision boundaries for three networks in the ensemble of 20. Figure 4.17b shows three of the “worst” decision boundaries for three other networks in the same ensemble. The shaded (circular) Bayesian decision boundary is included in both figures for reference. From these figures we observe that the decision boundaries constructed by the back-propagation algorithm are convex with respect to the region where they classify the observation vector $\mathbf{x}$ as belonging to class $\mathcal{C}_1$ or class $\mathcal{C}_2$.

The ensemble statistics of the performance measures, probability of correct classification and final mean-squared error, computed over the training sample are listed in Table 4.5. The probability of correct classification for the optimum Bayes classifier is 81.51%.

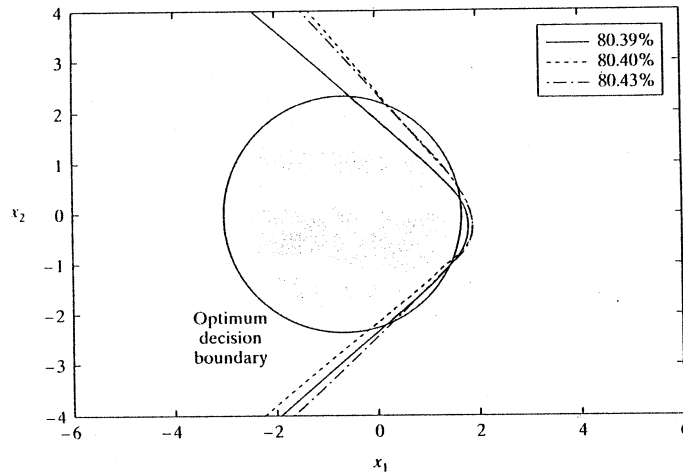


FIGURE 4.17A Plot of three "best" decision boundaries for the classification accuracies: 80.39, 80.40, and 80.43%.

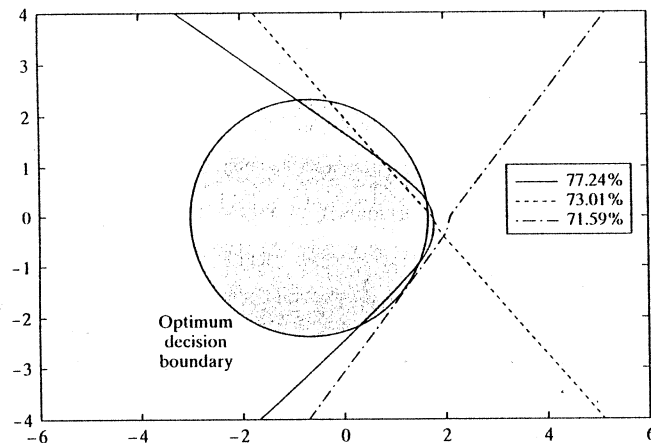


FIGURE 4.17B Plot of three "poorest" decision boundaries for the following classification accuracies: 77.24, 73.01, and 71.59%.

TABLE 4.5 Ensemble Statistics of Performance Measures (Sample Size = 20)

Performance Measure	Mean	Standard Deviation
Probability of correct classification	79.70%	0.44%
Final mean-square error	0.2277	0.0118

4.9 FEATURE DETECTION

Hidden neurons play a critical role in the operation of a multilayer perceptron with back-propagation learning because they act as *feature detectors*. As the learning process progresses, the hidden neurons begin to gradually "discover" the salient features that characterize the training data. They do so by performing a nonlinear transformation on the input data into a new space called the *hidden space*, or *feature space*; these two terminologies are used interchangeably throughout the book. In this new space the classes of interest in a pattern-classification task, for example, may be more easily separated from each other than in the original input space. This statement is well illustrated by the XOR problem considered in Section 4.5.

To put matters into a mathematical context, consider a multilayer perceptron with a single nonlinear layer of m_1 hidden neurons, and a linear layer of $m_2 = M$ output neurons. The choice of linear neurons in the output layer is motivated by the desire to focus attention on the role of hidden neurons on the operation of the multilayer perceptron. Let the synaptic weights of the network be adjusted to minimize the mean-square error between the target output (desired response) and the actual output of the network produced in response to a m_0 -dimensional input vector (pattern), with the ensemble averaging performed over a total of N patterns. Let $z_j(n)$ denote the output of hidden neuron j due to the presentation of input pattern n . The $z_j(n)$ is a nonlinear function of the pattern (vector) applied to the input layer of the network by virtue of the sigmoid activation function built into each hidden neuron.

The output of neuron k in the output layer is

$$y_k(n) = \sum_{j=1}^{m_1} w_{kj} z_j(n), \quad k = 1, 2, \dots, M \quad (4.69)$$

where w_{k0} represents the bias applied to neuron k . The cost function to be minimized is

$$\mathcal{E}_{av} = \frac{1}{2N} \sum_{n=1}^N \sum_{k=1}^M (d_k(n) - y_k(n))^2 \quad (4.70)$$

Note that the use of a batch mode of operation is assumed here. Using Eqs. (4.69) and (4.70), it is easy to reformulate the cost function $\mathcal{E}_{av}$ in the compact matrix form:

$$\mathcal{E}_{av} = \frac{1}{2N} \|\tilde{\mathbf{D}} - \mathbf{W}\tilde{\mathbf{Z}}\|^2 \quad (4.71)$$

where $\mathbf{W}$ is the M -by- m_1 matrix of synaptic weights pertaining to the output layer of the network. The matrix $\tilde{\mathbf{Z}}$ is the m_1 -by- N matrix of hidden neuronal outputs (with their mean values subtracted off), which are produced by the individual N input patterns applied to the input layer of the network; that is,

$$\tilde{\mathbf{Z}} = [(z_j(n) - \mu_{z_j}); \quad j = 1, 2, \dots, m_1; \quad n = 1, 2, \dots, N]$$

where μ_{z_j} is the mean value of $z_j(n)$. Correspondingly, the matrix $\tilde{\mathbf{D}}$ is the M -by- N matrix of target patterns (desired responses) presented to the output layer of the network; that is,

$$\tilde{\mathbf{D}} = [(d_k(n) - \mu_{d_k}); \quad k = 1, 2, \dots, M; \quad n = 1, 2, \dots, N]$$

where μ_{d_k} is the mean value of $d_k(n)$. The minimization of $\mathcal{E}_{av}$ defined by Eq. (4.70) is recognized as a linear least-squares problem, the solution of which is given by

$$\mathbf{W} = \tilde{\mathbf{D}} \tilde{\mathbf{Z}}^+ \quad (4.72)$$

where $\tilde{\mathbf{Z}}^+$ is the pseudo-inverse of matrix $\tilde{\mathbf{Z}}$. The minimum value of $\mathcal{E}_{av}$ is given by (see Problem 4.7)

$$\mathcal{E}_{av, \min} = \frac{1}{2N} \text{tr} \left[\tilde{\mathbf{D}} \tilde{\mathbf{D}}^T \tilde{\mathbf{D}} \tilde{\mathbf{Z}}^T (\tilde{\mathbf{Z}} \tilde{\mathbf{Z}}^T)^+ \tilde{\mathbf{Z}} \tilde{\mathbf{D}}^T \right] \quad (4.73)$$

where $\text{tr}[\cdot]$ denotes the trace operator. Since the target patterns represented by matrix $\tilde{\mathbf{D}}$ are all fixed, minimization of the cost function $\mathcal{E}_{av}$ with respect to the synaptic weights of the multilayer perceptron is equivalent to maximizing the *discriminant function* (Webb and Lowe, 1990)

$$\mathcal{D} = \text{tr}[\mathbf{C}_b \mathbf{C}_t^+] \quad (4.74)$$

where the matrices $\mathbf{C}_b$ and $\mathbf{C}_t$ are themselves defined as:

- The m_1 -by- m_1 matrix $\mathbf{C}_t$ is the *total covariance matrix* of the hidden neuronal outputs due to the presentations of N input patterns:

$$\mathbf{C}_t = \tilde{\mathbf{Z}} \tilde{\mathbf{Z}}^T \quad (4.75)$$

The matrix $\mathbf{C}_t^+$ is the pseudo-inverse of matrix $\mathbf{C}_t$.

- The m_1 -by- m_1 matrix $\mathbf{C}_b$ is defined by

$$\mathbf{C}_b = \tilde{\mathbf{Z}} \tilde{\mathbf{D}}^T \tilde{\mathbf{D}} \tilde{\mathbf{Z}}^T \quad (4.76)$$

Note that the discriminant function $\mathcal{D}$ defined in Eq. (4.74) is determined entirely by the hidden neurons of the multilayer perceptron. There is also no restriction on the number of hidden layers constituting the nonlinear transformation responsible for generating the discriminant function $\mathcal{D}$. In a multilayer perceptron with more than one hidden layer, the matrix $\tilde{\mathbf{Z}}$ refers to the entire set of patterns in the space defined by the final layer of hidden neurons.

For an interpretation of matrix $\mathbf{C}_b$, consider the specific choice of a *one-from- M coding scheme* (Webb and Lowe, 1990). That is, the target value (desired response) in a particular pattern is unity if the chosen pattern belongs to that class, and zero otherwise, as shown in (see page 185)

$$d(n) = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow k\text{th element,} \quad d(n) \in \mathcal{C}_k$$

Thus, if there are M classes, $\mathcal{C}_k$, $k = 1, 2, \dots, M$ with N_k patterns in class $\mathcal{C}_k$ and

$$\sum_{k=1}^M N_k = N$$

we may then expand the matrix $\mathbf{C}_b$ for this particular coding scheme into the form

$$\mathbf{C}_b = \sum_{k=1}^M N_k^2 (\mu_{z,k} - \mu_z)(\mu_{z,k} - \mu_z)^T \quad (4.77)$$

where the m_1 -by-1 vector $\mu_{z,k}$ is the mean value of the vector of the hidden neuronal outputs over all N_k patterns in class $\mathcal{C}_k$, and the vector μ_z is the mean value of the vector of hidden neuronal outputs over all N input presentations. According to Eq. (4.77), we may interpret $\mathbf{C}_b$ as the *weighted between-class covariance matrix* at the outputs of the hidden layer.

