7

PARAMETER OPTIMIZATION ALGORITHMS

In previous chapters, the problem of learning in neural networks has been formulated in terms of the minimization of an error function E. This error is a function of the adaptive parameters (weights and biases) in the network, which we can conveniently group together into a single W-dimensional weight vector $\mathbf{w}$ with components $w_1 \ldots w_W$.

In Chapter 4 it was shown that, for a multi-layer perceptron, the derivatives of an error function with respect to the network parameters can be obtained in a computationally efficient way using back-propagation. We shall see that the use of such gradient information is of central importance in finding algorithms for network training which are sufficiently fast to be of practical use for large-scale applications.

The problem of minimizing continuous, differentiable functions of many variables is one which has been widely studied, and many of the conventional approaches to this problem are directly applicable to the training of neural networks. In this chapter we shall review several of the most important practical algorithms. One of the simplest of these is gradient descent, which has been described briefly in earlier chapters. Here we investigate gradient descent in more detail, and discuss its limitations. We then describe a number of heuristic modifications to gradient descent which aim to improve its performance. Next we review an important class of conventional optimization algorithms based on the concept of conjugate gradients, including a relatively recent variation called scaled conjugate gradients. We then describe the other major class of conventional optimization algorithms known as quasi-Newton methods. Finally, we discuss the powerful Levenberg–Marquardt algorithm which is applicable specifically to a sum-of-squares error function. There are many standard textbooks which cover non-linear optimization techniques, including Polak (1971), Gill *et al.* (1981), Dennis and Schnabel (1983), Luenberger (1984), and Fletcher (1987).

It is sometimes argued that learning algorithms for neural networks should be local (in the sense of the network diagram) so that the computations needed to update each weight can be performed using information available locally to that weight. This requirement may be motivated by interest in modelling biological neural systems or by the desire to implement network algorithms in parallel hardware. Although the locality issue is relevant both to biological plausibility and to hardware implementation, it represents only one facet of these issues, and much more careful analyses are required. Since our goal is to find the most

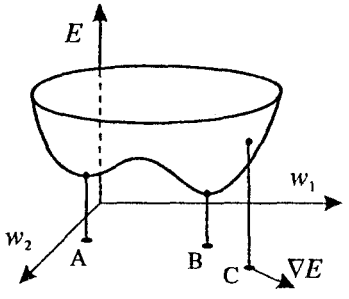

Figure 7.1. Geometrical picture of the error function $E(\mathbf{w})$ as a surface sitting above weight space. Points A and B represent minima of the error function. At any point C, the local gradient of the error surface is given by the vector ∇E.

effective techniques for pattern recognition, there is little point in introducing unnecessary restrictions. We shall therefore regard the issue of locality as irrelevant in the present context.

Most of the algorithms which are described in this chapter are ones which have been found to have good performance in a wide range of applications. However, different algorithms will perform best on different problems and it is therefore not possible to recommend a single universal optimization algorithm. Instead, we highlight the relative advantages and limitations of different algorithms as they are discussed.

7.1 Error surfaces

The problem addressed in this chapter is to find a weight vector $\mathbf{w}$ which minimizes an error function $E(\mathbf{w})$. It is useful to have a simple geometrical picture of the error minimization process, which can be obtained by viewing $E(\mathbf{w})$ as an *error surface* sitting above *weight space*, as shown in Figure 7.1. For networks having a single layer of weights, linear output-unit activation functions, and a sum-of-squares error, the error function will be a quadratic function of the weights. In this case the error surface will have a general multidimensional parabolic form. There is then a single minimum (or possibly a single continuum of degenerate minima), which can be located by solution of a set of coupled linear equations, as discussed in detail in Section 3.4.3.

However, for more general networks, in particular those with more than one layer of adaptive weights, the error function will typically be a highly non-linear function of the weights, and there may exist many minima all of which satisfy

$$\nabla E = 0 \tag{7.1}$$

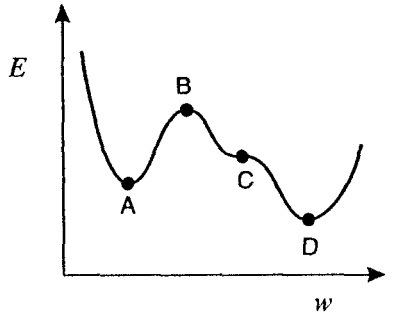

Figure 7.2. A schematic error function for a single parameter w, showing four stationary points at which the local gradient of the error function vanishes. Point A is a local minimum, point B is a local maximum, point C is a saddle-point, and point D is the global minimum.

where ∇E denotes the gradient of E in weight space. The minimum for which the value of the error function is smallest is called the *global minimum* while other minima are called *local minima*. There may also be other points which satisfy the condition (7.1) such as local maxima or saddlepoints. Any vector $\mathbf{w}$ for which this condition is satisfied is called a *stationary point*, and the different kinds of stationary point are illustrated schematically in Figure 7.2.

As a consequence of the non-linearity of the error function, it is not in general possible to find closed-form solutions for the minima. Instead, we consider algorithms which involve a search through weight space consisting of a succession of steps of the form

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta\mathbf{w}^{(\tau)} \tag{7.2}$$

where τ labels the iteration step. Different algorithms involve different choices for the weight vector increment $\Delta\mathbf{w}^{(\tau)}$. For some algorithms, such as conjugate gradients and the quasi-Newton algorithms discussed later, the error function is guaranteed not to increase as a result of a change to the weights (and hopefully will decrease). One potential disadvantage of such algorithms is that if they reach a local minimum they will remain there forever, as there is no mechanism for them to escape (as this would require a temporary increase in the error function). The choice of initial weights for the algorithm then determines which minimum the algorithm will converge to. Also, the presence of saddlepoints, or regions where the error function is very flat, can cause some iterative algorithms to become 'stuck' for extensive periods of time, thereby mimicking local minima.

Different algorithms can exhibit different behaviour in the neighbourhood of a minimum. If $\epsilon^{(\tau)}$ denotes the distance to the minimum at step τ, then convergence often has the general form

$$\epsilon^{(\tau+1)} \propto (\epsilon^{(\tau)})^L \tag{7.3}$$

where L governs the *order of convergence*. Values of $L = 1$ and $L = 2$ are known respectively as linear and quadratic convergence.

In Section 4.4 we discussed the high degree of symmetry which exists in the weight space of a multi-layered neural network. For instance, a two-layer network with M hidden units exhibits a symmetry factor of $M!2^M$. Thus, for any point in weight space, there will be $M!2^M$ equivalent points which generate the same network mapping, and which therefore give rise to the same value for the error function. Any local or global minimum will therefore be replicated a large number of times throughout weight space. Of course, in a practical application it is irrelevant which of these many equivalent solutions we use. Furthermore, the algorithms we shall be discussing make use of a local stepwise search through weight space, and will be completely unaffected by the presence of the numerous equivalent points elsewhere in weight space.

In Section 6.1.3 we showed that the sum-of-squares error function, in the limit of an infinite data set, can be written as the sum of two terms

$$E = \frac{1}{2}\sum_k \int \{y_k(\mathbf{x};\mathbf{w}) - \langle t_k|\mathbf{x}\rangle\}^2 p(\mathbf{x})\,d\mathbf{x} + \frac{1}{2}\sum_k \int \{\langle t_k^2|\mathbf{x}\rangle - \langle t_k|\mathbf{x}\rangle^2\} p(\mathbf{x})\,d\mathbf{x} \tag{7.4}$$

where $y_k(\mathbf{x};\mathbf{w})$ denotes the activation of output unit k when the network is presented with input vector $\mathbf{x}$, and $\langle t_k|\mathbf{x}\rangle$ denotes the conditional average of the corresponding target variable given by

$$\langle t_k|\mathbf{x}\rangle = \int t_k p(t_k|\mathbf{x})\,dt_k. \tag{7.5}$$

Since only the first term in (7.4) depends on the network weights, the global minimum of the error is obtained when $y_k(\mathbf{x};\mathbf{w}) = \langle t_k|\mathbf{x}\rangle$. This can be regarded as the optimal solution, as discussed in Section 6.1.3. In practice we must deal with finite data sets, however. If the network is relatively complex (for instance if it has a large number of adaptive parameters) then the best generalization performance might be obtained from a local minimum, or from some other point in weight space which is not a minimum of the error. This leads to a consideration of techniques in which the generalization performance is monitored as a function of time during the training, and the training is halted when the optimum generalization is achieved. Such methods are discussed briefly in Section 9.2.4.

7.2 Local quadratic approximation

A considerable degree of insight into the optimization problem, and into the various techniques for solving it, can be obtained by considering a local quadratic approximation to the error function. Consider the Taylor expansion of $E(\mathbf{w})$ around some point $\widehat{\mathbf{w}}$ in weight space

$$E(\mathbf{w}) = E(\widehat{\mathbf{w}}) + (\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{b} + \frac{1}{2}(\mathbf{w} - \widehat{\mathbf{w}})^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}) \tag{7.6}$$

where $\mathbf{b}$ is defined to be the gradient of E evaluated at $\widehat{\mathbf{w}}$

$$\mathbf{b} \equiv \nabla E|_{\widehat{\mathbf{w}}} \tag{7.7}$$

and the Hessian matrix $\mathbf{H}$ is defined by

$$(\mathbf{H})_{ij} \equiv \left.\frac{\partial E}{\partial w_i \partial w_j}\right|_{\widehat{\mathbf{w}}}. \tag{7.8}$$

From (7.6), the corresponding local approximation for the gradient is given by

$$\nabla E = \mathbf{b} + \mathbf{H}(\mathbf{w} - \widehat{\mathbf{w}}). \tag{7.9}$$

For points $\mathbf{w}$ which are close to $\widehat{\mathbf{w}}$, these expressions will give reasonable approximations for the error and its gradient, and they form the basis for much of the subsequent discussion of optimization algorithms.

Consider the particular case of a local quadratic approximation around a point $\mathbf{w}^*$ which is a minimum of the error function. In this case there is no linear term, since $\nabla E = 0$ at $\mathbf{w}^*$, and (7.6) becomes

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}(\mathbf{w} - \mathbf{w}^*)^{\mathrm{T}}\mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{7.10}$$

where the Hessian is evaluated at $\mathbf{w}^*$. In order to interpret this geometrically, consider the eigenvalue equation for the Hessian matrix

$$\mathbf{H}\mathbf{u}_i = \lambda_i \mathbf{u}_i \tag{7.11}$$

where the eigenvectors $\mathbf{u}_i$ form a complete orthonormal set (Appendix A) so that

$$\mathbf{u}_i^{\mathrm{T}}\mathbf{u}_j = \delta_{ij}. \tag{7.12}$$

We now expand $(\mathbf{w} - \mathbf{w}^*)$ as a linear combination of the eigenvectors in the form

$$\mathbf{w} - \mathbf{w}^* = \sum_i \alpha_i \mathbf{u}_i. \tag{7.13}$$

Substituting (7.13) into (7.10), and using (7.11) and (7.12), allows the error function to be written in the form

$$E(\mathbf{w}) = E(\mathbf{w}^*) + \frac{1}{2}\sum_i \lambda_i \alpha_i^2. \tag{7.14}$$

Equation (7.13) can be regarded as a transformation of the coordinate system in which the origin is translated to the point $\mathbf{w}^*$, and the axes are rotated to align with the eigenvectors (through the orthogonal matrix whose columns are the $\mathbf{u}_i$). This transformation is discussed in more detail in Appendix A.

