6.1 Introduction

In this chapter we study an important class of neural networks, namely, multilayer feedforward 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 4.

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 5 for the special case of a single linear neuron model.

Basically, the error back-propagation process 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 network are all fixed. During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with the 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 so as to make the actual response of the network move closer to the desired response. 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 this algorithm is called back-propagation learning.

A multilayer perceptron has three distinctive characteristics:

1. The model of each neuron in the network includes a nonlinearity at the output end. 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 common used form of nonlinearity that satisfies this requirement is a sigmoidal nonlinearity defined by the logistic function:

   \[ y_j = \frac{1}{1 + \exp(-v_j)} \]
where \( v_j \) is the net internal activity level 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 (Pineda, 1988b).

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 degree 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.

Indeed, 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, 1987).

Research interest in multilayer feedforward networks dates back to the pioneering work of Rosenblatt (1962) on perceptrons and that of Widrow and his students on Madalines (Widrow, 1962). Madalines were constructed with many inputs, many Adaline elements in the first layer, and with various logic devices such as AND, OR, and majority-votetaker elements in the second layer; the Adaline was described in Section 5.8. The Madalines of the 1960s had adaptive first layers and fixed threshold functions in the second (output) layers (Widrow and Lehr, 1990). However, the tool that was missing in those early days of multilayer feedforward networks was what we now call back-propagation learning.

The usage of the term "back-propagation" appears to have evolved after 1985. However, the basic idea of back-propagation was first described by Werbos in his Ph.D. thesis (Werbos, 1974), in the context of general networks with neural networks representing a special case. Subsequently, it was rediscovered by Rumelhart, Hinton, and Williams (1986b), and popularized through the publication of the seminal book entitled Parallel Distributed Processing (Rumelhart and McClelland, 1986). A similar generalization of the algorithm was derived independently by Parker in 1985, and interestingly enough, a roughly similar learning algorithm was also studied by LeCun (1985).

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 it cannot be claimed that the back-propagation algorithm can provide a solution for all solvable problems, it is fair to say that it has put to rest the pessimism about learning in multilayer machines that may have been inferred from the book by Minsky and Papert (1969).

**Organization of the Chapter**

In this rather long chapter we consider both the theory and applications of multilayer perceptrons. We begin with some preliminaries in Section 6.2 to pave the way for the
derivation of the back-propagation algorithm. In Section 6.3 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 6.4. Then, in Section 6.5, we address the issue of initialization, which plays a key role in successful applications of the back-propagation algorithm. In Section 6.6 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 6.7 we present some practical hints for making the back-propagation algorithm work better.

In Section 6.8 we address the development of a decision rule for the use of a back-propagation network to solve the statistical pattern-recognition problem. Then, in Section 6.9, we use a computer experiment to illustrate the application of back-propagation learning to distinguish between two classes of overlapping two-dimensional Gaussian distributions.

In Sections 6.10 through 6.14 we consider some basic issues relating to back-propagation learning. In Section 6.10 we discuss the issue of generalization, which is the very essence of back-propagation learning. This is naturally followed by Section 6.11 with a brief discussion of cross-validation, a standard tool in statistics, and how it can be applied to the design of neural networks. In Section 6.12 we consider the approximate realization of any continuous input–output mapping by a multilayer perceptron. In Section 6.13 we discuss the fundamental role of back-propagation in computing partial derivatives. In Section 6.14 we pause to summarize the important advantages and limitations of back-propagation learning.

In Section 6.15 we discuss some heuristics that provide guidelines for how to accelerate the rate of convergence of back-propagation learning. These heuristics are used to formulate a form of learning called fuzzy back-propagation, which is presented in Section 6.16. In Section 6.17 we describe procedures to orderly "prune" a multilayer perceptron and maintain (and frequently, improve) overall performance; network pruning is desirable when the use of VLSI technology is being considered for the hardware implementation of multilayer perceptrons.

In Sections 6.18 and 6.19 we reexamine the supervised learning of a multilayer perceptron as a problem in system identification or function optimization, respectively. In Section 6.20 we consider the use of a multilayer perceptron for learning a probability distribution. We conclude the chapter with some general discussion in Section 6.21 and important applications of back-propagation learning in Section 6.22.