Thus, for a one-from- M coding scheme, the multilayer perceptron maximizes a discriminant function that is the trace of the product of two matrices: the weighted between-class covariance matrix and the pseudo-inverse of the total covariance matrix. This result is interesting because it illustrates how a multilayer perceptron with back-propagation learning incorporates the proportions of samples within individual classes as *priors*.

Relation to Fisher's Linear Discriminant

The discriminant function $\mathcal{D}$ defined in Eq. (4.74) is unique to multilayer perceptrons. It bears a close resemblance to *Fisher's linear discriminant*, which describes a linear transformation from a multidimensional problem to a one-dimensional problem. Consider a variable y formed as a linear combination of the elements of an input vector $\mathbf{x}$; that is, it is defined as the inner product of $\mathbf{x}$ and a vector of adjustable parameters, $\mathbf{w}$ (that includes a bias as its first element), as shown by

$$y = \mathbf{w}^T \mathbf{x}$$

The vector $\mathbf{x}$ is drawn from one of two populations, $\mathcal{C}_1$ and $\mathcal{C}_2$, which differ from each other by virtue of their mean vectors μ_1 and μ_2 , respectively. The *Fisher criterion* for discriminating between these two classes is defined by

$$J(\mathbf{w}) = \frac{\mathbf{w}^T \mathbf{C}_b \mathbf{w}}{\mathbf{w}^T \mathbf{C}_t \mathbf{w}}$$

where $\mathbf{C}_b$ is the *between-class covariance matrix* defined by

$$\mathbf{C}_b = (\mu_2 - \mu_1)(\mu_2 - \mu_1)^T$$

and $\mathbf{C}_t$ is the *total within-class covariance matrix* defined by

$$\mathbf{C}_t = \sum_{n \in \mathcal{C}_1} (\mathbf{x}_n - \mu_1)(\mathbf{x}_n - \mu_1)^T + \sum_{n \in \mathcal{C}_2} (\mathbf{x}_n - \mu_2)(\mathbf{x}_n - \mu_2)^T$$

The within-class covariance matrix C_i is proportional to the sample covariance matrix of the training set. It is symmetric and nonnegative definite, and is usually nonsingular if the size of the training set is large. The between-class covariance matrix C_b is also symmetric and nonnegative definite, but singular. A property of particular interest is that the matrix product $C_b \mathbf{w}$ is always in the direction of the difference mean vector $\mu_1 - \mu_2$. This property follows directly from the definition of C_b .

The expression defining $J(\mathbf{w})$ is known as the *generalized Rayleigh quotient*. The vector $\mathbf{w}$ that maximizes $J(\mathbf{w})$ must satisfy the condition

$$C_b \mathbf{w} = \lambda C_i \mathbf{w} \quad (4.76)$$

Equation (4.76) is a generalized eigenvalue problem. Recognizing that in our case the matrix product $C_b \mathbf{w}$ is always in the direction of the difference vector $\mu_1 - \mu_2$, we find that the solution for Eq. (4.76) is simply

$$\mathbf{w} = C_i^{-1} (\mu_1 - \mu_2) \quad (4.77)$$

which is referred to as *Fisher's linear discriminant* (Duda and Hart, 1973).

Returning to the issue of feature detection, recall that the discriminant function $\mathcal{G}$ of Eq. (4.74) relates the between-class and total covariance matrices of patterns transformed into the hidden space of the network. The discriminant function $\mathcal{G}$ plays a role similar to that of Fisher's linear discriminant. This is precisely the reason why these neural networks are able to perform the task of pattern classification so well.

0 BACK-PROPAGATION AND DIFFERENTIATION

Back-propagation is a specific technique for implementing *gradient descent* in weight space for a multilayer feedforward network. The basic idea is to efficiently compute *partial derivatives* of an approximating function $F(\mathbf{w}, \mathbf{x})$ realized by the network with respect to all the elements of the adjustable weight vector $\mathbf{w}$ for a given value of input vector $\mathbf{x}$. Herein lies the computational power of the back-propagation algorithm.⁵

To be specific, consider a multilayer perceptron with an input layer of m_0 nodes, two hidden layers, and a single output neuron, as depicted in Fig. 4.18. The elements of the weight vector $\mathbf{w}$ are ordered by layer (starting from the first hidden layer), then by neurons in a layer, and then by the number of a synapse within a neuron. Let $w_{ij}^{(l)}$ denote the synaptic weight from neuron i to neuron j in layer $l = 0, 1, 2, \dots$. For $l = 1$, corresponding to the first hidden layer, the index i refers to a source node rather than to a neuron. For $l = 3$, corresponding to the output layer in Fig. 4.18, we have $j = 1$. We wish to evaluate the derivatives of the function $F(\mathbf{w}, \mathbf{x})$ with respect to all the elements of the weight vector $\mathbf{w}$, for a specified input vector $\mathbf{x} = [x_1, x_2, \dots, x_{m_0}]^T$. Note that for $l = 2$ (i.e., a single hidden layer), the function $F(\mathbf{w}, \mathbf{x})$ has a form similar to that on the right-hand side of Eq. (4.69). We have included the weight vector $\mathbf{w}$ as an argument of the function F to focus attention on it.

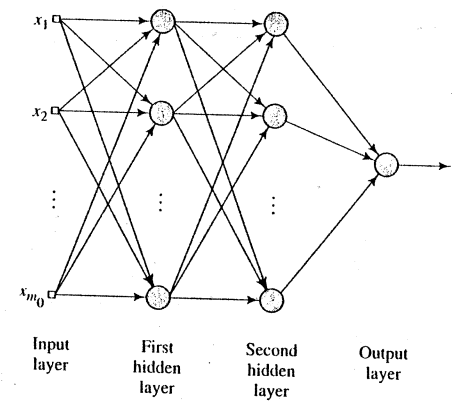


FIGURE 4.18 Multilayer perceptron with two hidden layers and one output neuron.

The multilayer perceptron of Fig. 4.18 is parameterized by an *architecture* $\mathcal{A}$ (representing a discrete parameter) and a *weight vector* $\mathbf{w}$ (made up of continuous elements). Let $\mathcal{A}_j^{(l)}$ denote that part of the architecture extending from the input layer ($l = 0$) to node j in layer $l = 1, 2, 3$. Accordingly, we may write

$$F(\mathbf{w}, \mathbf{x}) = \varphi(\mathcal{A}_1^{(3)}) \quad (4.80)$$

where φ is the activation function. However, $\mathcal{A}_1^{(3)}$ is to be interpreted merely as an architectural symbol rather than a variable. Thus, adapting Eqs. (4.1), (4.2), (4.11), and (4.23) for use in this situation, we obtain the following results:

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{ik}^{(3)}} = \varphi'(\mathcal{A}_1^{(3)}) \varphi(\mathcal{A}_k^{(2)}) \quad (4.81)$$

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{kj}^{(2)}} = \varphi'(\mathcal{A}_1^{(3)}) \varphi'(\mathcal{A}_k^{(2)}) \varphi(\mathcal{A}_j^{(1)}) w_{ik}^{(3)} \quad (4.82)$$

$$\frac{\partial F(\mathbf{w}, \mathbf{x})}{\partial w_{ji}^{(1)}} = \varphi'(\mathcal{A}_1^{(3)}) \varphi'(\mathcal{A}_j^{(1)}) x_i \left[\sum_k w_{ik}^{(3)} \varphi'(\mathcal{A}_k^{(2)}) w_{kj}^{(2)} \right] \quad (4.83)$$

where φ' is the partial derivative of the nonlinearity φ with respect to its input, and x_i is the i th element of the input vector $\mathbf{x}$. In a similar way we may derive the equations for the partial derivatives of a general network with more hidden layers and more neurons in the output layer.

Equations (4.81) through (4.83) provide the basis for calculating the sensitivity of the network function $F(\mathbf{w}, \mathbf{x})$ with respect to variations in the elements of the weight vector $\mathbf{w}$. Let ω denote an element of the weight vector $\mathbf{w}$. The *sensitivity* of $F(\mathbf{w}, \mathbf{x})$ with respect to ω is formally defined by

$$S_{\omega}^F = \frac{\partial F / F}{\partial \omega / \omega}, \quad \omega \in \mathbf{w}$$

It is for this reason that we referred to the lower part of the signal-flow graph in Fig. 4.7 as a "sensitivity graph."

Jacobian Matrix

Let W denote the total number of free parameters (i.e., synaptic weights and biases) of a multilayer perceptron, which are ordered in the manner described to form the weight vector $\mathbf{w}$. Let N denote the total number of examples used to train the network. Using back-propagation we may compute a set of W partial derivatives of the approximating function $F[\mathbf{w}, \mathbf{x}(n)]$ with respect to the elements of the weight vector $\mathbf{w}$ for a specific example $\mathbf{x}(n)$ in the training set. Repeating these computations for $n = 1, 2, \dots, N$, we end up with an N -by- W matrix of partial derivatives. This matrix is called the *Jacobian* $\mathbf{J}$ of the multilayer perceptron evaluated at $\mathbf{x}(n)$. Each row of the Jacobian corresponds to a particular example in the training set.