A matrix $\mathbf{H}$ is said to be *positive definite* if

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} > 0 \qquad \text{for all } \mathbf{v}. \tag{7.15}$$

Since the eigenvectors $\{\mathbf{u}_i\}$ form a complete set, an arbitrary vector $\mathbf{v}$ can be written

$$\mathbf{v} = \sum_i \beta_i \mathbf{u}_i \tag{7.16}$$

From (7.11) and (7.12) we then have

$$\mathbf{v}^{\mathrm{T}}\mathbf{H}\mathbf{v} = \sum_i \beta_i^2 \lambda_i \tag{7.17}$$

and so $\mathbf{H}$ will be positive definite if all of its eigenvalues are positive. In the new coordinate system whose basis vectors are given by the eigenvectors $\{\mathbf{u}_i\}$, the contours of constant E are ellipses centred on the origin, whose axes are aligned with the eigenvectors and whose lengths are inversely proportional to the square roots of the eigenvalues, as indicated in Figure 7.3. For a one-dimensional weight space, a stationary point w^* will be a minimum if

$$\partial E/\partial w|_{w^*} > 0. \tag{7.18}$$

The corresponding result in d-dimensions is that the Hessian matrix, evaluated at $\mathbf{w}^*$, should be positive definite (Exercise 7.1).

7.2.1 *Use of gradient information*

For most of the network models and error functions which are discussed in earlier chapters, it is possible to evaluate the gradient of the error function relatively efficiently, for instance by means of the back-propagation procedure. The use of

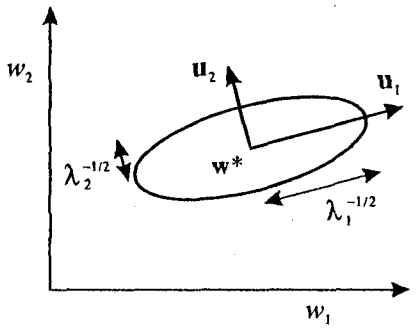

Figure 7.3. In the neighbourhood of a minimum $\mathbf{w}^*$, the error function can be approximated by a quadratic function. Contours of constant error are then ellipses whose axes are aligned with the eigenvectors $\mathbf{u}_i$ of the Hessian matrix, with lengths that are inversely proportional to the square roots of the corresponding eigenvectors λ_i.

this gradient information can lead to significant improvements in the speed with which the minima of the error function can be located. We can easily see why this is so, as follows.

In the quadratic approximation to the error function, given in (7.6), the error surface is specified by the quantities $\mathbf{b}$ and $\mathbf{H}$, which contain a total of $W(W+3)/2$ independent terms (since the matrix $\mathbf{H}$ is symmetric), where W is the dimensionality of $\mathbf{w}$ (i.e. the total number of adaptive parameters in the network). The location of the minimum of this quadratic approximation therefore depends on $O(W^2)$ parameters, and we should not expect to be able to locate the minimum until we have gathered $O(W^2)$ independent pieces of information. If we do not make use of gradient information, we would expect to have to perform at least $O(W^2)$ function evaluations, each of which would require $O(W)$ steps. Thus, the computational effort needed to find the minimum would scale like $O(W^3)$.

Now compare this with an algorithm which makes use of the gradient information. Since each evaluation of ∇E brings W items of information, we might hope to find the minimum of the function in $O(W)$ gradient evaluations. Using back-propagation, each such evaluation takes only $O(W)$ steps and so the minimum could now be found in $O(W^2)$ steps. This dramatically improved scaling with W strongly suggests that gradient information should be exploited, as is the case for the optimization algorithms discussed in this chapter.

7.3 Linear output units

As discussed at length in Section 3.4.3, if a sum-of-squares error function is used, and the network mapping depends linearly on the weights, then the minimization

of the error function represents a linear problem, which can be solved exactly in a single step using singular value decomposition (SVD). If we consider a more general multi-layer network with linear output units, then the dependence of the network mapping on the final-layer weights will again be linear. This means that the partial optimization of a sum-of-squares error function with respect to these weights (with all other parameters held fixed) can again be performed by linear methods, as discussed in Section 3.4.3. The computational effort involved in SVD is often very much less than that required for general non-linear optimization, which suggests that it may be worthwhile to use linear methods for the final-layer weights, and non-linear methods for all other parameters. This leads to the following hybrid procedure for optimizing the weights in such networks (Webb and Lowe, 1988).

Suppose the final-layer weights are collected together into a vector $\mathbf{w}_L$, with the remaining weights forming a vector $\widetilde{\mathbf{w}}$. The error function can then be expressed as $E(\mathbf{w}_L, \widetilde{\mathbf{w}})$, which is a quadratic function of $\mathbf{w}_L$. For any given value of $\widetilde{\mathbf{w}}$ we can perform a one-step exact minimization with respect to the $\mathbf{w}_L$ using SVD, in which $\widetilde{\mathbf{w}}$ is held fixed. We denote the optimum $\mathbf{w}_L$ by $\mathbf{w}_L(\widetilde{\mathbf{w}})$. A conventional non-linear optimization method (such as conjugate gradients, or the quasi-Newton methods to be described later) is used to minimize E with respect to $\widetilde{\mathbf{w}}$. Every time the value of $\widetilde{\mathbf{w}}$ is changed, the weights $\mathbf{w}_L$ are recomputed. We can therefore regard the final layer weights $\mathbf{w}_L$ as evolving on a fast time-scale compared to the remaining weights $\widetilde{\mathbf{w}}$. Effectively, the non-linear optimization is attempting to minimize a function $\widetilde{E}(\mathbf{w}_L(\widetilde{\mathbf{w}}), \widetilde{\mathbf{w}})$ with respect to $\widetilde{\mathbf{w}}$. An obvious advantage of this method is that the dimensionality of the effective search space for the non-linear algorithm is reduced, and we might hope that this would reduce the number of training iterations which is required to find a good solution. However, this is offset to some extent by the greater computational effort required at each such step. Webb and Lowe (1988) show that, for some problems, this hybrid approach can yield better solutions, or can require less computational effort, than full non-linear optimization of the complete network.

7.4 Optimization in practice

In order to apply the algorithms described in this chapter to real problems, we need to address a variety of practical issues. Here we discuss procedures for initializing the weights in a network, criteria used to terminate training, and normalized error functions for assessing the performance of trained networks.

All of the training algorithms which we consider in this chapter begin by initializing the weights in the network to some randomly chosen values. We have already seen that optimization algorithms which proceed by a steady monotonic reduction in the error function can become stuck in local minima. A suitable choice of initial weights is therefore potentially important in allowing the training algorithm to produce a good set of weights, and in addition may lead to improvements in the speed of training. Even stochastic algorithms such as gradient descent, which have the possibility of escaping from local minima, can show strong sensitivity to the initial conditions. The initialization of weights for radial

basis function networks has already been dealt with in Chapter 6. Here we shall concern ourselves with multi-layer perceptrons having sigmoidal hidden-unit activation functions.

The majority of initialization procedures in current use involve setting the weights to randomly chosen small values. Random values are used in order to avoid problems due to symmetries in the network. The initial weight values are chosen to be small so that sigmoidal activation functions are not driven into the saturation regions where $g'(a)$ is very small (which would lead to small ∇E, and consequently a very flat error surface). If the weights are too small, however, all of the sigmoidal activation functions will be approximately linear, which can again lead to slow training. This suggests that the summed inputs to the sigmoidal functions should be of order unity. A random initialization of the weights requires that some choice be made for the distribution function from which the weights are generated. We now examine the choice of this distribution in a little more detail.

We shall suppose that the input values to the network $x_1, \ldots x_d$ have been rescaled so as to have zero mean $\langle x_i \rangle = 0$ and unit variance $\langle x_i^2 \rangle = 1$, where the notation $\langle \cdot \rangle$ will be used to denote an average both over the training data set and over all the choices of initial network weights. The pre-processing of input data prior to network training, in order to achieve this normalization, is discussed in more detail in Section 8.2. The weights are usually generated from a simple distribution, such as a spherically symmetric Gaussian, for convenience, and this is generally taken to have zero mean, since there is no reason to prefer any other specific point in weight space. The choice of variance σ^2 for the distribution can be important, however. For a unit in the first hidden layer, the activation is given by $y = g(a)$ where

$$a = \sum_{i=0}^{d} w_i x_i. \tag{7.19}$$

Since the choice of weight values is uncorrelated with the inputs, the average of a is zero

$$\langle a \rangle = \sum_{i=0}^{d} \langle w_i x_i \rangle = \sum_{i=0}^{d} \langle w_i \rangle \langle x_i \rangle = 0 \tag{7.20}$$

since $\langle x_i \rangle = 0$. Next consider the variance of a

$$\langle a^2 \rangle = \left\langle \left(\sum_{i=0}^{d} w_i x_i \right) \left(\sum_{j=0}^{d} w_j x_j \right) \right\rangle = \sum_{i=0}^{d} \langle w_i^2 \rangle \langle x_i^2 \rangle = \sigma^2 d \tag{7.21}$$

where σ^2 is the variance of the distribution of weights, and we have used the fact

that the weight values are uncorrelated and hence $\langle w_i w_j \rangle = \delta_{ij}\sigma^2$, together with $\langle x_i^2 \rangle = 1$. As we have discussed already, we would like a to be of order unity so that the activations of the hidden units are determined by the non-linear part of the sigmoids, without saturating. From (7.21) this suggests that the standard deviation of the distribution used to generate the initial weights should scale like $\sigma \propto d^{-1/2}$. A similar argument can be applied to the weights feeding into any other unit in the network, if we assume that the outputs of hidden units are appropriately distributed.

Since a particular training run is sensitive to the initial conditions for the weights, it is common practice to train a particular network many times using different weight initializations. This leads to a set of different networks whose generalization performance can be compared by making use of independent data. In this case it is possible to keep the best network and simply discard the remainder. However, improved prediction capability can often be achieved by forming a *committee* of networks from amongst the better ones found during the training process, as discussed in Section 9.6. The use of multiple training runs also plays a related role in building a mixture model for the distribution of weight values in the Bayesian framework, as discussed in Section 10.7.

When using non-linear optimization algorithms, some choice must be made of when to stop the training process. Some of the possible choices are listed below:

1. Stop after a fixed number of iterations. The problem with this approach is that it is difficult to know in advance how many iterations would be appropriate, although an approximate idea can be obtained from some preliminary tests. If several networks are being trained (e.g. with various numbers of hidden units) then the appropriate number of iterations may be different for different networks.
2. Stop when a predetermined amount of CPU (central processing unit) time has been used. Again, it is difficult to know what constitutes a suitable time unless some preliminary tests are performed first. Some adjustment for different architectures may again be necessary.
3. Stop when the error function falls below some specified value. This suffers from the problem that the specified value may never be reached, so some limit on CPU time may also be required.
4. Stop when the relative change in error function falls below some specified value. This may lead to premature termination if the error function decreases relatively slowly during some part of the training run.
5. Stop training when the error measured using an independent validation set starts to increase. This approach is generally used as part of a strategy to optimize the generalization performance of the network, and will be discussed further in Section 9.2.4.

In practice some combination of the above methods may be employed as part of a largely empirical process of parameter optimization.

Since the value of the error function depends on the number of patterns, it is useful to consider a normalized error function for the purposes of assessing the

performance of a trained network. For a sum-of-squares error, an appropriate choice would be the normalized error function given by

$$\widetilde{E} = \sqrt{\frac{\sum_n \|\mathbf{y}(\mathbf{x}^n) - \mathbf{t}^n\|^2}{\sum_n \|\bar{\mathbf{t}} - \mathbf{t}^n\|^2}} \tag{7.22}$$

where $\bar{\mathbf{t}}$ is the mean of the target data over the test set (Webb *et al.*, 1988). This error function equals unity when the model is as good a predictor of the target data as the simple model $\mathbf{y} = \bar{\mathbf{t}}$, and equals zero if the model predicts the data values exactly. A value of $\widetilde{E}$ of 0.1 will often prove adequate for simple classification problems, while for regression applications a significantly smaller value may be needed. For reasons introduced in Chapter 1, and discussed at greater length in Chapter 9, the performance of the trained network should be assessed using a data set which is independent of the training data.

