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 $n = 1, 2, \ldots, N$, 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.

4.11 HESSIAN MATRIX

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

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

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 computation[7] 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 λ_{max} is the largest eigenvalue of the Hessian and λ_{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-

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

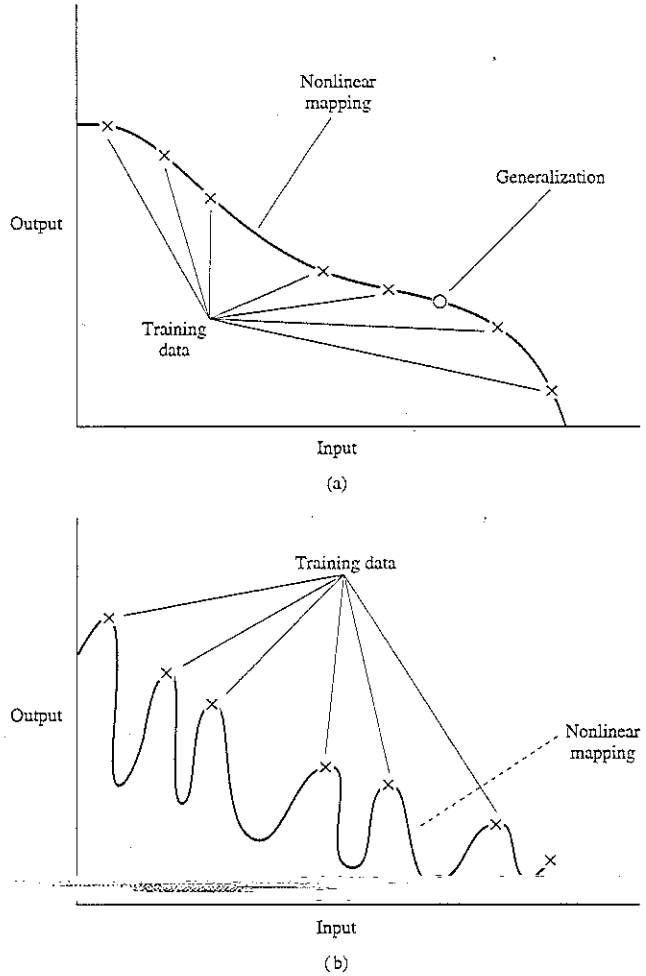

FIGURE 4.19 (a) Properly fitted data (good generalization) (b) Overfitted data (poor generalization).

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

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

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.

4.13 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 $\varphi(\cdot)$ 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 $f \ni C(I_{m_0})$ and $\epsilon > 0$, there exist an integer m_1 and sets of real constants α_i, b_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define*

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

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

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

> *for all $x_1, x_2, \ldots, x_{m_0}$ 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 nonconstant, bounded, and monotone-increasing function; it therefore satisfies the conditions imposed on the function $\varphi(\cdot)$. 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 $x_1, \ldots, x_{m_0}$.
2. Hidden neuron i has synaptic weights $w_{i1}, \ldots, w_{im_0}$, and bias b_i.
3. The network output is a linear combination of the outputs of the hidden neurons, with $\alpha_1, \ldots, \alpha_{m_1}$ 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* $x_1, \ldots, x_{m_0}$ and a desired (target) output $f(x_1, \ldots, x_{m_0})$. 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 $\tilde{f}(\boldsymbol{\omega})$ 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 $f(x)$ is defined in terms of its Fourier transform $\tilde{f}(\boldsymbol{\omega})$ by the inverse formula:

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

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:

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

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 $C_f' = (2\,r\,C_f)^2$.

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*:

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

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:

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

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

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

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 $\{(\mathbf{x}_i, d_i)\}$, $i = 1, 2, \ldots, N$, 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 $\mathbf{f}: \mathbb{R}^m \rightarrow \mathbb{R}^M$, a compact subset $\mathscr{C} \subseteq \mathbb{R}^M$ that is included in the image of $\mathbf{f}$, and an $\epsilon > 0$, find a vector-valued function $\varphi: \mathbb{R}^M \rightarrow \mathbb{R}^m$ such that the following condition is satisfied:
>
> $$\|\varphi(\mathbf{f}(\mathbf{u})) - \mathbf{u}\| < \epsilon \qquad \text{for } \mathbf{u} \in \mathscr{C}$$

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

$$\mathbf{x}(n) = \mathbf{f}(\mathbf{x}(n-1), \mathbf{u}(n))$$

It is assumed that $\mathbf{f}$ is *invertible*, so that we may solve for $\mathbf{u}(n)$ as a function of $\mathbf{x}(n)$ for any $\mathbf{x}(n-1)$. The function $\mathbf{f}$ 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 $\mathbf{f}$, $\mathscr{C}$, 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 principle[9] (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

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

In words, the kth 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

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

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 $\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n$ 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

$$\mathcal{F}_{cv} = \min_{k=1, 2, \ldots, \nu} \{e_t''(\mathcal{F}_k)\} \tag{4.92}$$

where ν corresponds to $W_\nu \leq W_{\max}((1-r)N)$, and $e_t''(\mathcal{F}_k)$ 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 $\nu = p + q$ fully-connected multilayer perceptrons structured as follows:

- p multilayer perceptrons with a single hidden layer of increasing size $h_1' < h_2' < \ldots < h_p'$.
- 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 $h_1'' < h_2'' < \ldots < h_q''$.

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

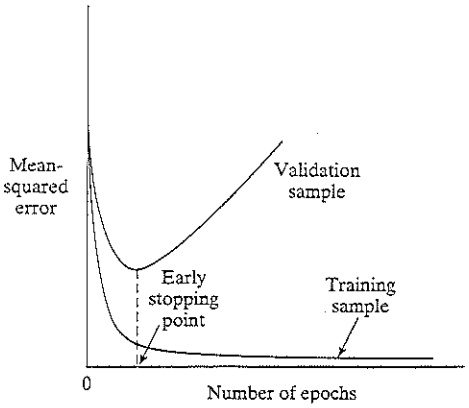

FIGURE 4.20 Illustration of the early-stopping rule based on cross-validation.

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

$$r_{\text{opt}} \simeq 1 - \frac{1}{\sqrt{2W}}, \qquad \text{large } W \tag{4.93}$$

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*

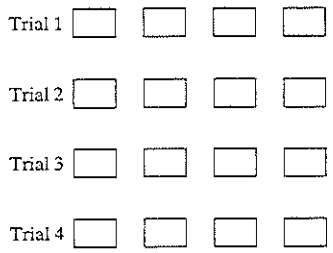

FIGURE 4.21 Illustration of the hold-out method of cross-validation. For a given trial, the shaded subset of data is used to validate the model trained on the remaining data.

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.

4.15 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:

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

The first term, $\mathscr{E}_s(\mathbf{w})$, 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, $\mathscr{E}_c(\mathbf{w})$, 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 $\mathscr{E}_c(\mathbf{w})$ is the kth order smoothing integral

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

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 kth 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

$$\mathscr{E}_c(\mathbf{w}) = \|\mathbf{w}\|^2 = \sum_{i \in \mathscr{E}_{\text{total}}} w_i^2 \tag{4.96}$$

where the set $\mathscr{E}_{\text{total}}$ 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)

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

where w_0 is a preassigned parameter, and w_i refers to the weight of some synapse i in the network. The set $\mathscr{E}_{\text{total}}$ 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 ith 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 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).

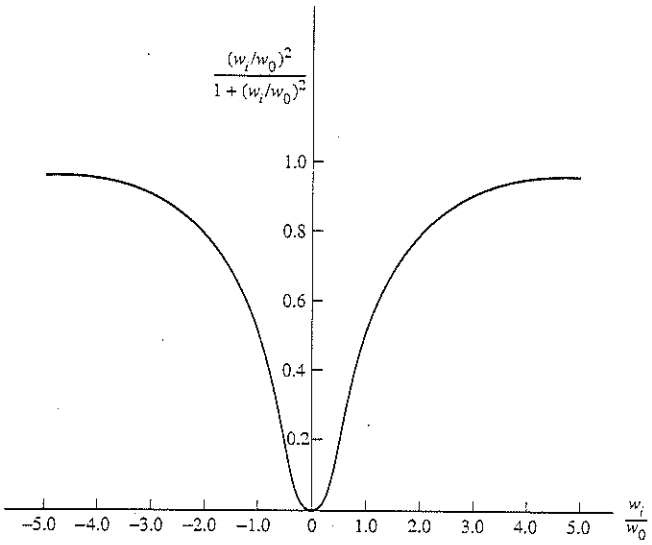

FIGURE 4.22 The complexity penalty term $(w_i/w_0)^2/[1 + (w_i/w_0)^2]$ plotted versus w_i/w_0.