There is experimental evidence to suggest that many neural network training problems are intrinsically *ill-conditioned*, leading to a Jacobian $\mathbf{J}$ that is almost rank deficient (Saarinen et al., 1991). The *rank* of a matrix is equal to the number of linearly independent columns or rows in the matrix, whichever one is smallest. The Jacobian $\mathbf{J}$ is said to be *rank-deficient* if its rank is less than $\min(N, W)$. Any rank deficiency in the Jacobian causes the back-propagation algorithm to obtain only partial information of the possible search directions, and also causes training times to be long.

HESSIAN MATRIX

The *Hessian matrix* of the cost function $\mathcal{E}_{av}(\mathbf{w})$, denoted by $\mathbf{H}$, is defined as the second derivative of $\mathcal{E}_{av}(\mathbf{w})$ with respect to the weight vector $\mathbf{w}$, as shown by

$$\mathbf{H} = \frac{\partial^2 \mathcal{E}_{av}(\mathbf{w})}{\partial \mathbf{w}^2} \quad (4.84)$$

The Hessian matrix plays an important role in the study of neural networks; specifically, we may mention the following:⁶

1. The eigenvalues of the Hessian matrix have a profound influence on the dynamics of back-propagation learning.
2. The inverse of the Hessian matrix provides a basis for pruning (i.e., deleting) insignificant synaptic weights from a multilayer perceptron, as discussed in Section 4.15.
3. The Hessian matrix is basic to the formulation of second-order optimization methods as an alternative to back-propagation learning, as discussed in Section 4.18.

An iterative procedure for the computation⁷ of the Hessian matrix is presented in Section 4.15. In this section we confine our attention to point 1.

In Chapter 3 we indicated that the eigenstructure of the Hessian matrix has a profound influence on the convergence properties of the LMS algorithm. So it is also with the back-propagation algorithm, but in a much more complicated way. Typically the Hessian matrix of the error surface pertaining to a multilayer perceptron trained with the back-propagation algorithm has the following composition of eigenvalues (LeCun, et al., 1991; LeCun, 1993):

- A small number of small eigenvalues.
- A large number of medium-sized eigenvalues.
- A small number of large eigenvalues.

The factors affecting this composition may be grouped as follows:

- Nonzero-mean input signals or nonzero-mean induced neuronal output signals.
- Correlations between the elements of the input signal vector and correlations between induced neuronal output signals.
- Wide variations in the second derivatives of the cost function with respect to synaptic weights of neurons in the network, as we proceed from one layer to the next. The second derivatives are often smaller in the lower layers, with the synaptic weights in the first hidden layer learning slowly and those in the last layers learning quickly.

From Chapter 3 we recall that the *learning time* of the LMS algorithm is sensitive to variations in the condition number $\lambda_{\max}/\lambda_{\min}$, where $\lambda_{\max}$ is the largest eigenvalue of the Hessian and $\lambda_{\min}$ is its smallest nonzero eigenvalue. Experimental results show that a similar result holds for the back-propagation algorithm, which is a generalization of the LMS algorithm. For inputs with nonzero mean, the ratio $\lambda_{\max}/\lambda_{\min}$ is larger than its corresponding value for zero-mean inputs: the larger the mean of the inputs, the larger the ratio $\lambda_{\max}/\lambda_{\min}$ (see Problem 3.10). This observation has a serious implication for the dynamics of back-propagation learning.

For the learning time to be minimized, the use of nonzero-mean inputs should be avoided. Now, insofar as the signal vector $\mathbf{x}$ applied to a neuron in the first hidden layer of a multilayer perceptron (i.e., the signal vector applied to the input layer) is concerned, it is easy to remove the mean from each element of $\mathbf{x}$ before its application to the network. But what about the signals applied to the neurons in the remaining hidden and output layers of the network? The answer to this question lies in the type of activation function used in the network. If the activation function is nonsymmetric, as in the case of the logistic function, the output of each neuron is restricted to the interval $[0, 1]$. Such a choice introduces a source of *systematic bias* for those neurons located beyond the first hidden layer of the network. To overcome this problem we need to use an antisymmetric activation function such as the hyperbolic tangent function. With this latter choice, the output of each neuron is permitted to assume both positive and negative values in the interval $[-1, 1]$, in which case it is likely for its mean to be zero. If the network connectivity is large, back-propagation learning with antisymmetric activation functions can yield faster convergence than a similar process with nonsymmetric activation functions, for which there is also empirical evidence (LeCun et al., 1991). This provides justification for heuristic 3 described in Section 4.6.

4.12 GENERALIZATION

In back-propagation learning, we typically start with a training sample and use the back-propagation algorithm to compute the synaptic weights of a multilayer perceptron by loading (encoding) as many of the training examples as possible into the network. The hope is that the neural network so designed will generalize. 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; 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.

The learning process (i.e., training of a neural network) may be viewed as a “curve-fitting” problem. The network itself may be considered simply as a nonlinear input–output mapping. Such a viewpoint then permits us to look on generalization not as a mystical property of neural networks but rather simply as the effect of a good nonlinear interpolation of the input data (Wieland and Leighton, 1987). The network performs useful interpolation primarily because multilayer perceptrons with continuous activation functions lead to output functions that are also continuous.

Figure 4.19a illustrates how generalization may occur in a hypothetical network. The nonlinear input–output mapping represented by the curve depicted in this figure is computed by the network as a result of learning the points labeled as “training data.” The point marked on the curve as “generalization” is thus 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 train the network, as illustrated in the figure. When, however, 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 (due to noise, for example) 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 is actually necessary, with the result that undesired contributions in the input space due to noise are stored in synaptic weights of the network. An example of how poor generalization due to memorization in a neural network may occur is illustrated in Fig. 4.19b for the same data depicted in Fig. 4.19a. “Memorization” is essentially a “look-up table,” which implies that the input–output mapping computed by the neural network is not smooth. As pointed out in Poggio and Girosi (1990a), smoothness of input–output mapping is closely related to such model-selection criteria as the *Occam’s razor*, the essence of which is to select the “simplest” function in the absence of any prior knowledge to the contrary. In the context of our present discussion, the simplest function means the smoothest function that approximates the mapping for a given error criterion, because such a choice generally demands the fewest computational resources. Smoothness is also natural in many applications, depending on the scale of the phenomenon being studied. It is therefore important 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 (Wieland and Leighton, 1987).

Sufficient Training Set Size for a Valid Generalization

Generalization is influenced by three factors: (1) the size of the training set, and how representative it is of the environment of interest, (2) the architecture of the neural network, and (3) the physical complexity of the problem at hand. Clearly, we have no

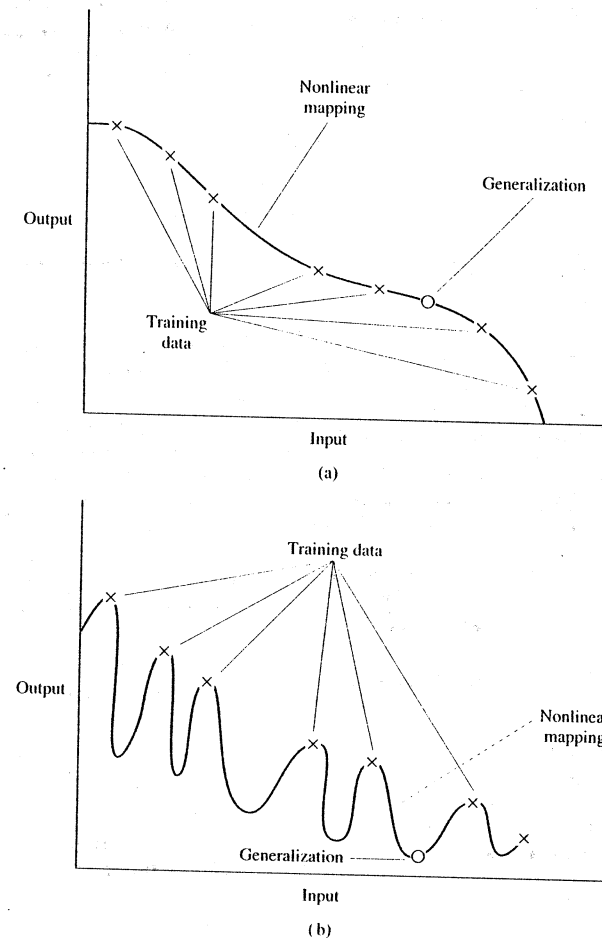


FIGURE 4.19 (a) Properly fitted data (good generalization) (b) Overfitted data (poor generalization).

control over the latter. In the context of the other two factors, we may view the issue of generalization from two different perspectives (Hush and Horne, 1993):

- The architecture of the network is fixed (hopefully in accordance with the physical complexity of the underlying problem), and the issue to be resolved is that of determining the size of the training set needed for a good generalization to occur.
- The size of the training set is fixed, and the issue of interest is that of determining the best architecture of network for achieving good generalization.

Both of these viewpoints are valid in their own individual ways. In the present discussion we focus on the first viewpoint.

The adequacy of the training sample size or the sample complexity problem is discussed in Chapter 2. As pointed out in that chapter, the VC dimension provides the theoretical basis for a principled solution to this important design problem. In particular, we have *distribution-free, worst-case* formulas for estimating the size of the training sample that is sufficient for a good generalization performance; see Section 2.14. Unfortunately, we often find that there is a huge numerical gap between the size of the training sample actually needed and that predicted by these formulas. It is this gap that has made the sample complexity problem a continuing open research area.

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

$$N = O\left(\frac{W}{\epsilon}\right) \quad (4.85)$$

where W is the total number of free parameters (i.e., synaptic weights and biases) in the network, and ϵ denotes the fraction of classification errors permitted on test data (as in pattern classification), and $O(\cdot)$ denotes the order of quantity enclosed within. For example, with an error of 10 percent the number of training examples needed should be about 10 times the number of free parameters in the network.