For classification problems, it is appropriate to test the performance of the trained network by assessing the number of misclassifications, or more generally the value of the total loss (Section 1.10).

7.5 Gradient descent

One of the simplest network training algorithms, and one which we have already encountered several times in previous chapters, is gradient descent, sometimes also known as *steepest descent*. In the *batch* version of gradient descent, we start with some initial guess for the weight vector (which is often chosen at random) denoted by $\mathbf{w}^{(0)}$. We then iteratively update the weight vector such that, at step τ, we move a short distance in the direction of the greatest rate of decrease of the error, i.e. in the direction of the negative gradient, evaluated at $\mathbf{w}^{(\tau)}$:

$$\Delta\mathbf{w}^{(\tau)} = -\eta \left.\nabla E\right|_{\mathbf{w}^{(\tau)}}. \tag{7.23}$$

Note that the gradient is re-evaluated at each step. In the *sequential*, or *pattern-based*, version of gradient descent, the error function gradient is evaluated for just one pattern at a time, and the weights updated using

$$\Delta\mathbf{w}^{(\tau)} = -\eta \left.\nabla E^n\right|_{\mathbf{w}^{(\tau)}} \tag{7.24}$$

where the different patterns n in the training set can be considered in sequence, or selected at random. The parameter η is called the *learning rate*, and, provided its value is sufficiently small, we expect that, in the batch version (7.23) of gradient descent, the value of E will decrease at each successive step, eventually leading to a weight vector at which the condition (7.1) is satisfied.

For the sequential update (7.24) we might also hope for a steady reduction in error since, for sufficiently small η, the average direction of motion in weight space should approximate the negative of the local gradient. In order to study this

more carefully, we note that sequential gradient descent (7.24) is reminiscent of the Robbins–Monro procedure (Section 2.4.1) for finding the zero of a regression function (in this case the error function gradient). The analogy becomes precise, and we are assured of convergence, if the learning rate parameter η is made to decrease at each step of the algorithm in accordance with the requirements of the theorem (Luo, 1991). These can be satisfied by choosing $\eta^{(\tau)} \propto 1/\tau$, although such a choice leads to very slow convergence. In practice, a constant value of η is often used as this generally leads to better results even though the guarantee of convergence is lost. There is still a serious difficulty with this approach, however. If η is too large, the algorithm may overshoot leading to an increase in E and possibly to divergent oscillations resulting in a complete breakdown in the algorithm. Conversely, if η is chosen to be too small the search can proceed extremely slowly, leading to very long computation times. Furthermore, the optimum value for η will typically change during the course of the minimization.

An important advantage of the sequential approach over batch methods arises if there is a high degree of redundant information in the data set. As a simple example, suppose that we create a larger training set from the original one simply by replicating the original data set ten times. Every evaluation of E then takes ten times as long, and so a batch algorithm will take ten times as long to find a given solution. By contrast, the sequential algorithm updates the weights after each pattern presentation, and so will be unaffected by the replication of data. Later in this chapter we describe a number of powerful optimization algorithms (such as conjugate gradients and quasi-Newton methods) which are intrinsically batch techniques. For such algorithms it is still possible to gain some of the advantages of sequential techniques by grouping the data into blocks and presenting the blocks sequentially as if each of them was representative of the whole data set. Some experimentation may be needed to determine a suitable size for the blocks.

Another potential advantage of the sequential approach is that, since it is a stochastic algorithm, it has the possibility of escape from local minima. Later in this chapter we shall discuss a number of algorithms which have the property that each step of the algorithm is guaranteed not to produce an increase in the error function. If such an algorithm finds its way into a local minimum it will typically remain there indefinitely.

7.5.1 *Convergence*

As we have already indicated, one of the limitations of the gradient descent technique is the need to choose a suitable value for the learning rate parameter η. The problems with gradient descent do not stop there, however. Figure 7.4 depicts the contours of E, for a hypothetical two-dimensional weight space, in which the curvature of E varies significantly with direction. At most points on the error surface, the local gradient does not point directly towards the minimum. Gradient descent then takes many small steps to reach the minimum, and is clearly a very inefficient procedure.

We can gain deeper insight into the nature of this problem by considering

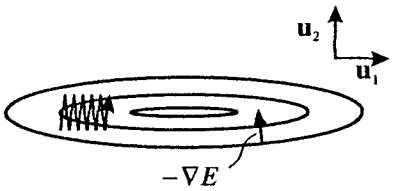

Figure 7.4. Schematic illustration of fixed-step gradient descent for an error function which has substantially different curvatures along different directions. Ellipses depict contours of constant E, so that the error surface has the form of a long valley. The vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ represent the eigenvectors of the Hessian matrix. Note that, for most points in weight space, the local negative gradient vector $-\nabla E$ does not point towards the minimum of the error function. Successive steps of gradient descent can oscillate across the valley, with very slow progress along the valley towards the minimum.

the quadratic approximation to the error function in the neighbourhood of the minimum, discussed earlier in Section 7.2. From (7.10), (7.11) and (7.13), the gradient of the error function in this approximation can be written as

$$\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i. \tag{7.25}$$

From (7.13) we also have

$$\Delta \mathbf{w} = \sum_i \Delta\alpha_i \mathbf{u}_i. \tag{7.26}$$

Combining (7.25) with (7.26) and the gradient descent formula (7.23), and using the orthonormality relation (7.12) for the eigenvectors of the Hessian, we obtain the following expression for the change in α_i at each step of the gradient descent algorithm

$$\Delta\alpha_i = -\eta\lambda_i\alpha_i \tag{7.27}$$

from which it follows that

$$\alpha_i^{\text{new}} = (1 - \eta\lambda_i)\alpha_i^{\text{old}} \tag{7.28}$$

where 'old' and 'new' denote values before and after a weight update. Using the orthonormality relation (7.12) for the eigenvectors, together with (7.13), we have

$$\mathbf{u}_i^{\mathrm{T}}(\mathbf{w} - \mathbf{w}^*) = \alpha_i \tag{7.29}$$

and so α_i can be interpreted as the distance to the minimum along the direction $\mathbf{u}_i$. From (7.28) we see that these distances evolve independently such that, at each step, the distance along the direction of $\mathbf{u}_i$ is multiplied by a factor $(1-\eta\lambda_i)$. After a total of T steps we have

$$\alpha_i^{(T)} = (1 - \eta\lambda_i)^T \alpha_i^{(0)} \tag{7.30}$$

and so, provided $|1 - \eta\lambda_i| < 1$, the limit $T \to \infty$ leads to $\alpha_i = 0$, which from (7.29) shows that $\mathbf{w} = \mathbf{w}^*$ and so the weight vector has reached the minimum of the error. Note that (7.30) demonstrates that gradient descent leads to linear convergence in the neighbourhood of a minimum. Also, convergence to the stationary point requires that all of the λ_i be positive, which in turn implies that the stationary point is indeed a minimum (Exercise 7.1).

By making η larger we can make the factor $(1 - \eta\lambda_i)$ smaller and hence improve the speed of convergence. There is a limit to how large η can be made, however. We can permit $(1-\eta\lambda_i)$ to go negative (which gives oscillating values of α_i) but we must ensure that $|1-\eta\lambda_i| < 1$ otherwise the α_i values will diverge. This limits the value of η to $\eta < 2/\lambda_{\max}$ where $\lambda_{\max}$ is the largest of the eigenvalues. The rate of convergence, however, is dominated by the smallest eigenvalue, so with η set to its largest permitted value, the convergence along the direction corresponding to the smallest eigenvalue (the long axis of the ellipse in Figure 7.4) will be governed by

$$\left(1 - \frac{2\lambda_{\min}}{\lambda_{\max}}\right) \tag{7.31}$$

where $\lambda_{\min}$ is the smallest eigenvalue. If the ratio $\lambda_{\min}/\lambda_{\max}$ (whose reciprocal is known as the *condition number* of the Hessian) is very small, corresponding to highly elongated elliptical error contours as in Figure 7.4, then progress towards the minimum will be extremely slow. From our earlier discussion of quadratic error surfaces, we might expect to be able to find the minimum exactly using as few as $W(W+3)/2$ error function evaluations. Gradient descent is an extremely inefficient algorithm for error function minimization, since the number of function evaluations can easily be very much greater than this. Later we shall encounter algorithms which are guaranteed to find the minimum of a quadratic error surface exactly in a small, fixed number of steps which is $\mathcal{O}(W^2)$.

The gradient descent procedure we have described so far involves taking a succession of finite steps through weight space. We can instead imagine the evolution of the weight vector taking place continuously as a function of time τ. The gradient descent rule is then replaced by a set of coupled non-linear ordinary differential equations of the form

$$\frac{dw_i}{d\tau} = -\eta \frac{\partial E}{\partial w_i} \tag{7.32}$$

where w_i represents any weight parameter in the network. These equations correspond to the motion of a massless particle with position vector $\mathbf{w}$ moving in a potential field $E(\mathbf{w})$ subject to viscous drag with viscosity coefficient η^{-1}. They represent a set of *stiff* differential equations (ones characterized by several widely differing time-scales) as a consequence of the fact that the Hessian matrix often has widely differing eigenvalues. The simple gradient descent formula (7.23) represents a 'fixed-step forward Euler' technique for solving (7.32), which is a particularly inefficient approach for stiff equations. Application of specialized techniques for solving stiff ordinary differential equations (Gear, 1971) to the system in (7.32) can give significant improvements in convergence time (Owens and Filkin, 1989).

7.5.2 *Momentum*

One very simple technique for dealing with the problem of widely differing eigenvalues is to add a *momentum* term to the gradient descent formula (Plaut *et al.*, 1986). This effectively adds inertia to the motion through weight space (Exercise 7.3) and smoothes out the oscillations depicted in Figure 7.4. The modified gradient descent formula is given by

$$\Delta \mathbf{w}^{(\tau)} = -\eta \, \nabla E|_{\mathbf{w}^{(\tau)}} + \mu \Delta \mathbf{w}^{(\tau-1)} \tag{7.33}$$

where μ is called the momentum parameter.

To understand the effect of the momentum term, consider first the motion through a region of weight space for which the error surface has relatively low curvature, as indicated in Figure 7.5. If we make the approximation that the gradient is unchanging, then we can apply (7.33) iteratively to a long series of weight updates, and then sum the resulting arithmetic series to give

$$\Delta \mathbf{w} = -\eta \nabla E \{1 + \mu + \mu^2 + \ldots\} \tag{7.34}$$

$$= -\frac{\eta}{1-\mu} \nabla E \tag{7.35}$$

and we see that the result of the momentum term is to increase the effective learning rate from η to $\eta/(1-\mu)$.

By contrast, in a region of high curvature in which the gradient descent is oscillatory, as indicated in Figure 7.6, successive contributions from the momentum term will tend to cancel, and the effective learning rate will be close to η. Thus, the momentum term can lead to faster convergence towards the minimum without causing divergent oscillations. A schematic illustration of the effect of a momentum term is shown in Figure 7.7. From (7.35) we see that μ must lie between in the range $0 \leq \mu \leq 1$.

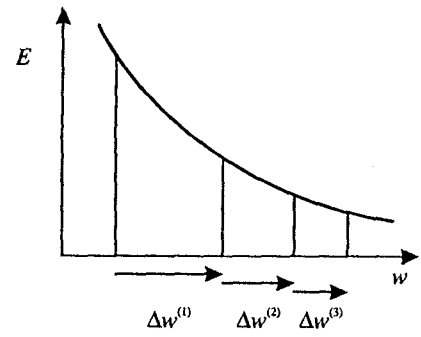

Figure 7.5. With a fixed learning rate parameter, gradient descent down a surface with low curvature leads to successively smaller steps (linear convergence). In such a situation, the effect of a momentum term is similar to an increase in the effective learning rate parameter.