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:

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

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

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

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 $\mathscr{E}_{av}$ using a *Taylor series* about the operating point, described as follows:

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

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 $\mathscr{E}_{av}$. 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

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

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 $\mathscr{E}_{av}$ 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

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

where $\mathbf{1}_i$ is the *unit vector* whose elements are all zero, except for the ith 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 ith 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*

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

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

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

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

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

where $\mathbf{H}^{-1}$ is the inverse of the Hessian matrix $\mathbf{H}$, and $[\mathbf{H}^{-1}]_{i,i}$ is the i, i-th element of this inverse matrix. The Lagrangian S_i optimized with respect to $\Delta\mathbf{w}$, subject to the constraint that the ith 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 ith 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 NETtalk 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. NETtalk, 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

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

where $o(n)$ is the actual output of the network on the presentation of the nth 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 therefore 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 $\mathscr{E}_{av}$ with respect to $\mathbf{w}$ is therefore

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

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

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

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 $\mathscr{E}_{av}$ 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

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

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

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

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:

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

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

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

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

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

$$\mathbf{D} = 1$$

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

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

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 nth example represented by the vector $\boldsymbol{\xi}(n)$. 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

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

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).

TABLE 4.6 Summary of the Optimal Brain Surgeon Algorithm

1. Train the given multilayer perceptron to minimum mean-square error.
2. Use the procedure described in Section 4.10 to compute the vector

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

where $F(\mathbf{w}, \mathbf{x}(n))$ is the input–output mapping realized by the multilayer perceptron with an overall weight vector $\mathbf{w}$, and $\mathbf{x}(n)$ is the input vector.
3. Use the recursion (4.111) to compute the inverse Hessian $\mathbf{H}^{-1}$.
4. Find the i that corresponds to the smallest saliency:

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

where $[\mathbf{H}^{-1}]_{i,i}$ is the (i, i)th element of $\mathbf{H}^{-1}$. If the saliency S_i is much smaller than the mean-square $\mathscr{E}_{av}$, then delete synaptic weight w_i, and proceed to step 4. Otherwise, go to step 5.
5. Update all the synaptic weights in the network by applying the adjustment:

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

Go to step 2.
6. Stop the computation when no more weights can be deleted from the network without a large increase in the mean-square error. (It may be desirable to retrain the network at this point).

4.16 VIRTUES AND LIMITATIONS OF BACK-PROPAGATION LEARNING

The back-propagation algorithm has emerged as the most popular algorithm for the supervised training of multilayer perceptrons. Basically, it is a gradient (derivative) technique and *not* an optimization technique. Back-propagation has two distinct properties:

- It is *simple* to compute locally.
- It performs *stochastic* gradient descent in weight space (for pattern-by-pattern updating of synaptic weights).

These two properties of back-propagation learning in the context of a multilayer perceptron are responsible for its advantages and disadvantages.

Connectionism

The back-propagation algorithm is an example of a *connectionist paradigm* that relies on local computations to discover the information-processing capabilities of neural networks. This form of computational restriction is referred to as the *locality constraint*, in the sense that the computation performed by the neuron is influenced solely by those neurons that are in physical contact with it. The use of local computations in the design of artificial neural networks is usually advocated for three principal reasons:

1. Artificial neural networks that perform local computations are often held up as metaphors for biological neural networks.

2. The use of local computations permits a graceful degradation in performance due to hardware errors, and therefore provides the basis for a fault-tolerant network design.
3. Local computations favor the use of parallel architectures as an efficient method for the implementation of artificial neural networks.

Taking these three points in reverse order, point 3 is entirely justified in the case of back-propagation learning. In particular, the back-propagation algorithm has been implemented successfully on parallel computers by many investigators, and VLSI architectures have been developed for the hardware realization of multilayer perceptrons (Hammerstrom, 1992a, 1992b). Point 2 is justified so long as certain precautions are taken in the application of the back-propagation algorithm, as described in Kerlirzin and Vallet (1993). As for point 1, relating to the biological plausibility of back-propagation learning, it has been seriously questioned on the following grounds (Shepherd, 1990b; Crick, 1989; Stork, 1989):

1. The reciprocal synaptic connections between the neurons of a multilayer perceptron may assume weights that are excitatory or inhibitory. In the real nervous system, however, neurons usually appear to be the one or the other. This is one of the most serious of the unrealistic assumptions made in neural network models.
2. In a multilayer perceptron, hormonal and other types of global communications are ignored. In real nervous systems, these types of global communication are critical for state-setting functions such as arousal, attention, and learning.
3. In back-propagation learning, a synaptic weight is modified by a presynaptic activity and an error (learning) signal independent of postsynaptic activity. There is evidence from neurobiology to suggest otherwise.
4. In a neurobiological sense, the implementation of back-propagation learning requires the rapid transmission of information backward along an axon. It appears highly unlikely that such an operation actually takes place in the brain.
5. Back-propagation learning implies the existence of a "teacher," which in the context of the brain would presumably be another set of neurons with novel properties. The existence of such neurons is biologically implausible.

However, these neurobiological misgivings do not belittle the engineering importance of back-propagation learning as a tool for information processing, as evidenced by its successful application in numerous highly diverse fields, including the simulation of neurobiological phenomena (see, for example, Robinson (1992)).

Feature Detection

As discussed in Section 4.9, the hidden neurons of a multilayer perceptron trained with the back-propagation algorithm play a critical role as feature detectors. A novel way in which this important property of the multilayer perceptron can be exploited is in its use as a *replicator* or *identity map* (Rumelhart et al., 1986b; Cottrel et al., 1987). Figure 4.23 illustrates how this can be accomplished for the case of a multilayer perceptron using a single hidden layer. The network layout satisfies the following structural requirements, as illustrated in Fig. 4.23a:

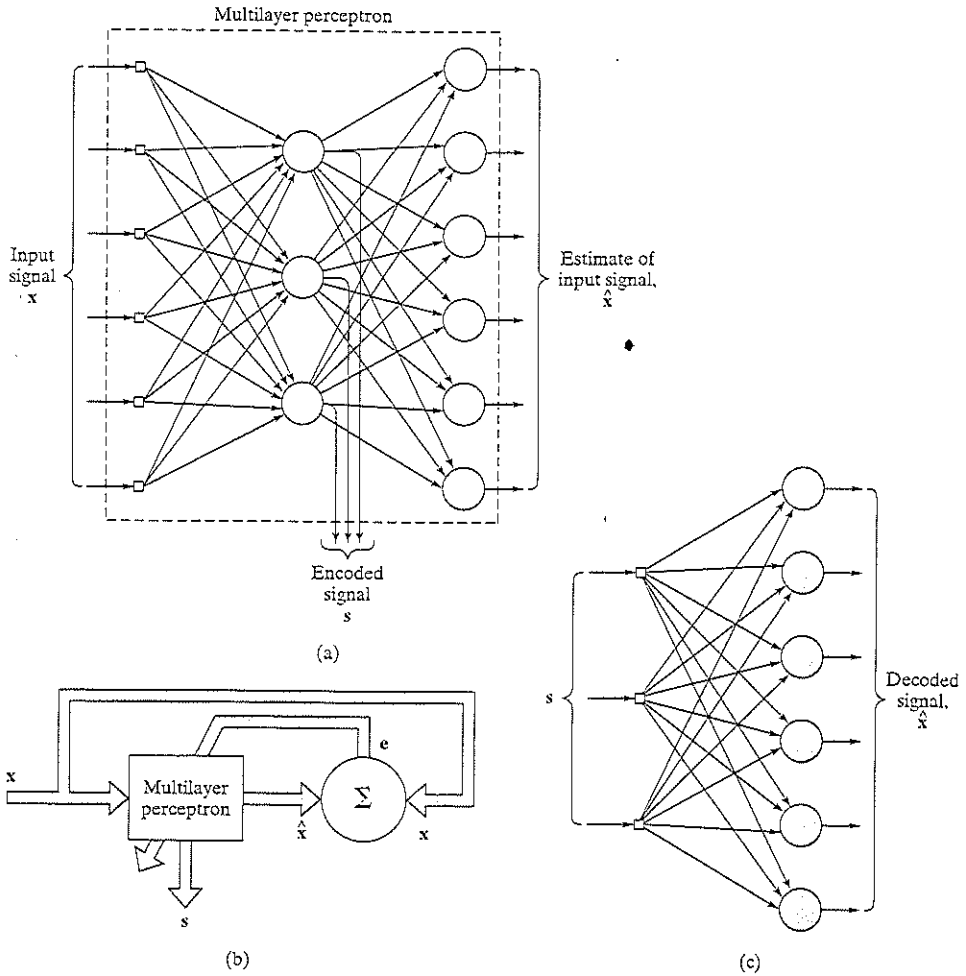

FIGURE 4.23 (a) Replicator network (identity map) with a single hidden layer used as an encoder. (b) Block diagram for the supervised training of the replicator network. (c) Part of the replicator network used as a decoder.