### 6.2 Some Preliminaries

Figure 6.1 shows the architectural graph of a multilayer perceptron with two hidden layers. To set the stage in its general form, the network shown here is _fully connected_, which 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 6.2 depicts a portion of the multilayer perceptron. In this network, two kinds of signals are identified (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 calculated as a function of the inputs and associated weights applied to that neuron.

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 layer 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 signals and synaptic weights associated with that neuron.

2. The computation of an instantaneous 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.

- The iteration \(n\) refers to the \(n\)th training pattern (example) presented to the network.

- The symbol \(E(n)\) refers to the instantaneous sum of error squares at iteration \(n\). The average of \(E(n)\) over all values of \(n\) (i.e., the entire training set) yields the average squared error \(E_{av}\).

- 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 net internal activity level of neuron \(j\) at iteration \(n\) is denoted by \(v_j(n)\); it constitutes the signal applied to the nonlinearity associated with neuron \(j\).

- The activation function describing the input–output functional relationship of the nonlinearity associated with neuron \(j\) is denoted by \(\phi(\cdot)\).

- The threshold applied to neuron \(j\) is denoted by \(\theta_j\); its effect is represented by a synapse of weight \(w_0 = \theta_j\) 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 \(\eta\).

### 6.3 Derivation of the Back-Propagation Algorithm

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

\[
e_j(n) = d_j(n) - y_j(n), \quad \text{neuron } j \text{ is an output node} \quad (6.1)
\]
We define the instantaneous value of the squared error for neuron \( j \) as \( \frac{1}{2} e_j^2(n) \). Correspondingly, the instantaneous value \( \mathcal{E}(n) \) of the sum of squared errors 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. The instantaneous sum of squared errors of the network is thus written as

\[
\mathcal{E}(n) = \frac{1}{2} \sum_{j \in C} e_j^2(n)
\]  

(6.2)

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

\[
\mathcal{E}_w = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}(n)
\]

(6.3)

The instantaneous sum of error squares \( \mathcal{E}(n) \), and therefore the average squared error \( \mathcal{E}_w \), is a function of all the free parameters (i.e., synaptic weights and thresholds) of the network. For a given training set, \( \mathcal{E}_w \) represents the cost function as the measure of training set learning performance. The objective of the learning process is to adjust the free parameters of the network so as to minimize \( \mathcal{E}_w \). To do this minimization we use an approximation similar in rationale to that we used for the derivation of the LMS algorithm in Chapter 5. Specifically, we consider a simple method of training in which the weights are updated on a pattern-by-pattern basis. The adjustments to the weights are made in accordance with the respective errors computed for each pattern presented to the network. 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 \( \mathcal{E}_w \) over the entire training set.

Consider then Fig. 6.3, which depicts neuron \( j \) being fed by a set of function signals produced by a layer of neurons to its left. The net internal activity level \( v_j(n) \) produced

![Signal-flow graph highlighting the details of output neuron \( j \).](image-url)
at the input of the nonlinearity associated with neuron \( j \) is therefore

\[
v_j(n) = \sum_{j' \in \mathcal{D}} w_{j'}(n) y_j(n)
\]  
(6.4)

where \( p \) is the total number of inputs (excluding the threshold) applied to neuron \( j \). The synaptic weight \( w_{j'} \) (corresponding to the fixed input \( y_{j'} = -1 \)) equals the threshold \( \theta \) applied to neuron \( j \). Hence the function signal \( y_j(n) \) appearing at the output of neuron \( j \) at iteration \( n \) is

\[
y_j(n) = \phi(v_j(n))
\]  
(6.5)

In a manner similar to the LMS algorithm, the back-propagation algorithm applies a correction \( \Delta w_{j'}(n) \) to the synaptic weight \( w_{j'}(n) \), which is proportional to the instantaneous gradient \( \delta E(n)/\partial w_{j'}(n) \). According to the chain rule, we may express this gradient as follows:

\[
\frac{\delta E(n)}{\partial w_{j'}(n)} = \frac{\delta E(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial u_j(n)} \frac{\partial u_j(n)}{\partial w_{j'}(n)}
\]  
(6.6)

The gradient \( \delta E(n)/\partial w_{j'}(n) \) represents a sensitivity factor, determining the direction of search in weight space for the synaptic weight \( w_{j'} \).