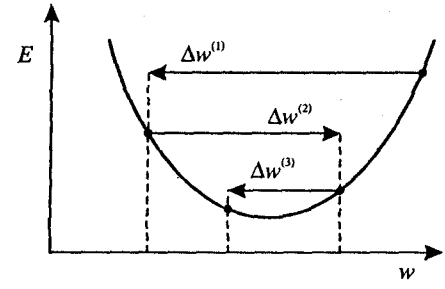

Figure 7.6. For a situation in which successive steps of gradient descent are oscillatory, a momentum term has little influence on the effective value of the learning rate parameter.

The inclusion of momentum generally leads to a significant improvement in the performance of gradient descent. Nevertheless, the algorithm remains relatively inefficient. The inclusion of momentum also introduces a second parameter μ whose value needs to be chosen, in addition to that of the learning rate parameter η.

7.5.3 *Enhanced gradient descent*

As we have seen, gradient descent, even with a momentum term included, is not a particularly efficient algorithm for error function minimization. There have been numerous attempts in recent years to improve the performance of basic gradient

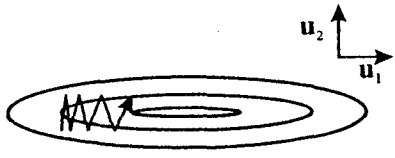

Figure 7.7. Illustration of the effect of adding a momentum term to the gradient descent algorithm, showing the more rapid progress along the valley of the error function, compared with the unmodified gradient descent shown in Figure 7.4.

descent for neural network training by making various *ad hoc* modifications. We shall not attempt to review them all here as the literature is much too extensive, and we will shortly be considering several robust, theoretically well-founded optimization algorithms. Instead we consider a few illustrative examples of such techniques which attempt to address various deficiencies of the basic gradient descent procedure.

One obvious problem with simple gradient descent plus momentum is that it contains two parameters, η and μ, whose values must be selected by trial and error. The optimum values for these parameters will depend on the particular problem, and will typically vary during training. We might therefore seek some procedure for setting these automatically as part of the training algorithm. One such approach is the *bold driver* technique (Vogl *et al.*, 1988; Battiti, 1989). Consider the situation without a momentum term first. The idea is to check whether the error function has actually decreased after each step of the gradient descent. If it has increased then the algorithm must have overshot the minimum (i.e. the minimum along the direction of the weight change) and so the learning rate parameter must have been too large. In this case the weight change is undone, and the learning rate is decreased. This process is repeated until a decrease in error is found. If, however, the error decreased at a given step, then the new weight values are accepted. However, the learning rate might have been too small, and so its value is increased. This leads to the following prescription for updating the learning rate parameter:

$$\eta_{\text{new}} = \begin{cases} \rho\eta_{\text{old}} & \text{if } \Delta E < 0 \\ \sigma\eta_{\text{old}} & \text{if } \Delta E > 0. \end{cases} \tag{7.36}$$

The parameter ρ is chosen to be slightly larger than unity (a typical value might be $\rho = 1.1$) in order to avoid frequent occurrences of an error increase, since in such cases the error evaluation is wasted. The parameter σ is taken to be significantly less than unity ($\sigma = 0.5$ is typical) so that the algorithm quickly reverts to finding a step which decreases the error, again to minimize wasted computation. Many variations of this heuristic are possible, such as increasing η

linearly (by a fixed increment) rather than exponentially (by a fixed factor). If we include momentum in the bold driver algorithm, the momentum coefficient can be set to some fixed value (selected in an *ad hoc* fashion), but the weight update is usually reset along the negative gradient direction after every occurrence of an error function increase, which is equivalent to setting the momentum coefficient temporarily to zero (Vogl *et al.*, 1988).

A more principled approach to setting the optimal learning rate parameter was introduced by Le Cun *et al.* (1993). In Section 7.5.1 we showed that the largest value which can be used for the learning rate parameter was given by $\eta_{\max} = 2/\lambda_{\max}$, where $\lambda_{\max}$ is the largest eigenvalue of the Hessian matrix. It is easily shown (Exercise 7.5) that if an arbitrary vector is alternately normalized and then multiplied by the Hessian, it eventually converges to $\lambda_{\max}$ times the corresponding eigenvector. The length of this vector then gives $\lambda_{\max}$ itself. Evaluation of the product of the Hessian with a vector can be performed efficiently by using the $\mathcal{R}\{\cdot\}$-operator technique discussed in Section 4.10.7. Once a suitable value for the learning rate has been determined, the standard gradient descent technique is applied.

We have already noted that the (negative) gradient vector need not point towards the error function minimum, even for a quadratic error surface, as indicated in Figure 7.4. In addition, we have seen that long narrow valleys in the error function, characterized by a Hessian matrix with widely differing eigenvalues, can lead to very slow progress down the valley, as a consequence of the need to keep the learning rate small in order to avoid divergent oscillations across the valley. One approach that has been suggested for dealing with this problem (Jacobs, 1988) is to introduce a separate learning rate for each weight in the network, with procedures for updating these learning rates during the training process. The gradient descent rule then becomes

$$\Delta w_i^{(\tau)} = -\eta_i^{(\tau)} \frac{\partial E}{\partial w_i^{(\tau)}}. \tag{7.37}$$

Heuristically, we might wish to increase a particular learning rate when the derivative of E with respect to the corresponding parameter has the same sign on consecutive steps since this weight is moving steadily in the downhill direction. Conversely, if the sign of the gradient changes on consecutive steps, this signals oscillation, and the learning rate parameter should be decreased.

One way to implement this is to take

$$\Delta \eta_i^{(\tau)} = \gamma g_i^{(\tau)} g_i^{(\tau-1)} \tag{7.38}$$

where

$$g_i^{(\tau)} = \frac{\partial E}{\partial w_i^{(\tau)}} \tag{7.39}$$

and $\gamma > 0$ is a step-size parameter. This prescription is called the *delta-delta* rule (since, in Jacobs (1988) the notation δ_i was used instead of g_i to denote the components of the local gradient vector). For the case of a quadratic error surface, it can be derived by minimizing the error with respect to the learning rate parameters (Exercise 7.6). This rule does not work well in practice since it can lead to negative values for the learning rate, which results in uphill steps, unless the value of γ is set very small, in which case the algorithm exhibits little improvement over conventional gradient descent. A modification to the algorithm, known as the *delta-bar-delta* rule is to take

$$\Delta\eta_i^{(\tau)} = \begin{cases} \kappa & \text{if } \bar{g}_i^{(\tau-1)} g_i^{(\tau)} > 0 \\ -\phi\eta_i^{(\tau)} & \text{if } \bar{g}_i^{(\tau-1)} g_i^{(\tau)} < 0 \end{cases} \tag{7.40}$$

where

$$\bar{g}_i^{(\tau)} = (1-\theta) g_i^{(\tau)} + \theta \bar{g}_i^{(\tau-1)} \tag{7.41}$$

so that $\bar{g}$ is an exponentially weighted average of the current and previous values of g. This algorithm appears to work moderately well in practice, at least for some problems. One of its obvious drawbacks, however, is that it now contains four parameters (θ, ϕ, κ and μ) if we include momentum. A more serious difficulty is that the algorithm rests on the assumption that we can regard the weight parameters as being relatively independent. This would be the case for a quadratic error function if the Hessian matrix were diagonal (so that the major axes of the ellipse in Figure 7.3 were aligned with the weight axes). In practice, the weights in a typical neural network are strongly coupled, leading to a Hessian matrix which is often far from diagonal. The solution to this problem lies in a number of standard optimization algorithms which we shall discuss shortly.

Another heuristic scheme, known as *quickprop* (Fahlman, 1988), also treats the weights as if they were quasi-independent. The idea is to approximate the error surface, as a function of each of the weights, by a quadratic polynomial (i.e. a parabola), and then to use two successive evaluations of the error function, and an evaluation of its gradient, to determine the coefficients of the polynomial. At the next step of the iteration, the weight parameter is moved to the minimum of the parabola. This leads to an expression for the weight update at step τ given by (Exercise 7.7)

$$\Delta w_i^{(\tau+1)} = \frac{g_i^{(\tau)}}{g_i^{(\tau-1)} - g_i^{(\tau)}} \Delta w_i^{(\tau)}. \tag{7.42}$$

The algorithm can be started using a single step of gradient descent. This assumes that the result of the local quadratic fit is to give a parabola with a minimum. If instead it leads to a parabola with a maximum, the algorithm can take an

uphill step. Also, some bound on the maximum size of step needs to be imposed to deal with the problem of a nearly flat parabola, and several other fixes are needed in order to get the algorithm to work in practice.

7.6 Line search

The algorithms which are described in this chapter involve taking a sequence of steps through weight space. It is convenient to consider each of these steps in two parts. First we must decide the direction in which to move, and second, we must decide how far to move in that direction. With simple gradient descent, the direction of each step is given by the local negative gradient of the error function, and the step size is determined by an arbitrary learning rate parameter. We might expect that a better procedure would be to move along the direction of the negative gradient to find the point at which the error is minimized. More generally we can consider some *search direction* in weight space, and then find the minimum of the error function along that direction. This procedure is referred to as a *line search*, and it forms the basis for several algorithms which are considerably more powerful than gradient descent. We first consider how line searches can be implemented in practice.

Suppose that at step τ in some algorithm the current weight vector is $\mathbf{w}^{(\tau)}$, and we wish to consider a particular search direction $\mathbf{d}^{(\tau)}$ through weight space. The minimum along the search direction then gives the next value for the weight vector:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \lambda^{(\tau)}\mathbf{d}^{(\tau)} \tag{7.43}$$

where the parameter $\lambda^{(\tau)}$ is chosen to minimize

$$E(\lambda) = E(\mathbf{w}^{(\tau)} + \lambda\mathbf{d}^{(\tau)}). \tag{7.44}$$

This gives us an automatic procedure for setting the step length, once we have chosen the search direction.

The line search represents a one-dimensional minimization problem. A simple approach would be to proceed along the search direction in small steps, evaluating the error function at each new position, and stop when the error starts to increase (Hush and Salas, 1988). It is possible, however, to find very much more efficient approaches (Press *et al.*, 1992). Consider first the issue of whether to make use of gradient information in performing a line search. We have already argued that there is generally a substantial advantage to be gained from using gradient information for the general problem of seeking the minimum of the error function E in the W-dimensional weight space. For the sub-problem of line search, however, the argument is somewhat different. Since this is now a one-dimensional problem, both the value of the error function and the gradient of the error function each represent just one piece of information. An error function calculation requires one forward propagation and hence needs $\sim 2NW$ operations,

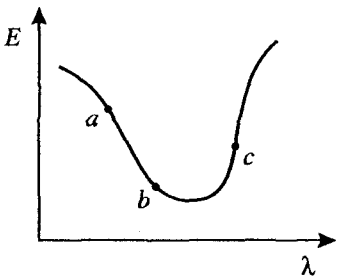

Figure 7.8. An example of an error function which depends on a parameter λ governing distance along the search direction, showing a minimum which has been bracketed. The three points $a < b < c$ are such that $E(a) > E(b)$ and $E(c) > E(b)$. This ensures that the minimum lies somewhere in the interval (a, c).

where N is the number of patterns in the data set. An error function gradient evaluation, however, requires a forward propagation, a backward propagation, and a set of multiplications to form the derivatives. It therefore needs $\sim 5NW$ operations, although it does allow the error function itself to be evaluated as well. On balance, the line search is slightly more efficient if it makes use of error function evaluations only.