Equation (4.85) is in accordance with *Widrow's rule of thumb* for the LMS algorithm, which states that the settling time for adaptation in linear adaptive temporal filtering is approximately equal to the memory span of an adaptive tapped-delay-line filter divided by the misadjustment (Widrow and Stearns, 1985). The misadjustment in the LMS algorithm plays a role somewhat analogous to the error ϵ in Eq. (4.85). Further justification for this empirical rule is presented in the next section.

3 APPROXIMATIONS OF FUNCTIONS

A multilayer perceptron trained with the back-propagation algorithm may be viewed as a practical vehicle for performing a *nonlinear input-output mapping* of a general nature. To be specific, let m_0 denote the number of input (source) nodes of a multilayer perceptron, and let $M = m_L$ denote the number of neurons in the output layer of the network. The input-output relationship of the network defines a mapping from an m_0 -dimensional Euclidean input space to an M -dimensional Euclidean output space, which is infinitely continuously differentiable when the activation function is likewise. In assessing the capability of the multilayer perceptron from this viewpoint of input-output mapping, the following fundamental question arises:

What is the minimum number of hidden layers in a multilayer perceptron with an input-output mapping that provides an approximate realization of any continuous mapping?

Universal Approximation Theorem

The answer to this question is embodied in the *universal approximation theorem*⁸ for a nonlinear input-output mapping, which may be stated as:

Let $\phi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_{m_0} denote the m_0 -dimensional unit hypercube $[0, 1]^{m_0}$. The space of continuous functions on I_{m_0}

is denoted by $C(I_{m_0})$. Then, given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, there exist an integer M and sets of real constants α_i , b_i , and w_{ij} , where $i = 1, \dots, m_1$ and $j = 1, \dots, m_0$ such that we may define

$$F(x_1, \dots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \phi\left(\sum_{j=1}^{m_0} w_{ij} x_j + b_i\right) \quad (4.86)$$

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

$$|F(x_1, \dots, x_{m_0}) - f(x_1, \dots, x_{m_0})| < \epsilon$$

for all $x_1, x_2, \dots, x_{m_0}$ that lie in the input space.

The universal approximation theorem is directly applicable to multilayer perceptrons. We first note that the logistic function $1/[1 + \exp(-v)]$ used as the nonlinearity in a neuronal model for the construction of a multilayer perceptron is indeed a non-constant, bounded, and monotone-increasing function; it therefore satisfies the conditions imposed on the function $\phi(\cdot)$. Next, we note that Eq. (4.86) represents the output of a multilayer perceptron described as follows:

1. The network has m_0 input nodes and a single hidden layer consisting of m_1 neurons; the inputs are denoted by $x_1, \dots, x_{m_0}$.
2. Hidden neuron i has synaptic weights $w_{i1}, \dots, w_{im_0}$, and bias b_i .
3. The network output is a linear combination of the outputs of the hidden neurons, with $\alpha_1, \dots, \alpha_{m_1}$ defining the synaptic weights of the output layer.

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 exact representation. Equation (4.86), which is the backbone of the theorem, merely generalizes approximations by finite Fourier series. In effect, the theorem states that *a single hidden layer is sufficient for a multilayer perceptron to compute a uniform ϵ approximation to a given training set represented by the set of inputs $x_1, \dots, x_{m_0}$ and a desired (target) output $f(x_1, \dots, x_{m_0})$* . However, the theorem does not say that a single hidden layer is optimum in the sense of learning time, ease of implementation, or (more importantly) generalization.

Bounds on Approximation Errors

Barron (1993) has established the approximation properties of a multilayer perceptron, assuming that the network has a single layer of hidden neurons using sigmoid functions and a linear output neuron. The network is trained using the back-propagation algorithm and then tested with new data. During training, the network learns specific points of a target function f in accordance with the training data, and thereby produces the approximating function F defined in Eq. (4.86). When the network is exposed to test data that have not been seen before, the network function F acts as an "estimator" of new points of the target function; that is, $F = \hat{f}$.

A smoothness property of the target function f is expressed in terms of its Fourier representation. In particular, the average of the norm of the frequency vector weighted by the Fourier magnitude distribution is used as a measure for the extent to which the function f oscillates. Let $\hat{f}(\omega)$ denote the multidimensional Fourier transform

of the function $f(\mathbf{x})$, $\mathbf{x} \in \mathbb{R}^{m_0}$; the m_0 -by-1 vector $\boldsymbol{\omega}$ is the frequency vector. The function $f(\mathbf{x})$ is defined in terms of its Fourier transform $\tilde{f}(\boldsymbol{\omega})$ by the inverse formula:

$$f(\mathbf{x}) = \int_{\mathbb{R}^{m_0}} \tilde{f}(\boldsymbol{\omega}) \exp(j\boldsymbol{\omega}^T \mathbf{x}) d\boldsymbol{\omega} \quad (4.87)$$

where $j = \sqrt{-1}$. For the complex-valued function $\tilde{f}(\boldsymbol{\omega})$ for which $\boldsymbol{\omega} \tilde{f}(\boldsymbol{\omega})$ is integrable, we define the *first absolute moment* of the Fourier magnitude distribution of the function f as:

$$C_f = \int_{\mathbb{R}^{m_0}} |\tilde{f}(\boldsymbol{\omega})| \times \|\boldsymbol{\omega}\|^{1/2} d\boldsymbol{\omega} \quad (4.88)$$

where $\|\boldsymbol{\omega}\|$ is the Euclidean norm of $\boldsymbol{\omega}$ and $|\tilde{f}(\boldsymbol{\omega})|$ is the absolute value of $\tilde{f}(\boldsymbol{\omega})$. The first absolute moment C_f quantifies the *smoothness* or *regularity* of the function f .

The first absolute moment C_f provides the basis for a *bound* on the error that results from the use of a multilayer perceptron represented by the input-output mapping function $F(\mathbf{x})$ of Eq. (4.86) to approximate $f(\mathbf{x})$. The approximation error is measured by the *integrated squared error* with respect to an arbitrary probability measure μ on the ball $B_r = \{\mathbf{x}: \|\mathbf{x}\| \leq r\}$ of radius $r > 0$. On this basis we may state the following proposition for a bound on the approximation error due to Barron (1993):

For every continuous function $f(\mathbf{x})$ with first moment C_f finite, and every $m_1 \geq 1$, there exists a linear combination of sigmoid functions $F(\mathbf{x})$ of the form defined in Eq. (4.86), such that

$$\int_{B_r} (f(\mathbf{x}) - F(\mathbf{x}))^2 \mu(d\mathbf{x}) \leq \frac{C_f'}{m_1}$$

where $C_f' = (2rC_f)^2$.

When the function $f(\mathbf{x})$ is observed at a set of values of the input vector $\mathbf{x}$ denoted by $\{\mathbf{x}_i\}_{i=1}^N$ that are restricted to lie inside the ball B_r , the result provides the following bound on the *empirical risk*:

$$R = \frac{1}{N} \sum_{i=1}^N (f(\mathbf{x}_i) - F(\mathbf{x}_i))^2 \leq \frac{C_f'}{m_1} \quad (4.89)$$

In Barron (1992), the approximation result of Eq. (4.89) is used to express the bound on the risk R resulting from the use of a multilayer perceptron with m_0 input nodes and m_1 hidden neurons as follows:

$$R \leq O\left(\frac{C_f^2}{m_1}\right) + O\left(\frac{m_0 m_1}{N} \log N\right) \quad (4.90)$$

The two terms in the bound on the risk R express the tradeoff between two conflicting requirements on the size of the hidden layer:

1. *Accuracy of best approximation.* For this requirement to be satisfied, m_1 , the size of the hidden layer, must be large in accordance with the universal approximation theorem.
2. *Accuracy of empirical fit to the approximation.* To satisfy this second requirement, we must use a small ratio m_1/N . For a fixed size of training sample, N , the

size of the hidden layer, m_1 , should be kept small, which is in conflict with the first requirement.

The bound on the risk R described in Eq. (4.90) has other interesting implications. Specifically, we see that an exponentially large sample size, large in the dimensionality m_0 of the input space, is *not* required to get an accurate estimate of the target function, provided that the first absolute moment C_f remains finite. This result makes multilayer perceptrons as universal approximators even more important in practical terms.

The error between the empirical fit and the best approximation may be viewed as an *estimation error* along the lines described in Chapter 2. Let ϵ_0 denote the mean-square value of this estimation error. Then ignoring the logarithmic factor $\log N$ in the second term of the bound in Eq. (4.90), we may infer that the size N of the training sample needed for a good generalization is about $m_0 m_1 / \epsilon_0$. This result has a mathematical structure similar to the empirical rule of Eq. (4.85), bearing in mind that $m_0 m_1$ is equal to the total number of free parameters W in the network. In other words, we may generally say that for good generalization, the number of training examples N should be larger than the ratio of the total number of free parameters in the network to the mean-square value of the estimation error.

Curse of Dimensionality

Another interesting result that emerges from the bounds described in (4.90) is that when the size of the hidden layer is optimized (i.e., the risk R is minimized with respect to N) by setting

$$m_1 \approx C_f \left(\frac{N}{m_0 \log N} \right)^{1/2}$$

then the risk R is bounded by $O(C_f \sqrt{m_0 (\log N / N)})$. A surprising aspect of this result is that in terms of the first-order behavior of the risk R , the rate of convergence expressed as a function of the training sample size N is of order $(1/N)^{1/2}$ (times a logarithmic factor). In contrast, for traditional smooth functions (e.g., polynomials and trigonometric functions) we have a different behavior. Let s denote a measure of smoothness, defined as the number of continuous derivatives of a function of interest. Then, for traditional smooth functions we find that the minimax rate of convergence of the total risk R is of order $(1/N)^{2s/(2s+m_0)}$. The dependence of this rate on the dimensionality of the input space, m_0 , is a curse of dimensionality, which severely restricts the practical application of these functions. The use of a multilayer perceptron for function approximation appears to offer an advantage over traditional smooth functions; this advantage is, however, subject to the condition that the first absolute moment C_f remains finite; this is a smoothness constraint.