- The input and output layers have the same size, m.
- The size of the hidden layer, M, is smaller than m.
- The network is fully connected.

A given pattern, $\mathbf{x}$, is simultaneously applied to the input layer as the stimulus and to the output layer as the desired response. The actual response of the output layer, $\hat{\mathbf{x}}$, is intended to be an "estimate" of $\mathbf{x}$. The network is trained using the back-propagation algorithm in the usual way, with the estimation error vector $(\mathbf{x} - \hat{\mathbf{x}})$ treated as the error signal, as illustrated in Fig. 4.23b. The training is performed in an *unsupervised* manner (i.e., without the need for a teacher). By virtue of the special structure built into the design of the multilayer perceptron, the network is *constrained* to perform identity mapping through its hidden layer. An *encoded* version of the input pattern, denoted by $\mathbf{s}$, is produced at the output of the hidden layer, as depicted in Fig. 4.23a. In effect, the fully trained multilayer perceptron performs the role of an "encoder." To reconstruct an estimate $\hat{\mathbf{x}}$ of the original input pattern $\mathbf{x}$ (i.e., to perform *decoding*), we apply the encoded signal to the hidden layer of the replicator network, as illustrated in Fig. 4.23c. In effect, this latter network performs the role of a "decoder." The smaller we make the size M of the hidden layer compared to the size m of the input/output layer, the more effective the configuration of Fig. 4.23a will be as a *data compression system*.[12]

Function Approximation

A multilayer perceptron trained with the back-propagation algorithm manifests itself as a *nested sigmoidal scheme,* written in the following compact form for the case of a single output:

$$F(\mathbf{x},\mathbf{w}) = \varphi\Big(\sum_k w_{ok}\,\varphi\Big(\sum_j w_{kj}\,\varphi\Big(\cdots\varphi\Big(\sum_i w_{ji}\,x_i\Big)\Big)\Big)\Big) \tag{4.113}$$

where $\varphi(\cdot)$ is a common sigmoid activation function, w_{ok} is the synaptic weight from neuron k in the last hidden layer to the single output neuron o, and so on for the other synaptic weights, and x_i is the ith element of the input vector $\mathbf{x}$. The weight vector $\mathbf{w}$ denotes the entire set of synaptic weights ordered by layer, then neurons in a layer, and then synapses in a neuron. The scheme of nested nonlinear functions described in Eq. (4.113) is unusual in classical approximation theory. It is a *universal approximator* as discussed in Section 4.13.

In the context of approximation, the use of back-propagation learning offers another useful property. Intuition suggests that a multilayer perceptron with smooth activation functions should have output function derivatives that can also approximate the derivatives of an unknown input–output mapping. A proof of this result is presented in Hornik et al. (1990). In fact, it is shown that multilayer perceptrons can approximate functions that are not differentiable in the classical sense, but possess a generalized derivative as in the case of piecewise differentiable functions. The approximation results reported by Hornik et al. provide a previously missing theoretical justification for the use of multilayer perceptrons in applications that require the approximation of a function and its derivatives.

Computational Efficiency

The *computational complexity* of an algorithm is usually measured in terms of the number of multiplications, additions, and storage involved in its implementation, as discussed in Chapter 2. A learning algorithm is said to be *computationally efficient* when its computational complexity is *polynomial* in the number of adjustable parameters that are to be updated from one iteration to the next. On this basis it can be said that the back-propagation algorithm is computationally efficient. Specifically, in using it to train a multilayer perceptron containing a total of W synaptic weights (including

biases), its computational complexity is linear in W. This important property of the back-propagation algorithm can be readily verified by examining the computations involved in performing the forward and backward passes summarized in Section 4.5. In the forward pass, the only computations involving the synaptic weights are those that pertain to the induced local fields of the various neurons in the network. Here we see from Eq. (4.44) that these computations are all linear in the synaptic weights of the network. In the backward pass, the only computations involving the synaptic weights are those that pertain to (1) the local gradients of the hidden neurons, and (2) the updating of the synaptic weights themselves, as shown in Eqs. (4.46) and (4.47), respectively. Here we also see that these computations are all linear in the synaptic weights of the network. The conclusion is therefore that the computational complexity of the back-propagation algorithm is linear in W, that is, it is $O(W)$.

Sensitivity Analysis

Another computational benefit gained from the use of back-propagation learning is the efficient manner in which we can carry out a sensitivity analysis of the input–output mapping realized by the algorithm. The *sensitivity* of an input–output mapping function F with respect to a parameter of the function, denoted by ω, is defined by

$$S_{\omega}^{F} = \frac{\partial F/F}{\partial \omega/\omega} \tag{4.114}$$

Consider then a multilayer perceptron trained with the back-propagation algorithm. Let the function $F(\mathbf{w})$ be the input–output mapping realized by this network; $\mathbf{w}$ denotes the vector of all synaptic weights (including biases) contained in the network. In Section 4.10 we showed that the partial derivatives of the function $F(\mathbf{w})$ with respect to all the elements of the weight vector $\mathbf{w}$ can be computed efficiently. In particular, examining Eqs. (4.81) to (4.83) together with Eq. (4.114), we see that the complexity involved in computing each of these partial derivatives is linear in W, the total number of weights contained in the network. This linearity holds irrespective of where the synaptic weight in question appears in the chain of computations.

Robustness

In Chapter 3 we pointed out that the LMS algorithm is robust in the sense that disturbances with small energy can only give rise to small estimation errors. If the underlying observation model is linear, the LMS algorithm is an H^{∞}-optimal filter (Hassibi et al., 1993, 1996). What this means is that the LMS algorithm minimizes the *maximum energy gain* from the disturbances to the estimation errors.

If, on the other hand, the underlying observation model is nonlinear, Hassibi and Kailath. (1995) have shown that the back-propagation algorithm is a *locally* H^{∞}-optimal filter. The term "local" used here means that the initial value of the weight vector used in the back-propagation algorithm is sufficiently close to the optimum value $\mathbf{w}^*$ of the weight vector to ensure that the algorithm does not get trapped in a poor local minimum. In conceptual terms, it is satisfying to see that the LMS and back-propagation algorithms belong to the same class of H^{∞}-optimal filters.

Convergence

The back-propagation algorithm uses an "instantaneous estimate" for the gradient of the error surface in weight space. The algorithm is therefore *stochastic* in nature; that is, it has a tendency to zigzag its way about the true direction to a minimum on the error surface. Indeed, back-propagation learning is an application of a statistical method known as *stochastic approximation* that was originally proposed by Robbins and Monro (1951). Consequently, it tends to converge slowly. We may identify two fundamental causes for this property (Jacobs, 1988):

1. The error surface is fairly flat along a weight dimension, which means that the derivative of the error surface with respect to that weight is small in magnitude. In such a situation, the adjustment applied to the weight is small, and consequently many iterations of the algorithm may be required to produce a significant reduction in the error performance of the network. Alternatively, the error surface is highly curved along a weight dimension, in which case the derivative of the error surface with respect to that weight is large in magnitude. In this second situation, the adjustment applied to the weight is large, which may cause the algorithm to overshoot the minimum of the error surface.
2. The direction of the negative gradient vector (i.e., the negative derivative of the cost function with respect to the vector of weights) may point away from the minimum of the error surface: hence the adjustments applied to the weights may induce the algorithm to move in the wrong direction.

Consequently, the rate of convergence in back-propagation learning tends to be relatively slow, which in turn may make it computationally excruciating. According to the empirical study of Saarinen et al. (1992), the local convergence rates of the back-propagation algorithm are *linear*, which is justified on the grounds that the Jacobian matrix is almost rank deficient, and so is the Hessian matrix. These are consequences of the intrinsically ill-conditioned nature of neural-network training problems. Saarinen et al. interpret the linear local convergence rates of back-propagation learning in one of two ways:

- It is vindication of back-propagation (gradient descent) in the sense that higher-order methods may not converge much faster while requiring more computational effort; or
- Large-scale neural-network training problems are so inherently difficult to perform that no supervised learning strategy is feasible, and other approaches such as the use of preprocessing may be necessary.