Each line search proceeds in two stages. The first stage is to *bracket* the minimum by finding three points $a < b < c$ along the search direction such that $E(a) > E(b)$ and $E(c) > E(b)$, as shown in Figure 7.8. Since the error function is continuous, this ensures that there is a minimum somewhere in the interval (a, c) (Press *et al.*, 1992). The second stage is to locate the minimum itself. Since the error function is smooth and continuous, this can be achieved by a process of parabolic interpolation. This involves fitting a quadratic polynomial to the error function evaluated at three successive points, and then moving to the minimum of the parabola, as illustrated in Figure 7.9. The process can be repeated by evaluating the error function at the new point, and then fitting a new parabola to this point and two of the previous points. In practice, several refinements are also included, leading to the very robust *Brent's* algorithm (Brent, 1973). Line-search algorithms, and termination criteria, are reviewed in Luenberger (1984).

An important issue concerns the accuracy with which the line searches are performed. Depending on the particular algorithm in which the line search is to be used, it may be wasteful to invest too much computational time in evaluating the minimum along each search direction to high accuracy. We shall return to this point later. For the moment, we make one comment regarding the limit of accuracy which can be achieved in a line search. Near a minimum at λ_0, the error function along the search direction can be approximated by

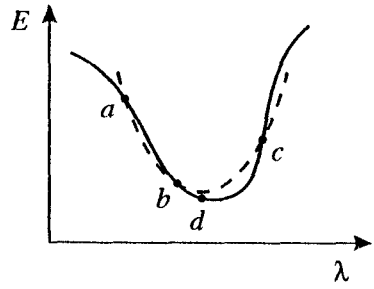

Figure 7.9. An illustration of the process of parabolic interpolation used to perform line-search minimization. The solid curve depicts the error as a function of distance λ along the search direction, and the error is evaluated at three points $a < b < c$ which are such that $E(a) > E(b)$ and $E(c) > E(b)$. A parabola (shown dotted) is fitted to the three points a, b, c. The minimum of the parabola, at d, gives an approximation to the minimum of $E(\lambda)$. The process can be repeated by fitting another parabola through three points given by d and whichever of two of the previous points have the smallest error values (b and c in this example).

$$E(\lambda) = E(\lambda_0) + \frac{1}{2}E''(\lambda_0)(\lambda - \lambda_0)^2. \tag{7.45}$$

Thus $\lambda - \lambda_0$ must typically be at least of the order of the *square root* of the machine precision before the difference between $E(\lambda)$ and $E(\lambda_0)$ is significant. This limits the accuracy with which the minimum can be found. For double-precision arithmetic this implies that the minimum can only be found to a relative accuracy of approximately 3×10^{-8}. In practice is may be better to settle for much lower accuracy than this.

7.7 Conjugate gradients

In the previous section we considered procedures for line-search minimization along a specified search direction. To apply line search to the problem of error function minimization we need to choose a suitable search direction at each stage of the algorithm. Suppose we have already minimized along a search direction given by the local negative gradient vector. We might suppose that the search direction at the next iteration should be given by the negative gradient vector at the new position. However, the use of successive gradient vectors turns out in general not to represent the best choice of search direction. To see why, we note that at the minimum of the line search we have, from (7.44)

$$\frac{\partial}{\partial \lambda} E(\mathbf{w}^{(\tau)} + \lambda \mathbf{d}^{(\tau)}) = 0 \tag{7.46}$$

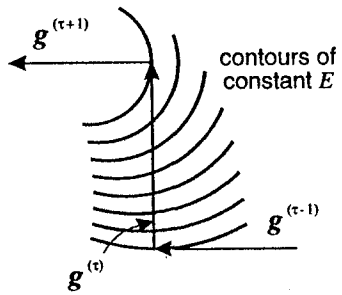

Figure 7.10. After a line minimization, the new gradient is orthogonal to the line-search direction. Thus, if the search directions are always chosen to coincide with the negative gradients of the error function, as indicated here, then successive search directions will be orthogonal, and the error function minimization will typically proceed very slowly.

which gives

$$\mathbf{g}^{(\tau+1)\mathrm{T}}\mathbf{d}^{(\tau)} = 0 \tag{7.47}$$

where $\mathbf{g} \equiv \nabla E$. Thus, the gradient at the new minimum is orthogonal to the previous search direction, as illustrated geometrically in Figure 7.10. Choosing successive search directions to be the local (negative) gradient directions can lead to the problem already indicated in Figure 7.4 in which the search point oscillates on successive steps while making little progress towards the minimum. The algorithm can then take many steps to converge, even for a quadratic error function.

The solution to this problem lies in choosing the successive search directions $\mathbf{d}^{(\tau)}$ such that, at each step of the algorithm, the component of the gradient parallel to the previous search direction, which has just been made zero, is unaltered (to lowest order). This is illustrated in Figure 7.11. Suppose we have already performed a line minimization along the direction $\mathbf{d}^{(\tau)}$, starting from the point $\mathbf{w}^{(\tau)}$, to give the new point $\mathbf{w}^{(\tau+1)}$. Then at the point $\mathbf{w}^{(\tau+1)}$ we have

$$\mathbf{g}(\mathbf{w}^{(\tau+1)})^{\mathrm{T}}\mathbf{d}^{(\tau)} = 0. \tag{7.48}$$

We now choose the next search direction $\mathbf{d}^{(\tau+1)}$ such that, along this new direction, we retain the property that the component of the gradient parallel to the previous search direction remains zero (to lowest order). Thus we require that

$$\mathbf{g}(\mathbf{w}^{(\tau+1)} + \lambda\mathbf{d}^{(\tau+1)})^{\mathrm{T}}\mathbf{d}^{(\tau)} = 0 \tag{7.49}$$

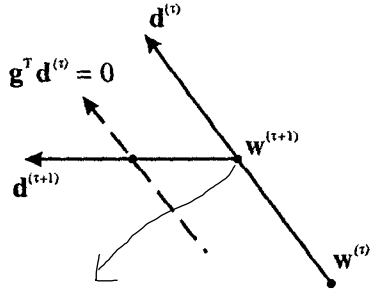

Figure 7.11. This diagram illustrates the concept of conjugate directions. Suppose a line search has been performed along the direction $\mathbf{d}^{(\tau)}$ starting from the point $\mathbf{w}^{(\tau)}$, to give an error *minimum* along the search path at the point $\mathbf{w}^{(\tau+1)}$. The direction $\mathbf{d}^{(\tau+1)}$ is said to be conjugate to the direction $\mathbf{d}^{(\tau)}$ if the component of the gradient parallel to the direction $\mathbf{d}^{(\tau)}$, which has just be made zero, remains zero (to lowest order) as we move along the direction $\mathbf{d}^{(\tau+1)}$.

as shown in Figure 7.11. If we now expand (7.49) to first order in λ, and note that the zeroth-order term vanishes as a consequence of (7.48), we obtain

$$\mathbf{d}^{(\tau+1)\mathrm{T}}\mathbf{H}\mathbf{d}^{(\tau)} = 0 \tag{7.50}$$

where $\mathbf{H}$ is the Hessian matrix evaluated at the point $\mathbf{w}^{(\tau+1)}$. If the error surface is quadratic, this relation holds for arbitrary values of λ in (7.49) since the Hessian matrix is constant, and higher-order terms in the expansion of (7.49) in powers of λ vanish. Search directions which satisfy (7.50) are said to be *non-interfering* or *conjugate*. In fact, we shall see that it is possible to construct a sequence of successive search directions $\mathbf{d}^{(\tau)}$ such that each direction is conjugate to all previous directions, up to the dimensionality W of the search space. This leads naturally to the conjugate gradient optimization algorithm.

7.7.1 *Quadratic error function*

In order to introduce the conjugate gradient algorithm, we follow Johansson *et al.* (1992) and consider first the case of a quadratic error function of the form

$$E(\mathbf{w}) = E_0 + \mathbf{b}^{\mathrm{T}}\mathbf{w} + \frac{1}{2}\mathbf{w}^{\mathrm{T}}\mathbf{H}\mathbf{w} \tag{7.51}$$

in which the parameters $\mathbf{b}$ and $\mathbf{H}$ are constant, and $\mathbf{H}$ is assumed to be positive definite. The local gradient of this error function is given by

$$\mathbf{g}(\mathbf{w}) = \mathbf{b} + \mathbf{H}\mathbf{w} \tag{7.52}$$

and the error function (7.51) is minimized at the point $\mathbf{w}^*$ given, from (7.52), by

$$\mathbf{b} + \mathbf{H}\mathbf{w}^* = 0. \tag{7.53}$$

Suppose we can find a set of W vectors (where W is the dimensionality of the weight space) which are mutually conjugate with respect to $\mathbf{H}$ so that

$$\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_i = 0 \qquad j \neq i \tag{7.54}$$

then it is easily shown that these vectors will be linearly independent if $\mathbf{H}$ is positive definite (Exercise 7.8). Such vectors therefore form a complete, but non-orthogonal, basis set in weight space. Suppose we are starting from some point $\mathbf{w}_1$, and we wish to get to the minimum $\mathbf{w}^*$ of the error function. The difference between the vectors $\mathbf{w}_1$ and $\mathbf{w}^*$ can be written as a linear combination of the conjugate direction vectors in the form

$$\mathbf{w}^* - \mathbf{w}_1 = \sum_{i=1}^{W} \alpha_i \mathbf{d}_i. \tag{7.55}$$

Note that, if we define

$$\mathbf{w}_j = \mathbf{w}_1 + \sum_{i=1}^{j-1} \alpha_i \mathbf{d}_i \tag{7.56}$$

then (7.55) can be written as an iterative equation in the form

$$\mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_j \mathbf{d}_j. \tag{7.57}$$

This represents a succession of steps parallel the conjugate directions, with step lengths controlled by the parameters α_j.

In order to find expressions for the α's we multiply (7.55) by $\mathbf{d}_j^{\mathrm{T}}\mathbf{H}$ and make use of (7.53) to give

$$-\mathbf{d}_j^{\mathrm{T}}(\mathbf{b} + \mathbf{H}\mathbf{w}_1) = \sum_{i=1}^{W} \alpha_i \mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_i. \tag{7.58}$$

We now see the significance of using mutually conjugate directions, since (7.54) shows that the terms on the right-hand side of (7.58) decouple, allowing an explicit solution for the α's in the form

$$\alpha_j = -\frac{\mathbf{d}_j^{\mathrm{T}}(\mathbf{b} + \mathbf{H}\mathbf{w}_1)}{\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_j}. \tag{7.59}$$

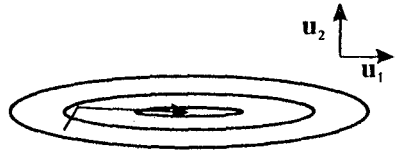

Figure 7.12. Schematic illustration of the application of the conjugate gradient algorithm to the minimization of a two-dimensional quadratic error function. The algorithm moves to the minimum of the error after two steps. This should be compared with Figures 7.4 and 7.7.

Without this property, (7.58) would represent a set of coupled equations for the α_i.