The *curse of dimensionality* was introduced by Richard Bellman in his studies of adaptive control processes (Bellman, 1961). For a geometric interpretation of this notion, let $\mathbf{x}$ denote an m_0 -dimensional input vector, and $\{(\mathbf{x}_i, d_i)\}$, $i = 1, 2, \dots, N$, denote the training sample. The *sampling density* is proportional to N^{1/m_0} . Let a function $f(\mathbf{x})$ represent a surface lying in the m_0 -dimensional input space, which passes near the data points $\{(\mathbf{x}_i, d_i)\}_{i=1}^N$. Now, if the function $f(\mathbf{x})$ is arbitrarily complex and (for the most

part) completely unknown, we need *dense* sample (data) points to learn it well. Unfortunately, dense samples are hard to find in “high dimensions,” hence the curse of dimensionality. In particular, there is an *exponential* growth in complexity as a result of an increase in dimensionality, which in turn leads to the deterioration of the space-filling properties for uniformly randomly distributed points in higher-dimension spaces. The basic reason for the curse of dimensionality is (Friedman, 1995):

A function defined in high-dimensional space is likely to be much more complex than a function defined in a lower-dimensional space, and those complications are harder to discern.

The only practical way to beat the curse of dimensionality is to incorporate *prior knowledge* about the function over and above the training data, which is known to be *correct*.

In practice, it may also be argued that to have any hope of good estimation in a high-dimensional space we must provide increasing smoothness of the unknown underlying function with increasing input dimensionality (Niyogi and Girosi, 1996). This viewpoint is pursued further in Chapter 5.

Practical Considerations

The universal approximation theorem is important from a theoretical viewpoint, because it provides the *necessary mathematical tool* for the viability of feedforward networks with a single hidden layer as a class of approximate solutions. Without such a theorem, we could conceivably be searching for a solution that cannot exist. However, the theorem is not constructive, that is, it does not actually specify how to determine a multilayer perceptron with the stated approximation properties.

The universal approximation theorem assumes that the continuous function to be approximated is given and that a hidden layer of unlimited size is available for the approximation. Both of these assumptions are violated in most practical applications of multilayer perceptrons.

The problem with multilayer perceptrons using a single hidden layer is that the neurons therein tend to interact with each other globally. In complex situations this interaction makes it difficult to improve the approximation at one point without worsening it at some other point. On the other hand, with two hidden layers the approximation (curve-fitting) process becomes more manageable. In particular, we may proceed as follows (Funahashi, 1989; Chester, 1990):

1. *Local features* are extracted in the first hidden layer. Specifically, some neurons in the first hidden layer are used to partition the input space into regions, and other neurons in that layer learn the local features characterizing those regions.
2. *Global features* are extracted in the second hidden layer. Specifically, a neuron in the second hidden layer combines the outputs of neurons in the first hidden layer operating on a particular region of the input space, and thereby learns the global features for that region and outputs zero elsewhere.

This two-stage approximation process is similar in philosophy to the spline technique for curve fitting, in the sense that the effects of neurons are isolated and the approximations in different regions of the input space may be individually adjusted. A *spline* is an example of a piecewise polynomial approximation.

Sontag (1992) provides further justification for the use of two hidden layers in the context of *inverse problems*. Specifically, the following inverse problem is considered:

Given a continuous vector-valued function $f: \mathbb{R}^m \rightarrow \mathbb{R}^M$, a compact subset $\mathcal{C} \subseteq \mathbb{R}^M$ that is included in the image of f , and an $\epsilon > 0$, find a vector-valued function $\varphi: \mathbb{R}^M \rightarrow \mathbb{R}^m$ such that the following condition is satisfied:

$$\|f(u) - u\| < \epsilon \quad \text{for } u \in \mathcal{C}$$

This problem arises in *inverse kinematics* (dynamics), where the observed state $x(n)$ of a system is a function of current actions $u(n)$ and the previous state $x(n-1)$ of the system, as shown by

$$x(n) = f(x(n-1), u(n))$$

It is assumed that f is *invertible*, so that we may solve for $u(n)$ as a function of $x(n)$ for any $x(n-1)$. The function f represents the direct kinematics, whereas the function φ represents the inverse kinematics. In practical terms, the motivation is to find a function φ that is computable by a multilayer perceptron. In general, discontinuous functions φ are needed to solve the inverse kinematics problem. It is interesting that even if the use of neuronal models with discontinuous activation functions is permitted, one hidden layer is *not* enough to guarantee the solution of all such inverse problems, whereas multilayer perceptrons with two hidden layers are sufficient for every possible f , $\mathcal{C}$, and ϵ (Sontag, 1992).

4.14 CROSS-VALIDATION

The essence of back-propagation 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. The hope is 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 this data set. More specifically, 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.

In this context, a standard tool in statistics known as *cross-validation* provides an appealing guiding principle⁹ (Stone, 1974, 1978). First the available data set is randomly partitioned into a training set and a test set. The training set is further partitioned into two disjoint subsets:

- *Estimation subset*, used to select the model.
- *Validation subset*, used to test or validate the model.

The motivation here 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 set to assess the performance of various candidate models, and thereby choose the “best” one. There is, however, a distinct possibility that the model with the best-performing parameter values so selected may end up overfitting the validation subset. To guard against this possibility, the generalization performance of the selected model is measured on the test set, which is different from the validation subset.

The use of cross-validation is appealing particularly when we have to design a large neural network with good generalization as the goal. For example, we may use cross-validation to determine the multilayer perceptron with the best number of hidden neurons, and when it is best to stop training, as described in the next two subsections.

Model Selection

The idea of selecting a model in accordance with cross-validation follows a philosophy similar to that of structural risk minimization described in Chapter 2. Consider then a nested structure of Boolean function classes denoted by

$$\begin{aligned}\mathcal{F}_1 &\subset \mathcal{F}_2 \subset \dots \subset \mathcal{F}_n \\ \mathcal{F}_k &= \{F_k\} \\ &= \{F(\mathbf{x}, \mathbf{w}); \mathbf{w} \in \mathcal{W}_k\}, \quad k = 1, 2, \dots, n\end{aligned}\quad (4.91)$$

In words, the k th function class $\mathcal{F}_k$ encompasses a family of multilayer perceptrons with similar architecture and weight vectors $\mathbf{w}$ drawn from a multidimensional weight space $\mathcal{W}_k$. A member of this class, characterized by the function or hypothesis $F_k = F(\mathbf{x}, \mathbf{w})$, $\mathbf{w} \in \mathcal{W}_k$, maps the input vector $\mathbf{x}$ into $\{0, 1\}$, where $\mathbf{x}$ is drawn from an input space $\mathcal{X}$ with some unknown probability P . Each multilayer perceptron in the structure described is trained with the back-propagation algorithm, which takes care of training the parameters of the multilayer perceptron. The model selection problem is essentially that of choosing the multilayer perceptron with the best value of W , the number of free parameters (i.e., synaptic weights and biases). More precisely, given that the scalar desired response for an input vector $\mathbf{x}$ is $d = \{0, 1\}$, we define the generalization error as

$$\epsilon_g(F) = P(F(\mathbf{x}) \neq d) \quad \text{for } \mathbf{x} \in \mathcal{X}$$

We are given a training set of labeled examples

$$\mathcal{T} = \{(\mathbf{x}_i, d_i)\}_{i=1}^N$$

The objective is to select the particular hypothesis $F(\mathbf{x}, \mathbf{w})$, which minimizes the generalization error $\epsilon_g(F)$ that results when it is given inputs from the test set.

In what follows we assume that the structure described by Eq. (4.91) has the property that for any sample size N we can always find a multilayer perceptron with a large enough number of free parameters $W_{\max}(N)$, such that the training data set $\mathcal{T}$ can be fitted adequately. This is merely restating the universal approximation theorem of Section 4.13. We refer to $W_{\max}(N)$ as the *fitting number*. The significance of $W_{\max}(N)$ is that a reasonable model selection procedure would choose a hypothesis $F(\mathbf{x}, \mathbf{w})$ that requires $W \leq W_{\max}(N)$; otherwise the network complexity would be increased.

Let a parameter r , lying in the range between 0 and 1, determine the split of the training data set $\mathcal{T}$ between the estimation subset and validation subset. With $\mathcal{T}$ consisting of N examples, $(1-r)N$ examples are allotted to the estimation subset and the remaining rN examples are allotted to the validation subset. The estimation subset, denoted by $\mathcal{T}'$, is used to train a nested sequence of multilayer perceptrons, resulting in the hypotheses $\mathcal{F}_1, \mathcal{F}_2, \dots, \mathcal{F}_n$ of increasing complexity. With $\mathcal{T}'$ made up of $(1-r)N$ examples, we consider values of W smaller than or equal to the corresponding fitting number $W_{\max}((1-r)N)$.

The use of cross-validation results in the choice

$$\mathcal{F}_{cv} = \min_{k=1,2,\dots,v} \{e_r''(\mathcal{F}_k)\} \quad (4.92)$$

where v corresponds to $W_v \leq W_{\max}((1-r)N)$, and $e_r''(\mathcal{F}_k)$ is the classification error produced by hypothesis $\mathcal{F}_k$ when it is tested on the validation subset $\mathcal{T}''$, consisting of rN examples.