We explore the issue of convergence more fully in Section 4.17, and explore the issue of preprocessing the input in Chapter 8.

Local Minima

Another peculiarity of the error surface that impacts the performance of the back-propagation algorithm is the presence of *local minima* (i.e., isolated valleys) in addition to global minima. Since back-propagation learning is basically a hill climbing

technique, it runs the risk of being trapped in a local minimum where every small change in synaptic weights increases the cost function. But somewhere else in the weight space there exists another set of synaptic weights for which the cost function is smaller than the local minimum in which the network is stuck. It is clearly undesirable to have the learning process terminate at a local minimum, especially if it is located far above a global minimum.

The issue of local minima in back-propagation learning has been raised in the epilogue of the enlarged edition on the classic book by Minsky and Papert (1988), where most of the attention is focused on a discussion of the two-volume book, *Parallel Distributed Processing,* by Rumelhart and McClelland (1986). In Chapter 8 of the latter book it is claimed that getting trapped in a local minimum is rarely a practical problem for back-propagation learning. Minsky and Papert counter by pointing out that the entire history of pattern recognition shows otherwise. Gori and Tesi (1992) describe a simple example where, although a nonlinearly separable set of patterns could be learned by the chosen network with a single hidden layer, back-propagation learning can get stuck in a local minimum.[13]

Scaling

In principle, neural networks such as multilayer perceptrons trained with the back-propagation algorithm offer the potential of universal computing machines. However, for that potential to be fully realized, we have to overcome the *scaling problem,* which addresses the issue of how well the network behaves (e.g., as measured by the time required for training or the best generalization performance attainable) as the computational task increases in size and complexity. Among the many possible ways of measuring the size or complexity of a computational task, the predicate order defined by Minsky and Papert (1969, 1988) provides the most useful and important measure.

To explain what we mean by a predicate, let $\psi(X)$ denote a function that can have only two values. Ordinarily we think of the two values of $\psi(X)$ as 0 and 1. But by taking the values to be FALSE or TRUE, we may think of $\psi(X)$ as a *predicate,* that is, a variable statement whose falsity or truth depends on the choice of argument X. For example, we may write

$$\psi_{\text{CIRCLE}}(X) = \begin{cases} 1 & \text{if the figure } X \text{ is a circle} \\ 0 & \text{if the figure } X \text{ is not a circle} \end{cases} \tag{4.115}$$

Using the idea of a predicate, Tesauro and Janssens (1988) performed an empirical study involving the use of a multilayer perceptron trained with the back-propagation algorithm to learn to compute the parity function. The *parity function* is a Boolean predicate defined by

$$\psi_{\text{PARITY}}(X) = \begin{cases} 1 & \text{if } |X| \text{ is an odd number} \\ 0 & \text{otherwise} \end{cases} \tag{4.116}$$

and whose order is equal to the number of inputs. The experiments performed by Tesauro and Janssens appear to show that the time required for the network to learn to compute the parity function scales exponentially with the number of inputs (i.e., the predicate order of the computation), and that projections of the use of the back-propagation algorithm to learn arbitrarily complicated functions may be overly optimistic.

It is generally agreed that it is inadvisable for a multilayer perceptron to be fully connected. In this context, we may therefore raise the following question: Given that a multilayer perceptron should not be fully connected, how should the synaptic connections of the network be allocated? This question is of no major concern in the case of small-scale applications, but it is certainly crucial to the successful application of back-propagation learning for solving large-scale, real-world problems.

One effective method of alleviating the scaling problem is to develop insight into the problem at hand (possibly through neurobiological analogy) and use it to put ingenuity into the architectural design of the multilayer perceptron. Specifically, the network architecture and the constraints imposed on synaptic weights of the network should be designed so as to incorporate prior information about the task into the makeup of the network. This design strategy is illustrated in Section 4.19 for the optical character recognition problem.

4.17 ACCELERATED CONVERGENCE OF BACK-PROPAGATION LEARNING

In the previous section we identified the main causes for the possible slow rate of convergence of the back-propagation algorithm. In this section we describe some *heuristics* that provide useful guidelines for thinking about how to accelerate the convergence of back-propagation learning through learning rate adaptation. Details of the heuristics are as follows (Jacobs, 1988):

HEURISTIC 1. Every adjustable network parameter of the cost function should have its own individual learning-rate parameter.

Here we note that the back-propagation algorithm may be slow to converge because the use of a fixed learning-rate parameter may not suit all portions of the error surface. In other words, a learning-rate parameter appropriate for the adjustment of one synaptic weight is not necessarily appropriate for the adjustment of other synaptic weights in the network. Heuristic 1 recognizes this fact by assigning a different learning-rate parameter to each adjustable synaptic weight (parameter) in the network.

HEURISTIC 2. Every learning-rate parameter should be allowed to vary from one iteration to the next.

The error surface typically behaves differently along different regions of a single weight dimension. In order to match this variation, heuristic 2 states that the learning-rate parameter needs to vary from iteration to iteration. It is interesting that this heuristic is well founded in the case of linear units (Luo, 1991).

HEURISTIC 3. When the derivative of the cost function with respect to a synaptic weight has the same algebraic sign for several consecutive iterations of the algorithm, the learning-rate parameter for that particular weight should be increased.

The current operating point in weight space may lie on a relatively flat portion of the error surface along a particular weight dimension. This may in turn account for the derivative of the cost function (i.e., the gradient of the error surface) with respect to that weight maintaining the same algebraic sign, and therefore pointing in the same direction, for several consecutive iterations of the algorithms. Heuristic 3 states that in

such a situation the number of iterations required to move across the flat portion of the error surface may be reduced by appropriately increasing the learning-rate parameter.

HEURISTIC 4. When the algebraic sign of the derivative of the cost function with respect to a particular synaptic weight alternates for several consecutive interations of the algorithm, the learning-rate parameter for that weight should be decreased.

When the current operating point in weight space lies on a portion of the error surface along a weight dimension of interest that exhibits peaks and valleys (i.e., the surface is highly curved), then it is possible for the derivative of the cost function with respect to that weight to change its algebraic sign from one iteration to the next. In order to prevent the weight adjustment from oscillating, heuristic 4 states that the learning-rate parameter for that particular weight should be decreased appropriately.

It is noteworthy that the use of a different and time-varying learning-rate parameter for each synaptic weight in accordance with these heuristics modifies the back-propagation algorithm in a fundamental way. Specifically, the modified algorithm no longer performs a steepest-descent search. Rather, the adjustments applied to the synaptic weights are based on (1) the partial derivatives of the error surface with respect to the weights, and (2) estimates of the curvatures of the error surface at the current operating point in weight space along the various weight dimensions.

Furthermore, all four heuristics satisfy the locality constraint, which is an inherent characteristic of back-propagation learning. Unfortunately, adherence to the locality constraint limits the domain of usefulness of these heuristics because error surfaces exist for which they do not work. Nevertheless, modifications of the back-propagation algorithm in accordance with these heuristics do have practical value.[14]

4.18 SUPERVISED LEARNING VIEWED AS AN OPTIMIZATION PROBLEM

In this section we take a viewpoint on supervised learning that is quite different from that pursued in previous sections of the chapter. Specifically, we view the supervised training of a multilayer perceptron as a problem in *numerical optimization*. In this context we first point out that the error surface of a multilayer perceptron with supervised learning is a highly nonlinear function of the synaptic weight vector $\mathbf{w}$. Let $\mathscr{E}_{av}(\mathbf{w})$ denote the cost function, averaged over the training sample. Using the Taylor series we may expand $\mathscr{E}_{av}(\mathbf{w})$ about the current point on the error surface $\mathbf{w}(n)$ for example, as described in Eq. (4.100), reproduced here with dependences on n included:

$$\mathscr{E}_{av}(\mathbf{w}(n) + \Delta\mathbf{w}(n)) = \mathscr{E}_{av}(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n) + \frac{1}{2}\Delta\mathbf{w}^T(n)\mathbf{H}(n)\Delta\mathbf{w}(n) + (\text{third- and higher-order terms}) \tag{4.117}$$

where $\mathbf{g}(n)$ is the local gradient vector defined by

$$\mathbf{g}(n) = \left.\frac{\partial\mathscr{E}_{av}(\mathbf{w})}{\partial\mathbf{w}}\right|_{\mathbf{w}=\mathbf{w}(n)} \tag{4.118}$$

and $\mathbf{H}(n)$ is the local Hessian matrix defined by

$$\mathbf{H}(n) = \left.\frac{\partial^2\mathscr{E}_{av}(\mathbf{w})}{\partial\mathbf{w}^2}\right|_{\mathbf{w}=\mathbf{w}(n)} \tag{4.119}$$

The use of an ensemble-averaged cost function $\mathscr{E}_{av}(\mathbf{w})$ presumes a batch mode of learning.