We can write (7.59) in a more convenient form as follows. From (7.56) we have

$$\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{w}_j = \mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{w}_1 \tag{7.60}$$

where we have again used the conjugacy condition (7.54). This allows the numerator on the right-hand side of (7.59) to be written in the form

$$\mathbf{d}_j^{\mathrm{T}}(\mathbf{b} + \mathbf{H}\mathbf{w}_1) = \mathbf{d}_j^{\mathrm{T}}(\mathbf{b} + \mathbf{H}\mathbf{w}_j) = \mathbf{d}_j^{\mathrm{T}}\mathbf{g}_j \tag{7.61}$$

where $\mathbf{g}_j \equiv \mathbf{g}(\mathbf{w}_j)$, and we have made use of (7.52). Thus, α_j can be written in the form

$$\alpha_j = -\frac{\mathbf{d}_j^{\mathrm{T}}\mathbf{g}_j}{\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_j}. \tag{7.62}$$

We now give a simple inductive argument to show that, if the weights are incremented using (7.57) with the α_j given by (7.62) then the gradient vector $\mathbf{g}_j$ at the jth step is orthogonal to all previous conjugate directions. It therefore follows that after W steps the components of the gradient along all directions have been made zero, and so we will have arrived at the minimum of the quadratic form. This is illustrated schematically for a two-dimensional space in Figure 7.12. To derive the orthogonality property, we note from (7.52) that

$$\mathbf{g}_{j+1} - \mathbf{g}_j = \mathbf{H}(\mathbf{w}_{j+1} - \mathbf{w}_j) = \alpha_j\mathbf{H}\mathbf{d}_j \tag{7.63}$$

where we have used (7.57). We now take the scalar product of this equation with $\mathbf{d}_j$, and use the definition of α_j given by (7.62), to give

$$\mathbf{d}_j^{\mathrm{T}}\mathbf{g}_{j+1} = 0. \tag{7.64}$$

Similarly, from (7.63), we have

$$\mathbf{d}_k^{\mathrm{T}}(\mathbf{g}_{j+1} - \mathbf{g}_j) = \alpha_j \mathbf{d}_k^{\mathrm{T}}\mathbf{H}\mathbf{d}_j = 0 \qquad \text{for all } k < j \leq W. \tag{7.65}$$

Applying the technique of induction to (7.64) and (7.65) we obtain the result that

$$\mathbf{d}_k^{\mathrm{T}}\mathbf{g}_j = 0 \qquad \text{for all } k < j \leq W \tag{7.66}$$

as required.

The next problem is how to construct a set of mutually conjugate directions. This can be achieved by selecting the first direction to be the negative gradient $\mathbf{d}_1 = -\mathbf{g}_1$, and then choosing each successive direction to be a linear combination of the current gradient and the previous search direction

$$\mathbf{d}_{j+1} = -\mathbf{g}_{j+1} + \beta_j \mathbf{d}_j. \tag{7.67}$$

The coefficients β_j can be found by imposing the conjugacy condition (7.54) which gives

$$\beta_j = \frac{\mathbf{g}_{j+1}^{\mathrm{T}}\mathbf{H}\mathbf{d}_j}{\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_j}. \tag{7.68}$$

In fact, it is easily shown by induction (Exercise 7.9) that successive use of the construction given by (7.67) and (7.68) generates a set of W mutually conjugate directions.

From (7.67) it follows that $\mathbf{d}_k$ is given by a linear combination of all previous gradient vectors

$$\mathbf{d}_k = -\mathbf{g}_k + \sum_{l=1}^{k-1} \gamma_l \mathbf{g}_l. \tag{7.69}$$

Using (7.66) we then have

$$\mathbf{g}_k^{\mathrm{T}}\mathbf{g}_j = \sum_{l=1}^{k-1} \gamma_l \mathbf{g}_l^{\mathrm{T}}\mathbf{g}_j \qquad \text{for all } k < j \leq W. \tag{7.70}$$

Since the initial search direction is just $\mathbf{d}_1 = -\mathbf{g}_1$, we can use (7.66) to show that $\mathbf{g}_1^{\mathrm{T}}\mathbf{g}_j = 0$, so that the gradient at step j is orthogonal to the initial gradient. If we apply induction to (7.70) we find that the current gradient is orthogonal to

all previous gradients

$$\mathbf{g}_k^{\mathrm{T}}\mathbf{g}_j = 0 \qquad \text{for all } k < j \leq W. \tag{7.71}$$

We have now developed an algorithm for finding the minimum of a general quadratic error function in at most W steps. Starting from a randomly chosen point $\mathbf{w}_1$, successive conjugate directions are constructed using (7.67) in which the parameters β_j are given by (7.68). At each step the weight vector is incremented along the corresponding direction using (7.57) in which the parameter α_j is given by (7.62).

7.7.2 *The conjugate gradient algorithm*

So far our discussion of conjugate gradients has been limited to quadratic error functions. For a general non-quadratic error function, the error in the neighbourhood of a given point will be approximately quadratic, and so we may hope that repeated application of the above procedure will lead to effective convergence to a minimum of the error. The step length in this procedure is governed by the coefficient α_j given by (7.62), and the search direction is determined by the coefficient β_j given by (7.68). These expressions depend on the Hessian matrix $\mathbf{H}$. For a non-quadratic error function, the Hessian matrix will depend on the current weight vector, and so will need to be re-evaluated at each step of the algorithm. Since the evaluation of $\mathbf{H}$ is computationally costly for non-linear neural networks, and since its evaluation would have to be done repeatedly, we would like to avoid having to use the Hessian. In fact, it turns out that the coefficients α_j and β_j can be found without explicit knowledge of $\mathbf{H}$. This leads to the *conjugate gradient algorithm* (Hestenes and Stiefel, 1952; Press *et al.*, 1992).

Consider first the coefficient β_j. If we substitute (7.63) into (7.68) we obtain

$$\beta_j = \frac{\mathbf{g}_{j+1}^{\mathrm{T}}(\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{d}_j^{\mathrm{T}}(\mathbf{g}_{j+1} - \mathbf{g}_j)} \tag{7.72}$$

which is known as the *Hestenes–Stiefel* expression. From (7.66) and (7.67) we have

$$\mathbf{d}_j^{\mathrm{T}}\mathbf{g}_j = -\mathbf{g}_j^{\mathrm{T}}\mathbf{g}_j \tag{7.73}$$

which, together with a further use of (7.66), allows (7.72) to be written in the *Polak–Ribiere* form

$$\beta_j = \frac{\mathbf{g}_{j+1}^{\mathrm{T}}(\mathbf{g}_{j+1} - \mathbf{g}_j)}{\mathbf{g}_j^{\mathrm{T}}\mathbf{g}_j}. \tag{7.74}$$

Similarly, we can use the orthogonality property (7.71) for the gradients to simplify (7.74) further, resulting in the *Fletcher–Reeves* form

$$\beta_j = \frac{\mathbf{g}_{j+1}^{\mathrm{T}}\mathbf{g}_{j+1}}{\mathbf{g}_j^{\mathrm{T}}\mathbf{g}_j}. \tag{7.75}$$

Note that these three expressions for β_j are equivalent provided the error function is exactly quadratic. In practice, the error function will not be quadratic, and these different expressions for β_j can give different results. The Polak–Ribiere form is generally found to give slightly better results than the other expressions. This is probably due to the fact that, if the algorithm is making little progress, so that successive gradient vectors are very similar, the Polak–Ribiere form gives a small value for β_j so that the search direction in (7.67) tends to be reset to the negative gradient direction, which is equivalent to restarting the conjugate gradient procedure.

We also wish to avoid the use of the Hessian matrix to evaluate α_j. In fact, in the case of a quadratic error function, the correct value of α_j can be found by performing a line minimization along the search direction. To see this, consider a quadratic error (7.51) as a function of the parameter α along the search direction $\mathbf{d}_j$, starting at the point $\mathbf{w}_j$, given by

$$E(\mathbf{w}_j + \alpha\mathbf{d}_j) = E_0 + \mathbf{b}^{\mathrm{T}}(\mathbf{w}_j + \alpha\mathbf{d}_j) + \frac{1}{2}(\mathbf{w}_j + \alpha\mathbf{d}_j)^{\mathrm{T}}\mathbf{H}(\mathbf{w}_j + \alpha\mathbf{d}_j). \tag{7.76}$$

If we set the derivative of this expression with respect to α equal to zero we obtain

$$\alpha_j = -\frac{\mathbf{d}_j^{\mathrm{T}}\mathbf{g}_j}{\mathbf{d}_j^{\mathrm{T}}\mathbf{H}\mathbf{d}_j} \tag{7.77}$$

where we have used the expression in (7.52) for the local gradient in the quadratic approximation. We see that the result in (7.77) is equivalent to that found in (7.62). Thus, we can replace the explicit evaluation of α_j by a numerical procedure involving a line minimization along the search direction $\mathbf{d}_j$.

We have seen that, for a quadratic error function, the conjugate gradient algorithm finds the minimum after at most W line minimizations, without calculating the Hessian matrix. This clearly represents a significant improvement on the simple gradient descent approach which could take a very large number of steps to minimize even a quadratic error function. In practice, the error function may be far from quadratic. The algorithm therefore generally needs to be run for many iterations until a sufficiently small error is obtained or until some other termination criterion is reached. During the running of the algorithm, the conjugacy of the search directions tends to deteriorate, and so it is common practice to restart the algorithm after every W steps by resetting the search vector to the negative gradient direction. More sophisticated restart procedures are described in Powell (1977).

The conjugate gradient algorithm has been derived on the assumption of a

quadratic error function with a positive-definite Hessian matrix. For general non-linear error functions, the local Hessian matrix need not be positive definite. The search directions defined by the conjugate gradient algorithm need not then be descent directions (Shanno, 1978). In practice, the use of robust line minimization techniques ensures that the error can not increase at any step, and such algorithms are generally found to have good performance in real applications.

As we have seen, the conjugate gradient algorithm provides a minimization technique which requires only the evaluation of the error function and its derivatives, and which, for a quadratic error function, is guaranteed to find the minimum in at most W steps. Since the derivation has been relatively complex, we now summarize the key steps of the algorithm:

1. Choose an initial weight vector $\mathbf{w}_1$.
2. Evaluate the gradient vector $\mathbf{g}_1$, and set the initial search direction $\mathbf{d}_1 = -\mathbf{g}_1$.
3. At step j, minimize $E(\mathbf{w}_j + \alpha \mathbf{d}_j)$ with respect to α to give $\mathbf{w}_{j+1} = \mathbf{w}_j + \alpha_{\min} \mathbf{d}_j$.
4. Test to see if the stopping criterion is satisfied.
5. Evaluate the new gradient vector $\mathbf{g}_{j+1}$.
6. Evaluate the new search direction using (7.67) in which β_j is given by the Hestenes–Stiefel formula (7.72), the Polak–Ribiere formula (7.74) or the Fletcher–Reeves formula (7.75).
7. Set $j = j + 1$ and go to 3.

Empirical results from the training of multi-layer perceptron networks using conjugate gradients can be found in Watrous (1987), Webb *et al.* (1988), Kramer and Sangiovanni-Vincentelli (1989), Makram-Ebeid *et al.* (1989), Barnard (1992) and Johansson *et al.* (1992).

The batch form of gradient descent with momentum, discussed in Section 7.5, involves two arbitrary parameters λ and μ, where λ determines the step length, and μ controls the momentum, i.e. the fraction of the previous step to be included in the current step. A major problem with gradient descent is how to determine values for λ and μ, particularly since the optimum values will typically vary from one iteration to the next. The conjugate gradient method can be regarded as a form of gradient descent with momentum, in which the parameters λ and μ are determined automatically at each iteration. The effective learning rate is determined by line minimization, while the momentum is determined by the parameter β_j in (7.72), (7.74) or (7.75) since this controls the search direction through (7.67).

7.8 Scaled conjugate gradients

We have seen how the use of a line search allows the step size in the conjugate gradient algorithm to be chosen without having to evaluate the Hessian matrix. However, the line search itself introduces some problems. In particular, every line minimization involves several error function evaluations, each of which is computationally expensive. Also, the line-search procedure itself necessarily involves

some parameter whose value determines the termination criterion for each line search. The overall performance of the algorithm can be sensitive to the value of this parameter since a line search which is insufficiently accurate implies that the value of α_j is not being determined correctly, while, an excessively accurate line search can represent a good deal of wasted computation.