The key issue is how to specify the parameter r that determines the split of the training set $\mathcal{T}$ between the estimation subset $\mathcal{T}'$ and validation subset $\mathcal{T}''$. In a study described in Kearns (1996) involving an analytic treatment of this issue using the VC dimension and supported with detailed computer simulations, several qualitative properties of the optimum r are identified:

- When the complexity of the target function, defining the desired response d in terms of the input vector $\mathbf{x}$, is small compared to the sample size N , the performance of cross-validation is relatively insensitive to the choice of r .
- As the target function becomes more complex relative to the sample size N , the choice of optimum r has a more pronounced effect on cross-validation performance, and its own value decreases.
- A single fixed value of r works *nearly* optimally for a wide range of target function complexity.

On the basis of the results reported in Kearns (1996), a fixed value of r equal to 0.2 appears to be a sensible choice, which means that 80 percent of the training set $\mathcal{T}$ is assigned to the estimation subset and the remaining 20 percent is assigned to the validation subset.

Earlier we spoke of a nested sequence of multilayer perceptrons of increasing complexity. For prescribed input and output layers, such a sequence can be created, for example, by having $v = p + q$ fully-connected multilayer perceptrons structured as follows:

- p multilayer perceptrons with a single hidden layer of increasing size $h'_1 < h'_2 < \dots < h'_p$.
- q multilayer perceptrons with two hidden layers; the first hidden layer is of size h''_p and the second hidden layer is of increasing size $h''_1 < h''_2 < \dots < h''_q$.

As we go from one multilayer perceptron to the next, there is a corresponding increase in the number of free parameters W . The model selection procedure based on cross-validation as described provides us with a principled approach to determine the number of hidden neurons in a multilayer perceptron. Although the procedure was described in the context of binary classification, it applies equally well to other applications of the multilayer perceptron.

Early Stopping Method of Training

Ordinarily, a multilayer perceptron trained with the back-propagation algorithm learns in stages, moving from the realization of fairly simple to more complex mapping functions as the training session progresses. This is exemplified by the fact that in a typical situation the mean-square error decreases with an increasing number of epochs during training: it starts off at a large value, decreases rapidly, and then continues to

decrease slowly as the network makes its way to a local minimum on the error surface. With good generalization as the goal, it is very difficult to figure out when it is best to stop training if we were to look at the learning curve for training all by itself. In particular, in light of what was said in Section 4.12 on generalization, it is possible for the network to end up overfitting the training data if the training session is not stopped at the right point.

We may identify the onset of overfitting through the use of cross-validation, for which the training data are split into an estimation subset and a validation subset. The estimation subset of examples is used to train the network in the usual way, except for a minor modification: the training session is stopped periodically (i.e., every so many epochs), and the network is tested on the validation subset after each period of training. More specifically, the periodic estimation-followed-by-validation process proceeds as follows:

- After a period of estimation (training), the synaptic weights and bias levels of the multilayer perceptron are all fixed, and the network is operated in its forward mode. The validation error is thus measured for each example in the validation subset.
- When the validation phase is completed, the estimation (training) is resumed for another period, and the process is repeated.

This procedure is referred to as the *early stopping method of training*.¹⁰

Figure 4.20 shows conceptualized forms of two learning curves, one pertaining to measurements on the estimation subset and the other pertaining to the validation subset. Typically, the model does not do as well on the validation subset as it does on the estimation subset, on which its design was based. The *estimation learning curve* decreases monotonically for an increasing number of epochs in the usual manner. In contrast, the *validation learning curve* decreases monotonically to a minimum, it then starts to increase as the training continues. When we look at the estimation learning curve it may appear that we could do better by going beyond the minimum point on

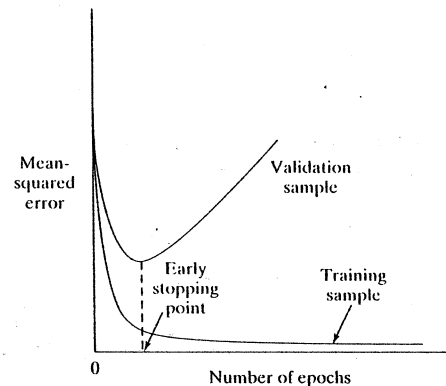


FIGURE 4.20 Illustration of the early-stopping rule based on cross-validation.

the validation learning curve. In reality, however, what the network is learning beyond this point is essentially noise contained in the training data. This heuristic suggests that the minimum point on the validation learning curve be used as a sensible criterion for stopping the training session.

What if the training data are noise free? How then would we justify early stopping for a deterministic scenario? Part of the answer in this case is that if both the estimation and validation errors cannot be simultaneously driven to zero, it implies that the network does not have the capacity to model the function exactly. The best that we can do in that situation is to try to minimize, for example, the integrated-squared error that is (roughly) equivalent to minimizing the usual global mean-square error with a uniform input density.

A statistical theory of the overfitting phenomenon presented in Amari et al. (1996) provides a word of caution for using the early stopping method of training. The theory is based on batch learning, and supported with detailed computer simulations involving a multilayer perceptron classifier with a single hidden layer. Two modes of behavior are identified depending on the size of the training set:

Nonasymptotic mode, for which $N < W$, where N is the size of the training set and W is the number of free parameters in the network. For this mode of behavior, the early stopping method of training does improve the generalization performance of the network over exhaustive training (i.e., when the complete set of examples is used for training and the training session is not stopped). This result suggests that overfitting may occur when $N < 30W$, and there is practical merit in the use of cross-validation to stop training. The optimum value of parameter r that determines the split of the training data between estimation and validation subsets is defined by

$$r_{\text{opt}} = 1 - \frac{\sqrt{2W - 1} - 1}{2(W - 1)}$$

For large W , this formula approximates to

$$r_{\text{opt}} \approx 1 - \frac{1}{\sqrt{2W}}, \quad \text{large } W \quad (4.93)$$

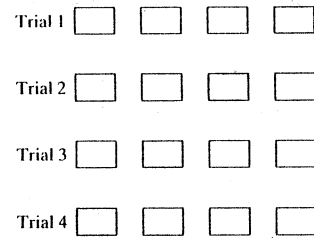
For example, for $W = 100$, $r_{\text{opt}} = 0.07$, which means that 93 percent of the training data are allotted to the estimation subset and the remaining 7 percent are allotted to the validation subset.

Asymptotic mode, for which $N > 30W$. For this mode of behavior, the improvement in generalization performance produced by the use of the early stopping method of training over exhaustive training is small. In other words, exhaustive learning is satisfactory when the size of the training sample is large compared to the number of network parameters.

Variants of Cross-Validation

The approach to cross-validation described is referred to as the *hold out method*. There are other variants of cross-validation that find their own uses in practice, particularly when there is a scarcity of labeled examples. In such a situation we may use *multifold*

FIGURE 4.21 Illustration of the hold-out method of cross-validation. For a given trial, the shaded subset of data is used to validate the model trained on the remaining data.



cross-validation by dividing the available set of N examples into K subsets, $K > 1$; this assumes that K is divisible into N . The model is trained on all the subsets except for one, and the validation error is measured by testing it on the subset left out. This procedure is repeated for a total of K trials, each time using a different subset for validation, as illustrated in Fig. 4.21 for $K = 4$. The performance of the model is assessed by averaging the squared error under validation over all the trials of the experiment. There is a disadvantage to multifold cross-validation: it may require an excessive amount of computation since the model has to be trained K times, where $1 < K \leq N$.

When the available number of labeled examples, N , is severely limited, we may use the extreme form of multifold cross-validation known as the *leave-one-out method*. In this case, $N - 1$ examples are used to train the model, and the model is validated by testing it on the example left out. The experiment is repeated for a total of N times, each time leaving out a different example for validation. The squared error under validation is then averaged over the N trials of the experiment.

15 NETWORK PRUNING TECHNIQUES

To solve real-world problems with neural networks usually requires the use of highly structured networks of a rather large size. A practical issue that arises in this context is that of minimizing the size of the network while maintaining good performance. A neural network with minimum size is less likely to learn the idiosyncrasies or noise in the training data, and may thus generalize better to new data. We may achieve this design objective in one of two ways:

- *Network growing*, in which case we start with a small multilayer perceptron, small for accomplishing the task at hand, and then add a new neuron or a new layer of hidden neurons only when we are unable to meet the design specification.¹¹
- *Network pruning*, in which case we start with a large multilayer perceptron with an adequate performance for the problem at hand, and then prune it by weakening or eliminating certain synaptic weights in a selective and orderly fashion.

In this section we focus on network pruning. In particular, we describe two approaches, one based on a form of “regularization,” and the other based on the “deletion” of certain synaptic connections from the network.