Differentiating both sides of Eq. (6.2) with respect to \( e_j(n) \), we get

\[
\frac{\delta E(n)}{\partial e_j(n)} = e_j(n)
\]  
(6.7)

Differentiating both sides of Eq. (6.1) with respect to \( y_j(n) \), we get

\[
\frac{\partial e_j(n)}{\partial y_j(n)} = -1
\]  
(6.8)

Next, differentiating Eq. (6.5) with respect to \( v_j(n) \), we get

\[
\frac{\partial y_j(n)}{\partial u_j(n)} = \phi'(v_j(n))
\]  
(6.9)

where the use of prime (on the right-hand side) signifies differentiation with respect to the argument. Finally, differentiating Eq. (6.4) with respect to \( w_{j'}(n) \) yields

\[
\frac{\partial u_j(n)}{\partial w_{j'}(n)} = y_j(n)
\]  
(6.10)

Hence, the use of Eqs. (6.7) to (6.10) in (6.6) yields

\[
\frac{\delta E(n)}{\partial w_{j'}(n)} = -e_j(n) \phi'(v_j(n)) y_j(n)
\]  
(6.11)

The correction \( \Delta w_{j'}(n) \) applied to \( w_{j'}(n) \) is defined by the delta rule

\[
\Delta w_{j'}(n) = -\eta \frac{\delta E(n)}{\partial w_{j'}(n)}
\]  
(6.12)

where \( \eta \) is a constant that determines the rate of learning; it is called the learning-rate parameter of the back-propagation algorithm. The use of the minus sign in Eq. (6.12) accounts for gradient descent in weight space. Accordingly, the use of Eq. (6.11) in (6.12) yields

\[
\Delta w_{j'}(n) = \eta \delta_j(n) y_j(n)
\]  
(6.13)
where the local gradient $\delta_j(n)$ is itself defined by

$$
\delta_j(n) = - \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}
$$

$$
= e_j(n) \varphi'_j(v_j(n))
$$

(6.14)

The local gradient points to required changes in synaptic weights. According to Eq. (6.14), the local gradient $\delta_j(n)$ for output neuron $j$ is equal to the product of the corresponding error signal $e_j(n)$ and the derivative $\varphi'_j(v_j(n))$ of the associated activation function.

From Eqs. (6.13) and (6.14) we note that a key factor involved in the calculation of the weight adjustment $\Delta w_{ij}(n)$ 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 I, 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 II, 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 indeed the credit-assignment problem considered in Section 2.6. It is solved in an elegant fashion by back-propagating the error signals through the network.

In the sequel, cases I and II are considered in turn.

**Case I: Neuron $j$ Is an Output Node**

When neuron $j$ is located in the output layer of the network, it would be supplied with a desired response of its own. Hence we may use Eq. (6.1) to compute the error signal $e_j(n)$ associated with this neuron; see Fig. 6.3. Having determined $e_j(n)$, it is a straightforward matter to compute the local gradient $\delta_j(n)$ using Eq. (6.14).

**Case II: 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. 6.4, which depicts neuron $j$ as a hidden node of the network. According to Eq. (6.14), we may redefine the local gradient $\delta_j(n)$ for hidden neuron $j$ as

$$
\delta_j(n) = - \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)}
$$

$$
= - \frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi'_j(v_j(n)), \quad \text{neuron } j \text{ is hidden}
$$

(6.15)

where, in the second line, we have made use of Eq. (6.9). To calculate the partial derivative $\partial \mathcal{E}(n)/\partial y_j(n)$, we may proceed as follows. From Fig. 6.4 we see that

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

(6.16)

which is a rewrite of Eq. (6.2) except for the use of index $k$ 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 II. In any event, differentiating Eq. (6.16) with respect to the function signal $y_j(n)$, we get