Møller (1993b) introduced the *scaled conjugate gradient* algorithm as a way of avoiding the line-search procedure of conventional conjugate gradients. First, note that the Hessian matrix enters the formula (7.62) for α_j only in the form of the Hessian multiplied by a vector $\mathbf{d}_j$. We saw in Section 4.10.7 that, for the multi-layer perceptron, and indeed for more general networks, the product of the Hessian with an arbitrary vector could be computed efficiently, in $\mathcal{O}(W)$ steps (per training pattern), by using central differences or, more accurately, by using the $\mathcal{R}\{\cdot\}$-operator technique.

This suggests that, instead of using line minimization, which typically involves several error function evaluations, each of which takes $\mathcal{O}(W)$ operations, we simply evaluate $\mathbf{H}\mathbf{d}_j$ using the methods of Section 4.10.7. This simple approach fails, however, because, in the case of a non-quadratic error function, the Hessian matrix need not be positive definite. In this case, the denominator in (7.62) can become negative, and the weight update can lead to an increase in the value of the error function. The problem can be overcome by modifying the Hessian matrix to ensure that it is positive definite. This is achieved by adding to the Hessian some multiple of the unit matrix, so that the Hessian becomes

$$\mathbf{H} + \lambda \mathbf{I} \tag{7.78}$$

where $\mathbf{I}$ is the unit matrix, and $\lambda \geq 0$ is a scaling coefficient. Provided λ is sufficiently large, this modified Hessian is guaranteed to be positive definite. The formula for the step length is then given by

$$\alpha_j = -\frac{\mathbf{d}_j^{\mathrm{T}} \mathbf{g}_j}{\mathbf{d}_j^{\mathrm{T}} \mathbf{H}_j \mathbf{d}_j + \lambda_j \|\mathbf{d}_j\|^2} \tag{7.79}$$

where the suffix j on λ_j reflects the fact that the optimum value for this parameter can vary from one iteration to the next. For large values of λ_j the step size becomes small. Techniques such as this are well known in standard optimization theory, where they are called *model trust region* methods, because the model is effectively only trusted in a small region around the current search point. The size of the trust region is governed by the parameter λ_j, so that for large λ_j the trust region is small. The model-trust-region technique is considered in more detail in the context of the Levenberg–Marquardt algorithm later in this chapter.

We now have to find a way to choose an appropriate value for λ_j. From the discussion in Section 7.7.2 we know that the expression (7.79) with $\lambda_j = 0$ will move the weight vector to the minimum along the search direction provided (i) the error function can be represented by a quadratic form, and (ii) the denomi-

nator is positive (corresponding to a positive-definite Hessian). If either of these conditions is not satisfied then the value of λ_j needs to be increased accordingly.

Consider first the problem of a Hessian which is not positive definite. The denominator in the expression (7.79) for the α_j can be written as

$$\delta_j = \mathbf{d}_j^{\mathrm{T}}\mathbf{H}_j\mathbf{d}_j + \lambda_j\|\mathbf{d}_j\|^2. \tag{7.80}$$

For a positive-definite Hessian we have $\delta_j > 0$. If, however, $\delta_j < 0$ then we can increase the value of λ_j in order to make $\delta_j > 0$. Let the raised value of λ_j be called $\overline{\lambda}_j$. Then the corresponding raised value of δ_j is given by

$$\overline{\delta}_j = \delta_j + (\overline{\lambda}_j - \lambda_j)\|\mathbf{d}_j\|^2. \tag{7.81}$$

This will be positive if $\overline{\lambda}_j > \lambda_j - \delta_j/\|\mathbf{d}_j\|^2$. Møller (1993b) chooses to set

$$\overline{\lambda}_j = 2\left(\lambda_j - \frac{\delta_j}{\|\mathbf{d}_j\|^2}\right). \tag{7.82}$$

Substituting (7.82) into (7.81) gives

$$\overline{\delta}_j = -\delta_j + \lambda_j\|\mathbf{d}_j\|^2 = -\mathbf{d}_j^{\mathrm{T}}\mathbf{H}_j\mathbf{d}_j \tag{7.83}$$

which is therefore now positive. This value is used as the denominator in (7.79) to compute the value of the step-size parameter α_j.

We now consider the effects of the local quadratic assumption. In regions where the quadratic approximation is good, the value of λ_j should be reduced, while if the quadratic approximation is poor, λ_j should be increased, so that the size of the trust region reflects the accuracy of the local quadratic approximation. This can be achieved by considering the comparison parameter defined by (Fletcher, 1987)

$$\Delta_j = \frac{E(\mathbf{w}_j) - E(\mathbf{w}_j + \alpha_j\mathbf{d}_j)}{E(\mathbf{w}_j) - E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j)} \tag{7.84}$$

where $E_Q(\mathbf{w})$ is the local quadratic approximation to the error function in the neighbourhood of the point $\mathbf{w}_j$, given by

$$E_Q(\mathbf{w}_j + \alpha_j\mathbf{d}_j) = E(\mathbf{w}_j) + \alpha_j\mathbf{d}_j^{\mathrm{T}}\mathbf{g}_j + \frac{1}{2}\alpha_j^2\mathbf{d}_j^{\mathrm{T}}\mathbf{H}_j\mathbf{d}_j. \tag{7.85}$$

From (7.84) we see that Δ_j gives a measure of the accuracy of the quadratic approximation. If Δ_j is close to 1 then the approximation is a good one and the value of λ_j can be decreased. Conversely a small value of Δ_j is an indication that λ_j should be increased. Substituting (7.85) into (7.84), and using the definition

(7.62) for α_j, we obtain

$$\Delta_j = \frac{2\{E(\mathbf{w}_j) - E(\mathbf{w}_j + \alpha_j \mathbf{d}_j)\}}{\alpha_j \mathbf{d}_j^{\mathrm{T}} \mathbf{g}_j}. \tag{7.86}$$

The value of λ_j can then be adjusted using the following prescription (Fletcher, 1987):

$$\text{if } \Delta_j > 0.75 \text{ then } \lambda_{j+1} = \lambda_j / 2 \tag{7.87}$$

$$\text{if } \Delta_j < 0.25 \text{ then } \lambda_{j+1} = 4\lambda_j \tag{7.88}$$

otherwise set $\lambda_{j+1} = \lambda_j$. Note that, if $\Delta_j < 0$ so that the step would actually lead to an increase in the error, then the weights are not updated, but instead the value of λ_j is increased in accordance with (7.88), and Δ_j is re-evaluated. Eventually an error decrease will be obtained since, for sufficiently large λ_j, the algorithm will be taking a small step in the direction of the negative gradient. The two stages of increasing λ_j (if necessary) to ensure that $\bar{\delta}_j$ is positive, and adjusting λ_j according to the validity of the local quadratic approximation, are applied in succession after each weight update.

Detailed step-by-step descriptions of the algorithm can be found in Møller (1993b) and Williams (1991). Results from software simulations indicate that this algorithm can sometimes offer a significant improvement in speed compared to conventional conjugate gradient algorithms.

7.9 Newton's method

In the conjugate gradient algorithm, implicit use was made of second-order information about the error surface, represented by the local Hessian matrix. We now turn to a class of algorithms which make explicit use of the Hessian.

Using the local quadratic approximation, we can obtain directly an expression for the location of the minimum (or more generally the stationary point) of the error function. From (7.10) the gradient at any point $\mathbf{w}$ is given by

$$\mathbf{g} = \nabla E = \mathbf{H}(\mathbf{w} - \mathbf{w}^*) \tag{7.89}$$

and so the weight vector $\mathbf{w}^*$ corresponding to the minimum of the error function satisfies

$$\mathbf{w}^* = \mathbf{w} - \mathbf{H}^{-1}\mathbf{g}. \tag{7.90}$$

The vector $-\mathbf{H}^{-1}\mathbf{g}$ is known as the *Newton direction* or the *Newton step*, and forms the basis for a variety of optimization strategies. Unlike the local gradient vector, the Newton direction for a quadratic error surface, evaluated at any $\mathbf{w}$, points directly at the minimum of the error function, as illustrated in Figure 7.13.

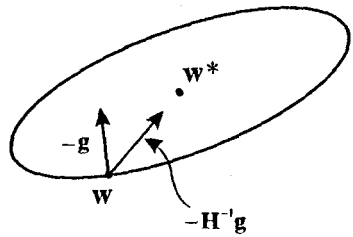

Figure 7.13. Illustration of the Newton direction for a quadratic error surface. The local negative gradient vector $-\mathbf{g}(\mathbf{w})$ does not in general point towards the minimum of the error function, whereas the Newton direction $-\mathbf{H}^{-1}\mathbf{g}(\mathbf{w})$ does.

Since the quadratic approximation used to obtain (7.90) is not exact it would be necessary to apply (7.90) iteratively, with the Hessian being re-evaluated at each new search point. From (7.90), we see that the gradient descent procedure (7.23) corresponds to one step of the Newton formula (7.90), with the inverse Hessian approximated by the unit matrix times η, where η is the learning rate parameter.

There are several difficulties with such an approach, however. First, the exact evaluation of the Hessian for non-linear networks is computationally demanding, since it requires $\mathcal{O}(NW^2)$ steps, where W is the number of weights in the network and N is the number of patterns in the data set. This evaluation would be prohibitively expensive if done at each stage of an iterative algorithm. Second, the Hessian must be inverted, which requires $\mathcal{O}(W^3)$ steps, and so is also computationally demanding. Third, the Newton step in (7.90) may move towards a maximum or a saddlepoint rather than a minimum. This occurs if the Hessian is not positive definite, so that there exist directions of negative curvature. Thus, the error is not guaranteed to be reduced at each iteration. Finally, the step size predicted by (7.90) may be sufficiently large that it takes us outside the range of validity of the quadratic approximation. In this case the algorithm could become unstable.

Nevertheless, by making various modifications to the full Newton rule it can be turned into a practical optimization method. Note first that, if the Hessian is positive definite (as is the case close to a minimum), then the Newton direction always represents a descent direction, as can be seen by considering the local directional derivative of the error function in the Newton direction evaluated at some point $\mathbf{w}$

$$\frac{\partial}{\partial \lambda} E(\mathbf{w} + \lambda \mathbf{d})\bigg|_{\lambda=0} = \mathbf{d}^{\mathrm{T}}\mathbf{g} = -\mathbf{g}^{\mathrm{T}}\mathbf{H}^{-1}\mathbf{g} < 0 \tag{7.91}$$

where we have used the Newton step formula $\mathbf{d} = -\mathbf{H}^{-1}\mathbf{g}$.

Away from the neighbourhood of a minimum, the Hessian matrix need not be positive definite. The problem can be resolved by adopting the *model trust region* approach, discussed earlier in Section 7.8, and described in more detail in Section 7.11. This involves adding to the Hessian a positive-definite symmetric matrix which comprises the unit matrix $\mathbf{I}$ times a constant factor λ. Provided λ is sufficiently large, the new matrix

$$\mathbf{H} + \lambda\mathbf{I} \tag{7.92}$$

will be positive definite. The corresponding step direction is a compromise between the Newton direction and the negative gradient direction. For very small values of λ we recover the Newton direction, while for large values of λ the direction approximates the negative gradient

$$-(\mathbf{H} + \lambda\mathbf{I})^{-1}\mathbf{g} \simeq -\frac{1}{\lambda}\mathbf{g}. \tag{7.93}$$

This still leaves the problem of computing and inverting the Hessian matrix. One approach is to approximate the Hessian by neglecting the off-diagonal terms (Becker and Le Cun, 1989; Ricotti *et al.*, 1988). This has the advantages that the inverse of the Hessian is trivial to compute, and the Newton update equations (7.90) decouple into separate equations for each weight. The problem of negative curvatures is dealt with by the simple heuristic of taking the modulus of the second derivative. This gives a Newton update for a weight w_i in the form

$$\Delta w_i = -\left(\left|\frac{\partial^2 E}{\partial w_i^2}\right| + \lambda\right)^{-1} \frac{\partial E}{\partial w_i} \tag{7.94}$$

where λ is treated as a small positive constant. For the multi-layer perceptron, the diagonal terms in the Hessian matrix can be computed by a back-propagation procedure as discussed in Section 4.10.1. A major drawback of this approach, however, is that the Hessian matrix for many neural network problems is typically far from diagonal.