In the steepest-descent method, exemplified by the back-propagation algorithm, the adjustment $\Delta\mathbf{w}(n)$ applied to the synaptic weight vector $\mathbf{w}(n)$ is defined by

$$\Delta\mathbf{w}(n) = -\eta\mathbf{g}(n) \tag{4.120}$$

where η is the learning-rate parameter. In effect, the steepest-descent method operates on the basis of a *linear approximation* of the cost function in the local neighborhood of the operating point $\mathbf{w}(n)$. In so doing, it relies on the gradient vector $\mathbf{g}(n)$ as the only source of local information about the error surface. This restriction has a beneficial effect: simplicity of implementation. Unfortunately, it also has a detrimental effect: a slow rate of convergence, which can be excruciating, particularly in the case of large-scale problems. The inclusion of the momentum term in the update equation for the synaptic weight vector is a crude attempt at using second-order information about the error surface, which is of some help. However, its use makes the training process more delicate to manage by adding one more item to the list of parameters that have to be "tuned" by the designer.

In order to produce a significant improvement in the convergence performance of a multilayer perceptron (compared to back-propagation learning), we have to use *higher-order information* in the training process. We may do so by invoking a *quadratic approximation* of the error surface around the current point $\mathbf{w}(n)$. We then find from Eq. (4.117) that the optimum value of the adjustment $\Delta\mathbf{w}(n)$ applied to the synaptic weight vector $\mathbf{w}(n)$ is given by

$$\Delta\mathbf{w}^*(n) = \mathbf{H}^{-1}(n)\mathbf{g}(n) \tag{4.121}$$

where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian matrix $\mathbf{H}(n)$, assuming that it exists. Equation (4.121) is the essence of *Newton's method*. If the cost function $\mathscr{E}_{av}(\mathbf{w})$ is quadratic (i.e., the third- and higher-order terms in Eq. (4.117) are zero), Newton's method converges to the optimum solution in one iteration. However, the practical application of Newton's method to the supervised training of a multilayer perceptron is handicapped by the following factors:

- It requires calculation of the inverse Hessian matrix $\mathbf{H}^{-1}(n)$, which can be computationally expensive.
- For $\mathbf{H}^{-1}(n)$ to be computable, $\mathbf{H}(n)$ has to be nonsingular. In the case when $\mathbf{H}(n)$ is positive definite, the error surface around the current point $\mathbf{w}(n)$ is describable by a "convex bowl." Unfortunately, there is no guarantee that the Hessian matrix of the error surface of a multilayer perceptron will always fit this description. Moreover, there is the potential problem of the Hessian matrix being rank deficient (i.e., not all the columns of $\mathbf{H}$ being linearly independent), which results from the intrinsically ill-conditioned nature of neural network training problems (Saarinen et al., 1992); this only makes the computational task more difficult.

- When the cost function $\mathscr{E}_{av}(\mathbf{w})$ is nonquadratic, there is no guarantee for convergence of Newton's method, which makes it unsuitable for the training of a multilayer perceptron.

To overcome some of these difficulties, we may use a *quasi-Newton method*, which only requires an estimate of the gradient vector $\mathbf{g}$. This modification of Newton's method maintains a positive definite estimate of the inverse matrix $\mathbf{H}^{-1}$ directly without matrix inversion. By using such an estimate, a quasi-Newton method is assured of going downhill on the error surface. However, we still have a computational complexity that is $O(W^2)$, where W is the size of weight vector $\mathbf{w}$. Quasi-Newton methods are therefore computationally impractical, except for the training of very small-scale neural networks. A description of quasi-Newton methods is presented later in the section.

Another class of second-order optimization methods includes the conjugate-gradient method, which may be regarded as being somewhat intermediate between the method of steepest descent and Newton's method. Use of the conjugate-gradient method is motivated by the desire to accelerate the typically slow rate of convergence experienced with the method of steepest descent, while avoiding the computational requirements associated with the evaluation, storage, and inversion of the Hessian matrix in Newton's method. Among second-order optimization methods, it is widely acknowledged that the conjugate-gradient method is perhaps the only method that is applicable to large-scale problems, that is, problems with hundreds or thousands of adjustable parameters (Fletcher, 1987). It is therefore well suited for the training of multilayer perceptrons, with typical applications including function approximation, control, and time series analysis (i.e., regression).

Conjugate-Gradient Method

The conjugate-gradient method belongs to a class of second-order optimization methods known collectively as *conjugate-direction methods*. We begin the discussion of these methods by considering the minimization of the *quadratic function*

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} + c \tag{4.122}$$

where $\mathbf{x}$ is a W-by-1 parameter vector, $\mathbf{A}$ is a W-by-W symmetric, positive definite matrix, $\mathbf{b}$ is a W-by-1 vector, and c is a scalar. Minimization of the quadratic function $f(\mathbf{x})$ is achieved by assigning to $\mathbf{x}$ the unique value

$$\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b} \tag{4.123}$$

Thus minimizing $f(\mathbf{x})$ and solving the linear system of equations $\mathbf{A}\mathbf{x}^* = \mathbf{b}$ are equivalent problems.

Given the matrix $\mathbf{A}$, we say that a set of nonzero vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ is $\mathbf{A}$-*conjugate* (i.e., noninterfering with each other in the context of matrix $\mathbf{A}$) if the following condition is satisfied:

$$\mathbf{s}^T(n)\mathbf{A}\mathbf{s}(j) = 0 \qquad \text{for all } n \text{ and } j \text{ such that } n \neq j \tag{4.124}$$

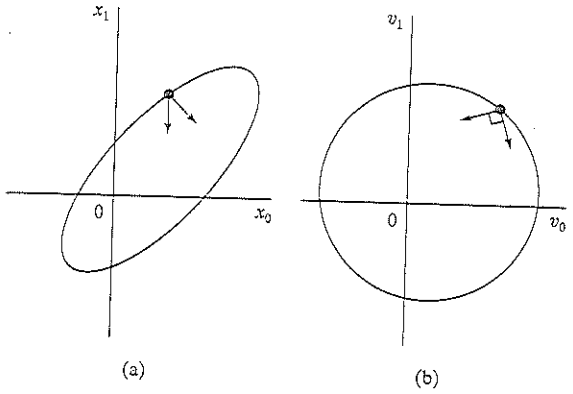

FIGURE 4.24 Interpretation of A-conjugate vectors. (a) Elliptic locus in two-dimesional weight space. (b) Transformation of the elliptic locus into a circular locus.

If $\mathbf{A}$ is equal to the identity matrix, conjugacy is equivalent to the usual notion of orthogonality.

Example 4.1

For an interpretation of A-conjugate vectors, consider the situation described in Fig. 4.24a, pertaining to a two-dimensional problem. The elliptic locus shown in this figure corresponds to a plot of Eq. (4.122) for

$$\mathbf{x} = [x_0, x_1]^T$$

at some constant value assigned to the quadratic function $f(\mathbf{x})$. Figure 4.24a also includes a pair of direction vectors that are conjugate with respect to the matrix $\mathbf{A}$. Suppose that we define a new parameter vector $\mathbf{v}$ related to $\mathbf{x}$ by the transformation

$$\mathbf{v} = \mathbf{A}^{1/2}\mathbf{x}$$

where $\mathbf{A}^{1/2}$ is the square root of $\mathbf{A}$. Then the elliptic locus of Fig. 4.24a is transformed into a circular locus, as shown in Fig. 4.24b. Correspondingly, the pair of $\mathbf{A}$-conjugate direction vectors in Fig. 4.24a is transformed into a pair of orthogonal direction vectors in Fig. 4.24b.

■

An important property of $\mathbf{A}$-conjugate vectors is that they are *linearly independent*. We prove this property by contradiction. Let one of these vectors, say $\mathbf{s}(0)$, be expressed as a linear combination of the remaining $W-1$ vectors as follows:

$$\mathbf{s}(0) = \sum_{j=1}^{W-1} \alpha_j \mathbf{s}(j)$$

Multiplying by $\mathbf{A}$ and then taking the inner product of $\mathbf{A}\mathbf{s}(0)$ with $\mathbf{s}(0)$ yields

$$\mathbf{s}^T(0)\mathbf{A}\mathbf{s}(0) = \sum_{j=1}^{W-1} \alpha_j \mathbf{s}^T(0)\mathbf{A}\mathbf{s}(j) = 0$$

However, it is impossible for the quadratic form $\mathbf{s}^T(0)\mathbf{A}\mathbf{s}(0)$ to be zero for two reasons: the matrix $\mathbf{A}$ is positive definite by assumption, and the vector $\mathbf{s}(0)$ is nonzero by definition. It

follows therefore that the A-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ cannot be linearly dependent; that is, they must be linearly independent.