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_i e_k \frac{\partial e_k(n)}{\partial y_j(n)}$$  \hspace{1cm} (6.17)

Next, we use the chain rule for the partial derivative $\partial e_k(n)/\partial y_j(n)$, and thus rewrite Eq. (6.17) in the equivalent form

$$\frac{\partial E(n)}{\partial y_j(n)} = \sum_i e_k(n) \frac{\partial e_k(n)}{\partial v_i(n)} \frac{\partial v_i(n)}{\partial y_j(n)}$$  \hspace{1cm} (6.18)

However, from Fig. 6.4, we note that

$$e_k(n) = d_k(n) - y_k(n)$$

$$= d_k(n) - \varphi(v_k(n)),$$  \hspace{0.5cm} neuron $k$ is an output node  \hspace{1cm} (6.19)

Hence

$$\frac{\partial e_k(n)}{\partial v_i(n)} = -\varphi'(v_k(n))$$  \hspace{1cm} (6.20)

We also note from Fig. 6.4 that for neuron $k$, the net internal activity level is

$$v_k(n) = \sum_{j=0}^{q} w_{kj}(n) y_j(n)$$  \hspace{1cm} (6.21)

where $q$ is the total number of inputs (excluding the threshold) applied to neuron $k$. Here again, the synaptic weight $w_{kj}(n)$ is equal to the threshold $\theta_k(n)$ applied to neuron $k$, and the corresponding input $y_0$ is fixed at the value $-1$. In any event, differentiating Eq. (6.21) with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$  \hspace{1cm} (6.22)
Thus, using Eqs. (6.20) and (6.22) in (6.18), we get the desired partial derivative:

\[
\frac{\partial \delta(n)}{\partial y_j(n)} = -\sum_i e_i(n) \varphi'_i(u_i(n)) w_{ij}(n) = -\sum_i \delta_i(n) w_{ij}(n)
\]

(6.23)

where, in the second line, we have used the definition of the local gradient \(\delta_i(n)\) given in Eq. (6.14) with the index \(k\) substituted for \(j\).

Finally, using Eq. (6.23) in (6.15), we get the local gradient \(\delta_j(n)\) for hidden neuron \(j\), after rearranging terms, as follows:

\[
\delta_j(n) = \varphi'_j(v_j(n)) \sum_i \delta_i(n) w_{ij}(n), \quad \text{neuron } j \text{ is hidden}
\]

(6.24)

The factor \(\varphi'_j(v_j(n))\) involved in the computation of the local gradient \(\delta_j(n)\) in Eq. (6.24) 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 \(\delta_i(n)\), requires knowledge of the error signals \(e_i(n)\), for all those neurons that lie in the layer to the immediate right of hidden neuron \(j\), and that are directly connected to neuron \(j\); see Fig. 6.4. The second set of terms, the \(w_{ij}(n)\), consists of the synaptic weights associated with these connections.

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

\[
\begin{pmatrix}
\text{Weight} \\
\text{correction}
\end{pmatrix}
= \begin{pmatrix}
\text{learning-} \\
\text{rate parameter}
\end{pmatrix}
\cdot \begin{pmatrix}
\text{local} \\
\text{gradient}
\end{pmatrix}
\cdot \begin{pmatrix}
\text{input signal} \\
\text{of neuron } j
\end{pmatrix}
\]

(6.25)

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

1. If neuron \(j\) is an output node, \(\delta_j(n)\) equals the product of the derivative \(\varphi'_j(v_j(n))\) and the error signal \(e_j(n)\), both of which are associated with neuron \(j\); see Eq. (6.14).

2. If neuron \(j\) is a hidden node, \(\delta_j(n)\) equals the product of the associated derivative \(\varphi'_j(v_j(n))\) and the weighted sum of the \(B's\) computed for the neurons in the next hidden or output layer that are connected to neuron \(j\); see Eq. (6.24).