Complexity-Regularization

In designing a multilayer perceptron by whatever method, we are in effect building a nonlinear *model* of the physical phenomenon responsible for the generation of the input–output examples used to train the network. Insofar as the network design is statistical in nature, we need an appropriate tradeoff between reliability of the training data and goodness of the model (i.e., a method for solving the bias-variance dilemma). In the context of back-propagation learning, or any other supervised learning procedure for that matter, we may realize this tradeoff by minimizing the total risk expressed as:

$$R(\mathbf{w}) = \mathcal{E}_s(\mathbf{W}) + \lambda \mathcal{E}_c(\mathbf{w}) \quad (4.94)$$

The first term, $\mathcal{E}_s(\mathbf{w})$, is the standard *performance measure*, which depends on both the network (model) and the input data. In back-propagation learning it is typically defined as a mean-square error whose evaluation extends over the output neurons of the network and which is carried out for all the training examples on an epoch-by-epoch basis. The second term, $\mathcal{E}_c(\mathbf{w})$, is the *complexity penalty*, which depends on the network (model) alone; its inclusion imposes on the solution prior knowledge that we may have on the models being considered. In fact, the form of the total risk defined in Eq. (4.94) is simply a statement of Tikhonov’s *regularization theory*; this subject is detailed in Chapter 5. For the present discussion, it suffices to think of λ as a *regularization parameter*, which represents the relative importance of the complexity-penalty term with respect to the performance-measure term. When λ is zero, the back-propagation learning process is unconstrained, with the network being completely determined from the training examples. When λ is made infinitely large, on the other hand, the implication is that the constraint imposed by the complexity penalty is by itself sufficient to specify the network, which is another way of saying that the training examples are unreliable. In practical applications of the weight-decay procedure, the regularization parameter λ is assigned a value somewhere between these two limiting cases. The viewpoint described here for the use of complexity regularization for improved generalization is entirely consistent with the structural risk minimization procedure discussed in Chapter 2.

In a general setting, one choice of complexity-penalty term $\mathcal{E}_c(\mathbf{w})$ is the k th order smoothing integral

$$\mathcal{E}_c(\mathbf{w}, k) = \frac{1}{2} \int \left\| \frac{\partial^k}{\partial \mathbf{x}^k} F(\mathbf{x}, \mathbf{w}) \right\|^2 \mu(\mathbf{x}) d\mathbf{x} \quad (4.95)$$

where $F(\mathbf{x}, \mathbf{w})$ is the input–output mapping performed by the model, and $\mu(\mathbf{x})$ is some weighting function that determines the region of the input space over which the function $F(\mathbf{x}, \mathbf{w})$ is required to be smooth. The motivation is to make the k th derivative of $F(\mathbf{x}, \mathbf{w})$ with respect to the input vector $\mathbf{x}$ small. The larger we choose k , the smoother (i.e., less complex) the function $F(\mathbf{x}, \mathbf{w})$ will become.

In the sequel, we describe three different complexity regularizations (of increasing sophistication) for multilayer perceptrons.

Weight Decay. In the *weight-decay procedure* (Hinton, 1989), the complexity penalty term is defined as the squared norm of the weight vector $\mathbf{w}$ (i.e., all the free parameters) in the network, as shown by

$$\begin{aligned}\mathcal{E}_c(\mathbf{w}) &= \|\mathbf{w}\|^2 \\ &= \sum_{i \in \mathcal{E}_{\text{total}}} w_i^2\end{aligned}\quad (4.96)$$

where the set $\mathcal{E}_{\text{total}}$ refers to all the synaptic weights in the network. This procedure operates by forcing some of the synaptic weights in the network to take values close to zero, while permitting other weights to retain their relatively large values. Accordingly, the weights of the network are grouped roughly into two categories: those that have a large influence on the network (model), and those that have little or no influence on it. The weights in the latter category are referred to as *excess weights*. In the absence of complexity regularization, these weights result in poor generalization by virtue of their high likelihood of taking on completely arbitrary values or causing the network to overfit the data in order to produce a slight reduction in the training error (Hush and Horne, 1993). The use of complexity regularization encourages the excess weights to assume values close to zero, and thereby improve generalization.

In the weight-decay procedure, all the weights in the multilayer perceptron are treated equally. That is, the prior distribution in weight space is assumed to be centered at the origin. Strictly speaking, weight decay is not the correct form of complexity regularization for a multilayer perceptron since it does not fit into the rationale described in Eq. (4.95). Nevertheless, it is simple and appears to work well in some applications.

Weight Elimination. In this second complexity-regularization procedure, the complexity penalty is defined by (Weigend et al., 1991)

$$\mathcal{E}_c(\mathbf{w}) = \sum_{i \in \mathcal{E}_{\text{total}}} \frac{(w_i/w_0)^2}{1 + (w_i/w_0)^2} \quad (4.97)$$

where w_0 is a preassigned parameter, and w_i refers to the weight of some synapse i in the network. The set $\mathcal{E}_{\text{total}}$ refers to all the synaptic connections in the network. An individual penalty term varies with w_i/w_0 in a symmetric fashion, as shown in Fig. 4.22. When $|w_i| \ll w_0$, the complexity penalty (cost) for that weight approaches zero. The implication of this condition is that insofar as learning from examples is concerned, the i th synaptic weight is unreliable and should therefore be eliminated from the network. On the other hand, when $|w_i| \gg w_0$, the complexity penalty (cost) for that weight approaches the maximum value of unity, which means that w_i is important to the back-propagation learning process. We thus see that the complexity penalty term of Eq. (4.97) does serve the desired purpose of identifying the synaptic weights of the network that are of significant influence. Note also that the weight-elimination procedure includes the weight-decay procedure as a special case; specifically, for large w_0 , Eq. (4.97) reduces to the form shown in Eq. (4.96) except for a scaling factor.

Strictly speaking, the weight-elimination procedure is also not the correct form of complexity regularization for multilayer perceptrons because it does not fit the

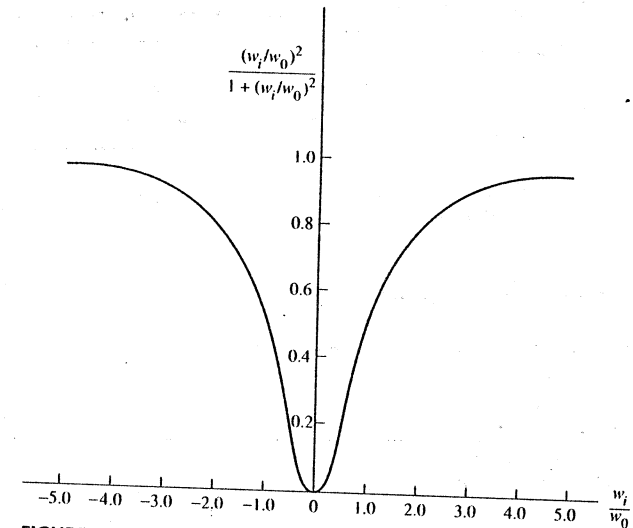


FIGURE 4.22 The complexity penalty term $(w_i/w_0)^2/[1 + (w_i/w_0)^2]$ plotted versus w_i/w_0 .

description specified in Eq. (4.95). Nevertheless, with the proper choice of parameter w_0 , it permits some weights in the network to assume values that are larger than with weight decay (Hush, 1997).

Approximate Smoother. In Moody and Rögnvaldsson (1997), the following complexity-penalty term is proposed for a multilayer perceptron with a single hidden layer and a single neuron in the output layer:

$$\mathcal{E}_c(\mathbf{w}) = \sum_{j=1}^M w_{oj}^2 \|\mathbf{w}_j\|^p \quad (4.98)$$

where the w_{oj} are the weights in the output layer, and $\mathbf{w}_j$ is the weight vector for the j th neuron in the hidden layer; the power p is defined by

$$p = \begin{cases} 2k - 1 & \text{for a global smoother} \\ 2k & \text{for a local smoother} \end{cases} \quad (4.99)$$

where k is the order of differentiation of $F(\mathbf{x}, \mathbf{w})$ with respect to $\mathbf{x}$.

The approximate smoother appears to be more accurate than weight decay or weight elimination for the complexity regularization of a multilayer perceptron. Unlike those earlier methods, it does two things:

1. It distinguishes between the roles of synaptic weights in the hidden layer and those in the output layer.
2. It captures the interactions between these two sets of weights.

However, it has a much more complicated form than weight decay or weight elimination, and is therefore more demanding in computational complexity.

Hessian-based Network Pruning

The basic idea of this second approach to network pruning is to use information on second-order derivatives of the error surface in order to make a trade-off between network complexity and training-error performance. In particular, a local model of the error surface is constructed for analytically predicting the effect of perturbations in synaptic weights. The starting point in the construction of such a model is the local approximation of the cost function $\mathcal{E}_{av}$ using a *Taylor series* about the operating point, described as follows:

$$\mathcal{E}_{av}(\mathbf{w} + \Delta\mathbf{w}) = \mathcal{E}_{av}(\mathbf{w}) + \mathbf{g}^T(\mathbf{w})\Delta\mathbf{w} + \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} + O(\|\Delta\mathbf{w}\|^3) \quad (4.100)$$

where $\Delta\mathbf{w}$ is a perturbation applied to the operating point $\mathbf{w}$, and $\mathbf{g}(\mathbf{w})$ is the gradient vector evaluated at $\mathbf{w}$. The Hessian is also evaluated at the point $\mathbf{w}$, and therefore, to be correct we should denote it by $\mathbf{H}(\mathbf{w})$. We have not done so in Eq. (4.100) merely to simplify the notation.

The requirement is to identify a set of parameters whose deletion from the multilayer perceptron will cause the least increase in the value of the cost function $\mathcal{E}_{av}$. To solve this problem in practical terms, we make the following approximations:

1. *Extremal Approximation.* We assume that parameters are deleted from the network only after the training process has converged (i.e., the network is fully trained). The implication of this assumption is that the parameters have a set of values corresponding to a local minimum or global minimum of the error surface. In such a case, the gradient vector $\mathbf{g}$ may be set equal to zero and the term $\mathbf{g}^T\Delta\mathbf{w}$ on the right-hand side of Eq. (4.100) may therefore be ignored. Otherwise the saliency measures (defined later) will be invalid for the problem at hand.

2. *Quadratic Approximation.* We assume that the error surface around a local minimum or global minimum is nearly "quadratic." Hence the higher-order terms in Eq. (4.100) may also be neglected.