For a given set of A-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$, the corresponding *conjugate direction method* for unconstrained minimization of the quadratic error function $f(\mathbf{x})$ is defined by (Luenberger, 1973; Fletcher, 1987; Bertsekas, 1995)

$$\mathbf{x}(n+1) = \mathbf{x}(n) + \eta(n)\mathbf{s}(n), \qquad n = 0, 1, \ldots, W-1 \tag{4.125}$$

where $\mathbf{x}(0)$ is an arbitrary starting vector, and $\eta(n)$ is a scalar defined by

$$f(\mathbf{x}(n) + \eta(n)\mathbf{s}(n)) = \min_{\eta} f(\mathbf{x}(n) + \eta\mathbf{s}(n)) \tag{4.126}$$

The procedure of choosing η so as to minimize the function $f(\mathbf{x}(n) + \eta\mathbf{s}(n))$ for some fixed n is referred to as a line search, which represents a one-dimensional minimization problem.

In light of Eqs. (4.124), (4.125) and (4.126), we now offer some observations:

1. Since the A-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ are linearly independent, they form a basis that spans the vector space of $\mathbf{w}$.
2. The update equation (4.125) and the line minimization of Eq. (4.126) lead to the same formula for the learning-rate parameter, namely,

$$\eta(n) = -\frac{\mathbf{s}^T(n)\mathbf{A}\boldsymbol{\epsilon}(n)}{\mathbf{s}^T(n)\mathbf{A}\mathbf{s}(n)}, \qquad n = 0, 1, \ldots, W-1 \tag{4.127}$$

   where $\boldsymbol{\epsilon}(n)$ is the *error vector* defined by

$$\boldsymbol{\epsilon}(n) = \mathbf{x}(n) - \mathbf{x}^* \tag{4.128}$$

3. Starting from an arbitrary point $\mathbf{x}(0)$, the conjugate direction method is guaranteed to find the optimum solution $\mathbf{x}^*$ of the quadratic equation $f(\mathbf{x}) = 0$ in at most W iterations.

The principal property of the conjugate-direction method is described as (Luenberger, 1984; Fletcher, 1987; Bertsekas, 1995):

> At successive iterations, the conjugate-direction method minimizes the quadratic function $f(\mathbf{x})$ over a progressively expanding linear vector space that eventually includes the global minimum of $f(\mathbf{x})$.

In particular, for each iteration n, the iterate $\mathbf{x}(n+1)$ minimizes the function $f(\mathbf{x})$ over a linear vector space $\mathcal{D}_n$ that passes through some arbitrary point $\mathbf{x}(0)$ and is spanned by the A-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(n)$, as shown by

$$\mathbf{x}(n+1) = \arg\min_{\mathbf{x} \in \mathcal{D}_n} f(\mathbf{x}) \tag{4.129}$$

where the space $\mathcal{D}_n$ is defined by

$$\mathcal{D}_n = \left\{ \mathbf{x}(n) \,|\, \mathbf{x}(n) = \mathbf{x}(0) + \sum_{j=0}^{n} \eta(j)\mathbf{s}(j) \right\} \tag{4.130}$$

For the conjugate-direction method to work, we require the availability of a set of A-conjugate vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$. In a special form of this method known as the *conjugate-gradient method*,[15] the successive direction vectors are generated as

A-conjugate versions of the successive gradient vectors of the quadratic function $f(\mathbf{x})$ as the method progresses, hence the name of the method. Thus, except for $n = 0$, the set of direction vectors $\{\mathbf{s}(n)\}$ is not specified beforehand, but rather it is determined in a sequential manner at successive steps of the method.

Define the *residual* as the steepest descent direction:

$$\mathbf{r}(n) = \mathbf{b} - \mathbf{A}\mathbf{x}(n) \tag{4.131}$$

Then to proceed, we use a linear combination of $\mathbf{r}(n)$ and $\mathbf{s}(n-1)$, as shown by

$$\mathbf{s}(n) = \mathbf{r}(n) + \beta(n)\mathbf{s}(n-1), \qquad n = 1, 2, \ldots, W-1 \tag{4.132}$$

where $\beta(n)$ is a scaling factor to be determined. Multiplying this equation by $\mathbf{A}$, taking the inner product of the resulting expression with $\mathbf{s}(n-1)$, invoking the A-conjugate property of the direction vectors, and then solving the resulting expression for $\beta(n)$, we get

$$\beta(n) = -\frac{\mathbf{s}^T(n-1)\mathbf{A}\mathbf{r}(n)}{\mathbf{s}^T(n-1)\mathbf{A}\mathbf{s}(n-1)} \tag{4.133}$$

Using Eqs. (4.132) and (4.133), we find that the vectors $\mathbf{s}(0), \mathbf{s}(1), \ldots, \mathbf{s}(W-1)$ so generated are indeed A-conjugate.

Generation of the direction vectors in accordance with the recursive equation (4.132) depends on the coefficient $\beta(n)$. The formula of Eq. (4.133) for evaluating $\beta(n)$, as it presently stands, requires knowledge of matrix $\mathbf{A}$. For computational reasons, it would be desirable to evaluate $\beta(n)$ without explicit knowledge of $\mathbf{A}$. This evaluation can be achieved by using one of two formulas (Fletcher, 1987):

1. *Polak–Ribière formula*, for which $\beta(n)$ is defined by

$$\beta(n) = \frac{\mathbf{r}^T(n)(\mathbf{r}(n) - \mathbf{r}(n-1))}{\mathbf{r}^T(n-1)\mathbf{r}(n-1)} \tag{4.134}$$

2. *Fletcher–Reeves formula*, for which $\beta(n)$ is defined by

$$\beta(n) = \frac{\mathbf{r}^T(n)\mathbf{r}(n)}{\mathbf{r}^T(n-1)\mathbf{r}(n-1)} \tag{4.135}$$

To use the conjugate-gradient method to attack the unconstrained minimization of the cost function $\mathscr{E}_{av}(\mathbf{w})$ pertaining to the unsupervised training of multilayer perceptrons, we do two things:

- Approximate the cost function $\mathscr{E}_{av}(\mathbf{w})$ by a quadratic function. That is, the third- and higher-order terms in Eq. (4.117) are ignored, which means that we are operating close to a local minimum on the error surface. On this basis, comparing Eqs. (4.117) and (4.122), we can make the associations indicated in Table 4.7.
- Formulate the computation of coefficients $\beta(n)$ and $\eta(n)$ in the conjugate-gradient algorithm so as to only require gradient information.

The latter point is particularly important in the context of multilayer perceptrons because it avoids using the Hessian matrix $\mathbf{H}(n)$, the evaluation of which is plagued with computational difficulties.

TABLE 4.7 Correspondence Between $f(\mathbf{x})$ and $\mathscr{E}_{av}(\mathbf{w})$

Quadratic function $f(\mathbf{x})$	Cost function $\mathscr{E}_{av}(\mathbf{w})$
Parameter vector $\mathbf{x}(n)$	Synaptic weight vector $\mathbf{w}(n)$
Gradient vector $\partial f(\mathbf{x})/\partial \mathbf{x}$	Gradient vector $\mathbf{g} = \partial \mathscr{E}_{av}/\partial \mathbf{w}$
Matrix $\mathbf{A}$	Hessian matrix $\mathbf{H}$

To compute the coefficient $\beta(n)$ that determines the search direction $\mathbf{s}(n)$ without explicit knowledge of the Hessian matrix $\mathbf{H}(n)$, we can use the Polak–Ribiére formula of Eq. (4.134) or the Fletcher–Reeves formula of Eq. (4.135). Both of these formulas involve the use of residuals only. In the linear form of the conjugate-gradient method, assuming a quadratic function, the Polak–Ribiére and Fletcher–Reeves formulas are equivalent. On the other hand, in the case of a nonquadratic cost function, they are no longer equivalent.