**The Two Passes of Computation**

In the application of the back-propagation algorithm, two distinct passes of computation may be distinguished. The first pass is referred to as the forward pass, and the second one 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. Specifically, the function signal appearing at the output of neuron \(j\) is computed as

\[
y_j(n) = \varphi(v_j(n))
\]

(6.26)

where \(v_j(n)\) is the net internal activity level of neuron \(j\), defined by

\[
v_j(n) = \sum_{i=0}^{p} w_{ij}(n) y_i(n)
\]

(6.27)

where \(p\) is the total number of inputs (excluding the threshold) applied to neuron \(j\), and \(w_{ij}(a)\) 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, then the index \( i \) refers to the \( i \)th input terminal of the network, for which we write
\[
y_i(n) = x_i(n)
\]
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, the index \( j \) refers to the \( j \)th output terminal of the network, for which we write
\[
y_j(n) = o_j(n)
\]
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 \( \delta \) (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. (6.25). For a neuron located in the output layer, the \( \delta \) is simply equal to the error signal of that neuron multiplied by the first derivative of its nonlinearity. Hence we use Eq. (6.25) to compute the changes to the weights of all the connections feeding into the output layer. Given the \( \delta \)'s for the neurons of the output layer, we next use Eq. (6.24) to compute the \( \delta \)'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 made.

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.

**Sigmoidal Nonlinearity**

The computation of the \( \delta \) for each neuron of the multilayer perceptron requires knowledge of the derivative of the activation function \( \varphi(\cdot) \) associated with that neuron. For this derivative to exist, we require the function \( \varphi(\cdot) \) to be continuous. In basic terms, *differentiability* is the only requirement that an activation function would have to satisfy. An example of a continuously differentiable nonlinear activation function commonly used in multilayer perceptrons is the **sigmoidal nonlinearity**, a particular form of which is defined for neuron \( j \) by the **logistic function**
\[
y_j(n) = \varphi_j(v_j(n))
\]
\[
= \frac{1}{1 + \exp(-u_j(n))}, \quad -\infty < u_j(n) < \infty
\]
where \( u_j(n) \) is the net internal activity level of neuron \( j \). According to this nonlinearity, the amplitude of the output lies inside the range \( 0 \leq y_j \leq 1 \). Another type of sigmoidal nonlinearity is the **hyperbolic tangent**, which is antisymmetric with respect to the origin and for which the amplitude of the output lies inside the range \( -1 \leq y_j \leq +1 \). We will have more to say on this latter form of nonlinearity in Section 6.7. For the time being, we are going to concentrate on the logistic function of Eq. (6.30) as the source of nonlinearity.
Differentiating both sides of Eq. (6.30) with respect to \( u_j(n) \), we get

\[
\frac{\partial y_j(n)}{\partial u_j(n)} = \phi'(v_j(n)) = \frac{\exp(-v_j(n))}{[1 + \exp(-v_j(n))]^2}
\] (6.31)

Using Eq. (6.30) to eliminate the exponential term \( \exp(-v_j(n)) \) from Eq. (6.31), we may express the derivative \( \phi'(v_j(n)) \) as

\[
\phi'(v_j(n)) = y_j(n)[1 - y_j(n)]
\] (6.32)

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

\[
\delta_j(n) = e_j(n)\phi'(v_j(n)) = [d_j(n) - o_j(n)]o_j(n)[1 - o_j(n)], \quad \text{neuron } j \text{ is an output node}
\] (6.33)

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

\[
\delta_j(n) = \phi'(v_j(n)) \sum_k \delta_k(n)w_{jk}(n) = y_j(n)[1 - y_j(n)] \sum_k \delta_k(n)w_{jk}(n), \quad \text{neuron } j \text{ is hidden}
\] (6.34)

Note from Eq. (6.32) that the derivative \( \phi'(v_j(n)) \) 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 \( \phi'(v_j(n)) \), it follows that for a sigmoidal activation function the synaptic weights are changed the most for those neurons in the network for which 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.