Under these two assumptions, Eq. (4.100) is approximated simply as

$$\begin{aligned} \Delta\mathcal{E}_{av} &= \mathcal{E}(\mathbf{w} + \Delta\mathbf{w}) - \mathcal{E}(\mathbf{w}) \\ &\approx \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} \end{aligned} \quad (4.101)$$

The *optimal brain damage* (OBD) procedure (LeCun et al., 1990b) simplifies the computations by making a further assumption: The Hessian matrix $\mathbf{H}$ is a diagonal matrix. However, no such assumption is made in the *optimal brain surgeon* (OBS) procedure (Hassibi et al., 1992); accordingly, it contains the OBD procedure as a special case. From here on, we follow the OBS strategy.

The goal of OBS is to set one of the synaptic weights to zero to minimize the incremental increase in $\mathcal{E}_{av}$ given in Eq. (4.101). Let $w_i(n)$ denote this particular synaptic weight. The elimination of this weight is equivalent to the condition

$$\Delta w_i + w_i = 0$$

or

$$\mathbf{1}_i^T \Delta\mathbf{w} + w_i = 0 \quad (4.102)$$

where $\mathbf{1}_i$ is the *unit vector* whose elements are all zero, except for the i th element, which is equal to unity. We may now restate the goal of OBS as (Hassibi et al., 1992):

Minimize the quadratic form $\frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w}$ with respect to the incremental change in the weight vector, $\Delta\mathbf{w}$, subject to the constraint that $\mathbf{1}_i^T\Delta\mathbf{w} + w_i$ is zero, and then minimize the result with respect to the index i .

There are two levels of minimization going on here. One minimization is over the synaptic weight vectors that remain after the i th weight vector is set equal to zero. The second minimization is over which particular vector is pruned.

To solve this constrained optimization problem, we first construct the *Lagrangian*

$$S = \frac{1}{2}\Delta\mathbf{w}^T\mathbf{H}\Delta\mathbf{w} - \lambda(\mathbf{1}_i^T\Delta\mathbf{w} + w_i) \quad (4.103)$$

where λ is the *Lagrange multiplier*. Then, taking the derivative of the Lagrangian S with respect to $\Delta\mathbf{w}$, applying the constraint of Eq. (4.102), and using matrix inversion, we find that the optimum change in the weight vector $\mathbf{w}$ is

$$\Delta\mathbf{w} = -\frac{w_i}{[\mathbf{H}^{-1}]_{i,i}}\mathbf{H}^{-1}\mathbf{1}_i \quad (4.104)$$

and the corresponding optimum value of the Lagrangian S for element w_i is

$$S_i = \frac{w_i^2}{2[\mathbf{H}^{-1}]_{i,i}} \quad (4.105)$$

where $\mathbf{H}^{-1}$ is the inverse of the Hessian matrix $\mathbf{H}$, and $[\mathbf{H}^{-1}]_{i,i}$ is the i -th element of this inverse matrix. The Lagrangian S_i optimized with respect to $\Delta\mathbf{w}$, subject to the constraint that the i th synaptic weight w_i be eliminated, is called the *saliency* of w_i . In effect, the saliency S_i represents the increase in the mean-square error (performance measure) that results from the deletion of w_i . Note that the saliency S_i is proportional to w_i^2 . Thus small weights have a small effect on the mean-square error. However, from Eq. (4.105) we see that the saliency S_i is also inversely proportional to the diagonal elements of the inverse Hessian. Thus if $[\mathbf{H}^{-1}]_{i,i}$ is small, then even small weights may have a substantial effect on the mean-square error.

In the OBS procedure, the weight corresponding to the smallest saliency is the one selected for deletion. Moreover, the corresponding optimal changes in the remainder of the weights are given in Eq. (4.104), which show that they should be updated along the direction of the i th column of the inverse of the Hessian.

In their paper, Hassibi et al. report that on some benchmark problems the OBS procedure resulted in smaller networks than those obtained using the weight-decay procedure. It is also reported that as a result of applying the OBS procedure to the NETalk multilayer perceptron involving a single hidden layer and 18,000 weights, the network was pruned to a mere 1560 weights, a dramatic reduction in the size of the network. NETalk, due to Sejnowski and Rosenberg (1987), is described in Chapter 13.

Computing the inverse Hessian matrix. The inverse Hessian matrix $\mathbf{H}^{-1}$ is fundamental to the formulation of the OBS procedure. When the number of free parameters, W , in the network is large, the problem of computing $\mathbf{H}^{-1}$ may be intractable. In what follows we describe a manageable procedure for computing $\mathbf{H}^{-1}$, assuming that the multilayer perceptron is fully trained to a local minimum on the error surface (Hassibi et al., 1992).

To simplify the presentation, suppose that the multilayer perceptron has a single output neuron. Then, for a given training set we may express the cost function as

$$\mathcal{E}_{av}(\mathbf{w}) = \frac{1}{2N} \sum_{n=1}^N (d(n) - o(n))^2$$

where $o(n)$ is the actual output of the network on the presentation of the n th example, $d(n)$ is the corresponding desired response, and N is the total number of examples in the training set. The output $o(n)$ may itself be expressed as

$$o(n) = F(\mathbf{w}, \mathbf{x})$$

where F is the input-output mapping function realized by the multilayer perceptron, $\mathbf{x}$ is the input vector, and $\mathbf{w}$ is the synaptic weight vector of the network. The first derivative of $\mathcal{E}_{av}$ with respect to $\mathbf{w}$ is therefore

$$\frac{\partial \mathcal{E}_{av}}{\partial \mathbf{w}} = -\frac{1}{N} \sum_{n=1}^N \frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} (d(n) - o(n)) \quad (4.106)$$

and the second derivative of $\mathcal{E}_{av}$ with respect to $\mathbf{w}$ or the Hessian matrix is

$$\begin{aligned} \mathbf{H}(N) &= \frac{\partial^2 \mathcal{E}_{av}}{\partial \mathbf{w}^2} \\ &= \frac{1}{N} \sum_{n=1}^N \left\{ \left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \right) \left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \right)^T \right. \\ &\quad \left. - \frac{\partial^2 F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}^2} (d(n) - o(n)) \right\} \end{aligned} \quad (4.107)$$

where we have emphasized the dependence of the Hessian matrix on the size of the training sample, N .

Under the assumption that the network is fully trained, that is, the cost function $\mathcal{E}_{av}$ has been adjusted to a local minimum on the error surface, it is reasonable to say that $o(n)$ is close to $d(n)$. Under this condition we may ignore the second term and approximate Eq. (4.107) as

$$\mathbf{H}(N) \approx \frac{1}{N} \sum_{n=1}^N \left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \right) \left(\frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \right)^T \quad (4.108)$$

To simplify the notation, define the W -by-1 vector

$$\xi(n) = \frac{1}{\sqrt{N}} \frac{\partial F(\mathbf{w}, \mathbf{x}(n))}{\partial \mathbf{w}} \quad (4.109)$$

which may be computed using the procedure described in Section 4.10. We may then rewrite Eq. (4.108) in the form of a recursion as:

$$\begin{aligned} \mathbf{H}(n) &= \sum_{k=1}^n \xi(k) \xi^T(k) \\ &= \mathbf{H}(n-1) + \xi(n) \xi^T(n), \quad n = 1, 2, \dots, N \end{aligned} \quad (4.110)$$

This recursion is in the right form for application of the so-called *matrix inversion lemma*, also known as *Woodbury's equality*.

Let $\mathbf{A}$ and $\mathbf{B}$ denote two positive definite matrices related by

$$\mathbf{A} = \mathbf{B}^{-1} + \mathbf{C} \mathbf{D} \mathbf{C}^T$$

where $\mathbf{C}$ and $\mathbf{D}$ are two other matrices. According to the matrix inversion lemma, the inverse of matrix $\mathbf{A}$ is defined by

$$\mathbf{A}^{-1} = \mathbf{B} - \mathbf{B} \mathbf{C} (\mathbf{D} + \mathbf{C}^T \mathbf{B} \mathbf{C})^{-1} \mathbf{C}^T \mathbf{B}$$

For the problem described in Eq. (4.110) we have

$$\mathbf{A} = \mathbf{H}(n)$$

$$\mathbf{B}^{-1} = \mathbf{H}(n-1)$$

$$\mathbf{C} = \xi(n)$$

$$\mathbf{D} = 1$$

Application of the matrix inversion lemma therefore yields the desired formula for recursive computation of the inverse Hessian:

$$\mathbf{H}^{-1}(n) = \mathbf{H}^{-1}(n-1) - \frac{\mathbf{H}^{-1}(n-1) \xi(n) \xi^T(n) \mathbf{H}^{-1}(n-1)}{1 + \xi^T(n) \mathbf{H}^{-1}(n-1) \xi(n)} \quad (4.111)$$

Note that the denominator in Eq. (4.111) is a scalar; it is therefore straightforward to calculate its reciprocal. Thus, given the past value of the inverse Hessian, $\mathbf{H}^{-1}(n-1)$, we may compute its updated value $\mathbf{H}^{-1}(n)$ on the presentation of the n th example represented by the vector $\xi(n)$. This recursive computation is continued until the entire set of N examples has been accounted for. To initialize the algorithm we need to make $\mathbf{H}^{-1}(0)$ large, since it is being constantly reduced according to Eq. (4.111). This requirement is satisfied by setting

$$\mathbf{H}^{-1}(0) = \delta^{-1} \mathbf{I} \quad (4.112)$$

where δ is a small positive number and $\mathbf{I}$ is the identity matrix. This form of initialization assures that $\mathbf{H}^{-1}(n)$ is always positive definite. The effect of δ becomes progressively smaller as more and more examples are presented to the network.

A summary of the brain surgeon algorithm is presented in Table 4.6 (Hassibi and Stork, 1992).