For nonquadratic optimization problems, the Polak–Ribiére form of the conjugate-gradient algorithm is typically superior to the Fletcher–Reeves form of the algorithm, for which we offer the following heuristic explanation (Bertsekas, 1995). Due to the presence of third- and higher-order terms in the cost function $\mathscr{E}_{av}(\mathbf{w})$ and possible inaccuracies in the line search, conjugacy of the generated search directions is progressively lost. This may in turn cause the algorithm to "jam" in the sense that the generated direction vector $\mathbf{s}(n)$ is nearly orthogonal to the residual $\mathbf{r}(n)$. When this phenomenon occurs, we have $\mathbf{r}(n) = \mathbf{r}(n-1)$, in which case the scalar $\beta(n)$ will be nearly zero. Correspondingly, the direction vector $\mathbf{s}(n)$ will be close to $\mathbf{r}(n)$, thereby breaking the jam. In contrast, when the Fletcher–Reeves formula is used, the conjugate gradient algorithm typically continues to jam under similar conditions.

In rare cases, however, the Polak–Ribiére method can cycle indefinitely without converging. Fortunately, convergence of the Polak–Ribiére method can be guaranteed by choosing (Shewchuk, 1994)

$$\beta = \max\{\beta_{PR}, 0\} \tag{4.136}$$

where β_{PR} is the value defined by the Polak–Ribiére formula of Eq. (4.134). Using the value of β defined in Eq. (4.136) is equivalent to restarting the conjugate gradient algorithm if $\beta_{PR} < 0$. To restart the algorithm is equivalent to forgetting the last search direction and starting it anew in the direction of steepest descent (Shewchuk, 1994).

Consider next the issue of computing the parameter $\eta(n)$, which determines the learning rate of the conjugate-gradient algorithm. As with $\beta(n)$, the preferred method for computing $\eta(n)$ is one that avoids having to use the Hessian matrix $\mathbf{H}(n)$. We recall that the line minimization based on Eq. (4.126) leads to the same formula for $\eta(n)$ as that derived from the update equation (4.125). We therefore need a *line search*,[16] the purpose of which is to minimize the function $\mathscr{E}_{av}(\mathbf{w} + \eta\mathbf{s})$ with respect to η. That is, given fixed values of the vectors $\mathbf{w}$ and $\mathbf{s}$, the problem is to vary η such that this function is minimized. As η varies, the argument $\mathbf{w} + \eta\mathbf{s}$ traces a line in the W-dimensional vector space of $\mathbf{w}$, hence the name "line search." A *line search algorithm* is an iterative procedure that generates a sequence of estimates $\{\eta(n)\}$ for each iteration of the conjugate-gradient algorithm. The line search is terminated when a satisfactory solution is found. A line search must be performed along each search direction.

Several line search algorithms have been proposed in the literature, and a good choice is important because it has a profound impact on the performance of the conjugate-gradient algorithm in which it is embedded. There are two phases to any line search algorithm (Fletcher, 1987):

- *Bracketing phase*, which searches for a *bracket*, that is, a nontrivial interval that is known to contain a minimum.
- *Sectioning phase*, in which the bracket is *sectioned* (i.e., divided), thereby generating a sequence of brackets whose length is progressively reduced.

We now describe a *curve-fitting procedure* that takes care of these two phases in a straightforward manner.

Let $\mathscr{E}_{av}(\eta)$ denote the cost function of the multilayer perceptron, expressed as a function of η. It is assumed that $\mathscr{E}_{av}(\eta)$ is strictly *unimodal* (i.e., it has a single minimum in the neighborhood of the current point $\mathbf{w}(n)$) and is twice continuously differentiable. We initiate the search procedure by searching along the line until we find three points η_1, η_2, and η_3 such that the following condition is satisfied:

$$\mathscr{E}_{av}(\eta_1) \geq \mathscr{E}_{av}(\eta_3) \geq \mathscr{E}_{av}(\eta_2) \qquad \text{for } \eta_1 < \eta_2 < \eta_3 \tag{4.137}$$

as illustrated in Fig. 4.25. Since $\mathscr{E}_{av}(\eta)$ is a continuous function of η, the choice described in Eq. (4.137) ensures that the bracket $[\eta_1, \eta_3]$ contains a minimum of the function $\mathscr{E}_{av}(\eta)$. Provided that the function $\mathscr{E}_{av}(\eta)$ is sufficiently smooth, we may consider this function to be parabolic in the immediate neighborhood of the minimum. Accordingly, we may use *inverse parabolic interpolation* to do the sectioning (Press et al., 1988). Specifically, a parabolic function is fitted through the three original points η_1, η_2, and η_3, as illustrated in Fig. 4.26, where the solid line corresponds to $\mathscr{E}_{av}(\eta)$ and the dashed line corresponds to the first iteration of the sectioning procedure. Let the minimum of the parabola passing through the three points η_1, η_2, and η_3 be denoted by η_4. In the example illustrated in Fig. 4.26, we have $\mathscr{E}_{av}(\eta_4) < \mathscr{E}_{av}(\eta_2)$ and $\mathscr{E}_{av}(\eta_4) < \mathscr{E}_{av}(\eta_1)$. Point η_3 is replaced in favor of η_4, making $[\eta_1, \eta_4]$ the new bracket. The process is repeated by constructing a new parabola through the points η_1, η_2, and η_4. The bracketing-followed-by-sectioning procedure, as illustrated, is repeated several times until a point close enough to the minimum of $\mathscr{E}_{av}(\eta)$ is located, at which time the line search is terminated.

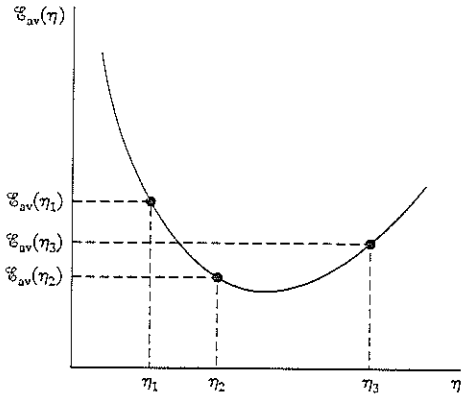

FIGURE 4.25 Illustration of the line search.

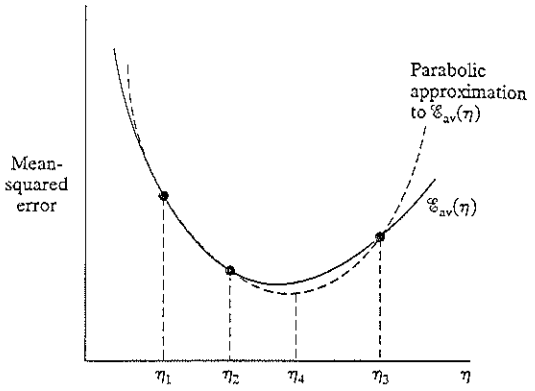

FIGURE 4.26 Inverse parabolic interpolation.

Brent's method constitutes a highly refined version of the three-point curve-fitting procedure just described (Press et al., 1988). At any particular stage of the computation, Brent's method keeps track of six points on the function $\mathscr{E}_{av}(\eta)$, which may not all be necessarily distinct. As before, parabolic interpolation is attempted through three of these points. For the interpolation to be acceptable, certain criteria involving the remaining three points must be satisfied. The net result is a robust line search algorithm.

Summary of the Nonlinear Conjugate Gradient Algorithm

All the ingredients we need to formally describe the nonlinear (nonquadratic) form of the conjugate-gradient algorithm for the supervised training of a multilayer perceptron are now in place. A summary of the algorithm is presented in Table 4.8.

Quasi-Newton Methods

Resuming the discussion on quasi-Newton methods, we find that these are basically gradient methods described by the update equation:

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n) \tag{4.138}$$

where the direction vector $\mathbf{s}(n)$ is defined in terms of the gradient vector $\mathbf{g}(n)$ by

$$\mathbf{s}(n) = -\mathbf{S}(n)\mathbf{g}(n) \tag{4.139}$$

The matrix $\mathbf{S}(n)$ is a positive definite matrix that is adjusted from one iteration to the next. This is done in order to make the direction vector $\mathbf{s}(n)$ approximate the *Newton direction*, namely

$$-(\partial^2 \mathscr{E}_{av}/\partial \mathbf{w}^2)^{-1}\,(\partial \mathscr{E}_{av}/\partial \mathbf{w})$$

Quasi-Newton methods use second-order (curvature) information about the error surface without actually requiring knowledge of the Hessian matrix $\mathbf{H}$. They do so by using two successive iterates $\mathbf{w}(n)$ and $\mathbf{w}(n+1)$, together with the respective gradient vectors $\mathbf{g}(n)$ and $\mathbf{g}(n+1)$. Let

$$\mathbf{q}(n) = \mathbf{g}(n+1) - \mathbf{g}(n) \tag{4.140}$$

and

$$\Delta\mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) \tag{4.141}$$

We may then derive curvature information by using the approximate formula:

$$\mathbf{q}(n) \approx \left(\frac{\partial}{\partial \mathbf{w}}\,\mathbf{g}(n)\right)\Delta\mathbf{w}(n) \tag{4.142}$$

In particular, given W linearly independent weight increments $\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), \ldots, \Delta\mathbf{w}(W-1)$ and the respective gradient increments $\mathbf{q}(0), \mathbf{q}(1), \ldots, \mathbf{q}(W-1)$, we may approximate the Hessian matrix $\mathbf{H}$ as:

$$\mathbf{H} \approx [\mathbf{q}(0), \mathbf{q}(1), \ldots, \mathbf{q}(W-1)]\,[\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), \ldots, \Delta\mathbf{w}(W-1)]^{-1} \tag{4.143}$$

TABLE 4.8 Summary of the Nonlinear Conjugate Gradient Algorithm for the Supervised Training of a Multilayer Perceptron

Initialization

Unless prior knowledge on the weight vector $\mathbf{w}$ is available, choose the initial value $\mathbf{w}(0)$ using a procedure similar to that described for the back-propagation algorithm.