**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 \( \eta \), the smaller will be the changes to the synaptic weights in the network 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 \( \eta \) too large so as 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 and yet avoiding the danger of instability is to modify the delta rule of Eq. (6.13) by including a *momentum* term, as shown by \(^1\) (Rumelhart et al., 1986a)

\[
\Delta w_{jk}(n) = \alpha \Delta w_{jk}(n - 1) + \eta \delta_k(n)y_j(n)
\] (6.35)

\(^1\) For the special case of the LMS algorithm, which is a linear adaptive filtering algorithm, it has been shown that use of the momentum constant \( \alpha \) reduces the stable range of the learning-rate parameter \( \eta \), and could thus lead to instability if \( \eta \) is not adjusted appropriately. Moreover, the misadjustment increases with increasing \( \alpha \) (for details, see Roy and Shynk, 1990).
where $\alpha$ is usually a positive number called the momentum constant. It controls the feedback loop acting around $\Delta w_p(n)$, as illustrated in Fig. 6.5, where $z^{-1}$ is the unit-delay operator. Equation (6.35) is called the generalized delta rule; it includes the delta rule of Eq. (6.13) 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 $\alpha$, we rewrite Eq. (6.35) as a time series with index $t$. The index $t$ goes from the initial time 0 to the current time $n$. Equation (6.35) may be viewed as a first-order difference equation in the weight correction $\Delta w_p(n)$. Hence, solving this equation for $\Delta w_p(n)$, we have

$$\Delta w_p(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta(t) y(t)$$

which represents a time series of length $n + 1$. From Eqs. (6.11) and (6.14) we note that the product $\delta(t) y(n)$ is equal to $-\partial \xi(n)/\partial w_p(t)$. Accordingly, we may rewrite Eq. (6.36) in the equivalent form

$$\Delta w_p(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \xi(t)}{\partial w_p(t)}$$

Based on this relation, we may make the following insightful observations (Watrous, 1987; Jacobs, 1988; Goggin et al., 1989):

1. The current adjustment $\Delta w_p(n)$ 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 $\alpha$ is zero, the back-propagation algorithm operates without momentum. Note also that the momentum constant $\alpha$ can be positive or negative, although it is unlikely that a negative $\alpha$ would be used in practice.

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

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

Thus, the incorporation of momentum in the back-propagation algorithm represents a minor modification to the weight update, and yet it can have highly beneficial effects on learning behavior of the algorithm. The momentum term may also have the benefit of

---

$^2$For a derivation of the back-propagation algorithm including the momentum constant from first principles, see Hagiwara (1992).
6.3 / Derivation of the Back-Propagation Algorithm

preventing the learning process from terminating in a shallow local minimum on the error surface.

**Additional Notes.** In deriving the back-propagation algorithm, it was assumed that the learning-rate parameter is a constant denoted by \( \eta \). In reality, however, it should be defined as \( \eta_{ij} \); 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 shall have more to say on this issue in Sections 6.7 and 6.15.

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 \( \eta_{ij} \) for synaptic weight \( w_{ij} \) equal to zero.

Another point of interest concerns the manner in which the various layers of the back-propagation network are interconnected. In the development of the back-propagation algorithm presented here, we proceeded on the premise that the neurons in each layer of the network receive their inputs from other units in the previous layer, as illustrated in Fig. 6.1. In fact, there is no reason why a neuron in a certain layer may not receive inputs from other units in earlier layers of the network. In handling such a neuron, there are two kinds of error signals to be considered: (1) an error signal that results from the direct comparison of the output signal of that neuron with a desired response; and (2) an error signal that is passed through the other units whose activation it affects. In this situation, the correct procedure to deal with the network is simply to add the changes in synaptic weights dictated by the direct comparison to those propagated back from the other units (Rumelhart et al., 1986b).

**Pattern 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. 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 threshold 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.

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

1. **Pattern Mode.** In the pattern mode of back-propagation learning, weight updating is performed after the presentation of each training example; this is indeed the very mode of operation for which the derivation of the back-propagation algorithm presented here applies. To be specific, consider an epoch consisting of \( N \) training examples (patterns) arranged in the order \([x(1),d(1)], \ldots, [x(N),d(N)]\). The first example \([x(1),d(1)]\) 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 threshold levels of the network. Then, the second example \([x(2),d(2)]\) in the epoch is presented, and the sequence of forward and backward computations is repeated, resulting...
in further adjustments to the synaptic weights and threshold levels. This process is continued until the last example \( x(N), d(N) \) in the epoch is accounted for. Let \( \Delta w_j(n) \) denote the change applied to synaptic weight \( w_j \) after the presentation of pattern \( n \). Then the net weight change \( \Delta \tilde{w}_j \), averaged over the entire training set of \( N \) patterns, is given by

\[
\Delta \tilde{w}_j = \frac{1}{N} \sum_{n=1}^{N} \Delta w_j(n)
\]

\[
= -\eta \sum_{n=1}^{N} \frac{\partial \tilde{e}(n)}{\partial w_j(n)}
\]

\[
= -\eta \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_j(n)}
\]

(6.38)

where in the second and third lines we have made use of Eqs. (6.12) and (6.2), respectively.

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. (6.2) and (6.3), reproduced here in the composite form:

\[
\bar{e}_{av} = \frac{1}{2N} \sum_{n=1}^{N} \sum_{j \in \text{output}} e_j^2(n)
\]

(6.39)

where the error signal \( e_j(n) \) pertaining to output neuron \( j \) for training example \( n \) and which is defined by Eq. (6.1). The error \( e_i(n) \) equals the difference between \( d_i(n) \) and \( y_i(n) \), which represent the \( j \)th element of the desired response vector \( d(n) \) and the corresponding value of the network output, respectively. In Eq. (6.39) 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 \( \eta \), the adjustment applied to synaptic weight \( w_i \), connecting neuron \( i \) to neuron \( j \), is defined by the delta rule

\[
\Delta w_{ji} = -\eta \frac{\partial \bar{e}_{av}}{\partial w_{ji}}
\]

\[
= -\eta \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}}
\]

(6.40)

To calculate the partial derivative \( \partial e_j(n)/\partial w_{ji} \) we proceed in the same way as before. According to Eq. (6.40), in the batch mode the weight adjustment \( \Delta w_{ji} \) is made only after the entire training set has been presented to the network.

Comparing Eqs. (6.38) and (6.40), we clearly see that the average weight change \( \Delta \tilde{w}_j \) made in the pattern mode of training is different from the corresponding value \( \Delta w_{ji} \) made in the batch mode, presumably for the same reduction in the average squared error \( \bar{e}_{av} \) that results from the presentation of the entire training set. Indeed, \( \Delta \tilde{w}_j \) for the pattern-to-pattern mode represents an estimate of \( \Delta w_{ji} \) for the batch mode.

From an “on-line” operational point of view, the pattern 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, which, in turn, makes it less likely for the back-propagation algorithm to be trapped in a local minimum. On the other hand, the use of batch mode of training provides a more accurate estimate of the gradient vector. In the final analysis, however, the relative
effectiveness of the two training modes depends on the problem at hand (Hertz et al., 1991).

Stopping Criteria

The back-propagation algorithm cannot, in general, be shown to converge, nor are there 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, the logical thing to do is to think in terms of the unique properties of a local or global minimum of the error surface. Let the weight vector \( w^* \) denote a minimum, be it local or global. A necessary condition for \( w^* \) to be a minimum is that the gradient vector \( g(w) \) (i.e., first-order partial derivative) of the error surface with respect to the weight vector \( w \) be zero at \( w = w^* \). 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 \( g(w) \).

Another unique property of a minimum that we can use is the fact that the cost function or error measure \( E_n(w) \) is stationary at the point \( w = w^* \). 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.

Typically, the rate of change in the average squared error is 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.

A variation of this second criterion for convergence of the algorithm is to require that the maximum value of the average squared error \( E_n(w) \) be equal to or less than a sufficiently small threshold. Kramer and Sangiovanni-Vincentelli (1989) suggest a hybrid criterion of convergence consisting of this latter threshold and a gradient threshold, as stated here:

- The back-propagation algorithm is terminated at the weight vector \( w_{\text{final}} \) when \( \|g(w_{\text{final}})\| \leq \epsilon \), where \( \epsilon \) is a sufficiently small gradient threshold, or \( E_n(w_{\text{final}}) \leq \tau \), where \( \tau \) is a sufficiently small error energy threshold.