Computation

1. For $\mathbf{w}(0)$, use back-propagation to compute the gradient vector $\mathbf{g}(0)$.
2. Set $\mathbf{s}(0) = \mathbf{r}(0) = -\mathbf{g}(0)$.
3. At time step n, use a line search to find $\eta(n)$ that minimizes $\mathscr{E}_{av}(\eta)$ sufficiently, representing the cost function $\mathscr{E}_{av}$ expressed as a function of η for fixed values of $\mathbf{w}$ and $\mathbf{s}$.
4. Test to determine if the Euclidean norm of the residual $\mathbf{r}(n)$ has fallen below a specified value, that is, a small fraction of the initial value $\|\mathbf{r}(0)\|$.
5. Update the weight vector:
$$\mathbf{w}(n+1) = \mathbf{w}(n) + \eta(n)\mathbf{s}(n)$$
6. For $\mathbf{w}(n+1)$, use back-propagation to compute the updated gradient vector $\mathbf{g}(n+1)$.
7. Set $\mathbf{r}(n+1) = -\mathbf{g}(n+1)$.
8. Use the Polak–Ribiére method to calculate $\beta(n+1)$:
$$\beta(n+1) = \max\left\{\frac{\mathbf{r}^T(n+1)(\mathbf{r}(n+1) - \mathbf{r}(n))}{\mathbf{r}^T(n)\mathbf{r}(n)}, 0\right\}$$
9. Update the direction vector:
$$\mathbf{s}(n+1) = \mathbf{r}(n+1) + \beta(n+1)\mathbf{s}(n)$$
10. Set $n = n+1$, and go back to step 3.

Stopping criterion. Terminate the algorithm when the following condition is satisfied:

$$\|\mathbf{r}(n)\| \leq \epsilon\|\mathbf{r}(0)\|$$

where ϵ is a prescribed small number.

We may also approximate the inverse Hessian matrix as:

$$\mathbf{H}^{-1} \approx [\Delta\mathbf{w}(0), \Delta\mathbf{w}(1), \ldots, \Delta\mathbf{w}(W-1)]\,[\mathbf{q}(0), \mathbf{q}(1), \ldots, \mathbf{q}(W-1)]^{-1} \quad (4.144)$$

When the cost function $\mathscr{E}_{av}(\mathbf{w})$ is quadratic, Eqs. (4.143) and (4.144) are exact.

In the most popular class of quasi-Newton methods, the matrix $\mathbf{S}(n+1)$ is obtained from its previous value $\mathbf{S}(n)$, the vectors $\Delta\mathbf{w}(n)$ and $\mathbf{q}(n)$, by using the recursion (Fletcher, 1987; Bertsekas, 1995):

$$\mathbf{S}(n+1) = \mathbf{S}(n) + \frac{\Delta\mathbf{w}(n)\Delta\mathbf{w}^T(n)}{\mathbf{q}^T(n)\mathbf{q}(n)} - \frac{\mathbf{S}(n)\mathbf{q}(n)\mathbf{q}^T(n)\mathbf{S}(n)}{\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)} + \xi(n)[\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)]\,[\mathbf{v}(n)\mathbf{v}^T(n)] \quad (4.145)$$

where

$$\mathbf{v}(n) = \frac{\Delta\mathbf{w}(n)}{\Delta\mathbf{w}^T(n)\Delta\mathbf{w}(n)} - \frac{\mathbf{S}(n)\mathbf{q}(n)}{\mathbf{q}^T(n)\mathbf{S}(n)\mathbf{q}(n)} \quad (4.146)$$

and

$$0 \le \xi(n) \le 1 \qquad \text{for all } n \quad (4.147)$$

The algorithm is initiated with some arbitrary positive definite matrix $\mathbf{S}(0)$. The particular form of the quasi-Newton method is parameterized by how the scalar $\eta(n)$ is defined, as indicated here (Fletcher, 1987):

- For $\xi(n) = 0$ for all n, we obtain the *Davidon–Fletcher–Powell (DFP) algorithm*, which is historically the first quasi-Newton method.
- For $\xi(n) = 1$ for all n, we obtain the *Broyden–Fletcher–Goldfarb–Shanno algorithm*, which is considered to be the best form of quasi-Newton methods currently known.

Comparison of Quasi-Newton Methods with Conjugate-Gradient Methods

We conclude this brief discussion of quasi-Newton methods by comparing them with conjugate-gradient methods in the context of nonquadratic optimization problems (Bertsekas, 1995):

- Both quasi-Newton and conjugate-gradient methods avoid the need for using the Hessian matrix. However, quasi-Newton methods go one step further by generating an approximation to the inverse Hessian matrix. Accordingly, when the line search is accurate and we are in close proximity to a local minimum with a positive definite Hessian, a quasi-Newton method tends to approximate Newton's method, thereby attaining faster convergence than would be possible with the conjugate-gradient method.
- Quasi-Newton methods are not as sensitive to accuracy in the line search stage of the optimization as the conjugate-gradient method.
- Quasi-Newton methods require storage of the matrix $\mathbf{S}(n)$, in addition to the matrix-vector multiplication overhead associated with the computation of the director vector $\mathbf{s}(n)$. The net result is that the computational complexity of quasi-Newton's methods is $O(W^2)$. In contrast, the computational complexity of the conjugate-gradient method is $O(W)$. Thus, when the dimension W (i.e., size of the weight vector $\mathbf{w}$) is large, conjugate-gradient methods are preferable to quasi-Newton methods in computational terms.

It is because of this latter point that the use of quasi-Newton methods is restricted, in practice, to the design of small-scale neural networks.

4.19 CONVOLUTIONAL NETWORKS

Up to this point, we have been concerned with the algorithmic design of multilayer perceptrons and related issues. In this section we focus on the structural layout of the multilayer perceptron itself. In particular, we describe a special class of multilayer perceptrons known collectively as *convolutional networks;* the idea behind these networks was briefly highlighted in Chapter 1.

A *convolutional network* is a multilayer perceptron designed specifically to recognize two-dimensional shapes with a high degree of invariance to translation, scaling, skewing, and other forms of distortion. This difficult task is learned in a supervised manner by means of a network whose structure includes the following forms of *constraints* (LeCun and Bengio, 1995):

1. *Feature extraction.* Each neuron takes its synaptic inputs from a local *receptive field* in the previous layer, thereby forcing it to extract local features. Once a feature has been extracted, its exact location becomes less important so long as its position relative to other features is approximately preserved.
2. *Feature mapping.* Each computational layer of the network is composed of multiple *feature maps*, with each feature map being in the form of a plane within which the individual neurons are *constrained* to share the same set of synaptic weights. This second form of structural constraint has the following beneficial effects:
   - *Shift invariance*, forced into the operation of a feature map through the use of *convolution* with a kernel of small size, followed by a sigmoid (squashing) function.
   - *Reduction in the number of free parameters*, accomplished through the use of *weight sharing*.
3. *Subsampling.* Each convolutional layer is followed by a computational layer that performs *local averaging* and *subsampling*, whereby the resolution of the feature map is reduced. This operation has the effect of reducing the sensitivity of the feature map's output to shifts and other forms of distortion.

The development of convolutional networks, as described, is neurobiologically motivated, which goes back to the pioneering work of Hubel and Wiesel (1962, 1977) on locally sensitive and orientation-selective neurons in the visual cortex of a cat.

We emphasize that all weights in all layers of a convolutional network are learned through training. Moreover, the network learns to extract its own features automatically.