7.10 Quasi-Newton methods

We have already argued that a direct application of the Newton method, as given by (7.90), would be computationally prohibitive since it would require $\mathcal{O}(NW^2)$ operations to evaluate the Hessian matrix and $\mathcal{O}(W^3)$ operations to compute its inverse. Alternative approaches, known as *quasi-Newton* or *variable metric* methods, are based on (7.90), but instead of calculating the Hessian directly, and then evaluating its inverse, they build up an approximation to the inverse Hessian over a number of steps. As with conjugate gradients, these methods can find the minimum of a quadratic form in at most W steps, giving an overall

computational cost which is $\mathcal{O}(NW^2)$.

The quasi-Newton approach involves generating a sequence of matrices $\mathbf{G}^{(\tau)}$ which represent increasingly accurate approximations to the inverse Hessian $\mathbf{H}^{-1}$, using only information on the first derivatives of the error function. The problems arising from Hessian matrices which are not positive definite are solved by starting from a positive-definite matrix (such as the unit matrix) and ensuring that the update procedure is such that the approximation to the inverse Hessian is guaranteed to remain positive definite.

From the Newton formula (7.90) we see that the weight vectors at steps τ and $\tau + 1$ are related to the corresponding gradients by

$$\mathbf{w}^{(\tau+1)} - \mathbf{w}^{(\tau)} = -\mathbf{H}^{-1}(\mathbf{g}^{(\tau+1)} - \mathbf{g}^{(\tau)}) \tag{7.95}$$

which is known as the quasi-Newton condition. The approximation $\mathbf{G}$ of the inverse Hessian is constructed so as to satisfy this condition also.

The two most commonly used update formulae are the *Davidson–Fletcher–Powell* (DFP) and the *Broyden–Fletcher–Goldfarb–Shanno* (BFGS) procedures. Here we give only the BFGS expression, since this is generally regarded as being superior:

$$\mathbf{G}^{(\tau+1)} = \mathbf{G}^{(\tau)} + \frac{\mathbf{p}\mathbf{p}^{\mathrm{T}}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}} - \frac{(\mathbf{G}^{(\tau)}\mathbf{v})\mathbf{v}^{\mathrm{T}}\mathbf{G}^{(\tau)}}{\mathbf{v}^{\mathrm{T}}\mathbf{G}^{(\tau)}\mathbf{v}} + (\mathbf{v}^{\mathrm{T}}\mathbf{G}^{(\tau)}\mathbf{v})\mathbf{u}\mathbf{u}^{\mathrm{T}} \tag{7.96}$$

where we have defined the following vectors:

$$\mathbf{p} = \mathbf{w}^{(\tau+1)} - \mathbf{w}^{(\tau)} \tag{7.97}$$

$$\mathbf{v} = \mathbf{g}^{(\tau+1)} - \mathbf{g}^{(\tau)} \tag{7.98}$$

$$\mathbf{u} = \frac{\mathbf{p}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}} - \frac{\mathbf{G}^{(\tau)}\mathbf{v}}{\mathbf{v}^{\mathrm{T}}\mathbf{G}^{(\tau)}\mathbf{v}}. \tag{7.99}$$

Derivations of this expression can be found in many standard texts on optimization methods such as Polak (1971), or Luenberger (1984). It is straightforward to verify by direct substitution that (7.96) does indeed satisfy the quasi-Newton condition (7.95).

Initializing the procedure using the identity matrix corresponds to taking the first step in the direction of the negative gradient. At each step of the algorithm, the direction $-\mathbf{G}\mathbf{g}$ is guaranteed to be a descent direction, since the matrix $\mathbf{G}$ is positive definite. However, the full Newton step given by (7.90) may take the search outside the range of validity of the quadratic approximation. The solution is to use a line-search algorithm (Section 7.6), as used with conjugate gradients, to find the minimum of the error function along the search direction. Thus, the weight vector is updated using

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \alpha^{(\tau)}\mathbf{G}^{(\tau)}\mathbf{g}^{(\tau)} \tag{7.100}$$

where $\alpha^{(\tau)}$ is found by line minimization.

A significant advantage of the quasi-Newton approach over the conjugate gradient method is that the line search does not need to be performed with such great accuracy since it does not form a critical factor in the algorithm. For conjugate gradients, the line minimizations need to be performed accurately in order to ensure that the system of conjugate directions and orthogonal gradients is set up correctly.

A potential disadvantage of the quasi-Newton method is that it requires the storage and update of a matrix $\mathbf{G}$ of size $W \times W$. For small networks this is of little consequence, but for networks with more than a few thousand weights it could lead to prohibitive memory requirements. In such cases, techniques such as conjugate gradients, which require only $\mathcal{O}(W)$ storage, have a significant advantage.

For an W-dimensional quadratic form, the sequence of matrices $\mathbf{G}^{(\tau)}$ is guaranteed to converge exactly to the true Hessian after W steps, and the quasi-Newton algorithm would find the exact minimum of the quadratic form after W steps, assuming the line minimizations were performed exactly. Results from the application of quasi-Newton methods to the training of neural networks can be found in Watrous (1987), Webb *et al.* (1988), and Barnard (1992).

7.10.1 *Limited memory quasi-Newton methods*

Shanno (1978) investigated the accuracy needed for line searches in both conjugate gradient and quasi-Newton algorithms, and concluded that conjugate gradient algorithms require relatively accurate line searches, while quasi-Newton methods remain robust even if the line searches are only performed to relatively low accuracy. This implies that, for conjugate gradient methods, significant computational effort needs to be expended on each line minimization.

The advantage of conjugate gradient algorithms, however, is that they require $\mathcal{O}(W)$ storage rather than the $\mathcal{O}(W^2)$ storage needed by quasi-Newton methods. The question therefore arises as to whether we can find an algorithm which uses $\mathcal{O}(W)$ storage but which does not require accurate line searches (Shanno, 1978). One way to reduce the storage requirement of quasi-Newton methods is to replace the approximate inverse Hessian matrix $\mathbf{G}$ at each step by the unit matrix. If we make this substitution into the BFGS formula in (7.96), and multiply the resulting approximate inverse Hessian by the current gradient $\mathbf{g}^{(\tau+1)}$, we obtain the following expression for the search direction

$$\mathbf{d}^{(\tau+1)} = -\mathbf{g}^{(\tau+1)} + A\mathbf{p} + B\mathbf{v} \tag{7.101}$$

where the scalars A and B are defined by

$$A = -\left(1 + \frac{\mathbf{v}^{\mathrm{T}}\mathbf{v}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}}\right)\frac{\mathbf{p}^{\mathrm{T}}\mathbf{g}^{(\tau+1)}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}} + \frac{\mathbf{v}^{\mathrm{T}}\mathbf{g}^{(\tau+1)}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}} \tag{7.102}$$

$$B = \frac{\mathbf{p}^{\mathrm{T}}\mathbf{g}^{(\tau+1)}}{\mathbf{p}^{\mathrm{T}}\mathbf{v}} \tag{7.103}$$

and the vectors $\mathbf{p}$ and $\mathbf{v}$ are defined in (7.97) and (7.98). If exact line searches are performed, then (7.101) produces search directions which are mutually conjugate (Shanno, 1978). The difference compared with standard conjugate gradients is that if approximate line searches are used, the algorithm remains well behaved. As with conjugate gradients, the algorithm is restarted in the direction of the negative gradient every W steps. This is known as the *limited memory BFGS* algorithm, and has been applied to the problem of neural network training by Battiti (1989).

7.11 The Levenberg–Marquardt algorithm

Many of the optimization algorithms we have discussed up to now have been general-purpose methods designed to work with a wide range of error functions. We now describe an algorithm designed specifically for minimizing a sum-of-squares error.

Consider the sum-of-squares error function in the form

$$E = \frac{1}{2}\sum_n (\epsilon^n)^2 = \frac{1}{2}\|\boldsymbol{\epsilon}\|^2 \tag{7.104}$$

where ϵ^n is the error for the nth pattern, and $\boldsymbol{\epsilon}$ is a vector with elements ϵ^n. Suppose we are currently at a point $\mathbf{w}_{\mathrm{old}}$ in weight space and we move to a point $\mathbf{w}_{\mathrm{new}}$. If the displacement $\mathbf{w}_{\mathrm{new}} - \mathbf{w}_{\mathrm{old}}$ is small then we can expand the error vector $\boldsymbol{\epsilon}$ to first order in a Taylor series

$$\boldsymbol{\epsilon}(\mathbf{w}_{\mathrm{new}}) = \boldsymbol{\epsilon}(\mathbf{w}_{\mathrm{old}}) + \mathbf{Z}(\mathbf{w}_{\mathrm{new}} - \mathbf{w}_{\mathrm{old}}) \tag{7.105}$$

where we have defined the matrix $\mathbf{Z}$ with elements

$$(\mathbf{Z})_{ni} \equiv \frac{\partial \epsilon^n}{\partial w_i}. \tag{7.106}$$

The error function (7.104) can then be written as

$$E = \frac{1}{2}\left\|\boldsymbol{\epsilon}(\mathbf{w}_{\mathrm{old}}) + \mathbf{Z}(\mathbf{w}_{\mathrm{new}} - \mathbf{w}_{\mathrm{old}})\right\|^2. \tag{7.107}$$

If we minimize this error with respect to the new weights $\mathbf{w}_{\mathrm{new}}$ we obtain

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - (\mathbf{Z}^{\mathrm{T}}\mathbf{Z})^{-1}\mathbf{Z}^{\mathrm{T}}\boldsymbol{\epsilon}(\mathbf{w}_{\text{old}}). \tag{7.108}$$

Note that this has the same structure as the pseudo-inverse formula for linear networks introduced in Section 3.4.3, as we would expect, since we are indeed minimizing a sum-of-squares error function for a linear model.

For the sum-of-squares error function (7.104), the elements of the Hessian matrix take the form

$$(\mathbf{H})_{ik} = \frac{\partial^2 E}{\partial w_i \partial w_k} = \sum_n \left\{ \frac{\partial \epsilon^n}{\partial w_i}\frac{\partial \epsilon^n}{\partial w_k} + \epsilon^n \frac{\partial^2 \epsilon^n}{\partial w_i \partial w_k} \right\}. \tag{7.109}$$

If we neglect the second term, then the Hessian can be written in the form

$$\mathbf{H} = \mathbf{Z}^{\mathrm{T}}\mathbf{Z}. \tag{7.110}$$

For a linear network (7.110) is exact. We therefore see that (7.108) involves the inverse Hessian, as we might expect since it corresponds to the Newton step applied to the linearized model in (7.105). For non-linear networks it represents an approximation, although we note that in the limit of an infinite data set the expression (7.110) is exact at the global minimum of the error function, as discussed in Section 6.1.4. Recall that in this approximation the Hessian is relatively easy to compute, since first derivatives with respect to network weights can be obtained very efficiently using back-propagation as shown in Section 4.8.3.

In principle, the update formula (7.108) could be applied iteratively in order to try to minimize the error function. The problem with such an approach is that the step size which is given by (7.108) could turn out to be relatively large, in which case the linear approximation (7.107) on which it is based would no longer be valid. In the *Levenberg–Marquardt* algorithm (Levenberg, 1944; Marquardt, 1963), this problem is addressed by seeking to minimize the error function while at the same time trying to keep the step size small so as to ensure that the linear approximation remains valid. This is achieved by considering a modified error function of the form

$$\widetilde{E} = \frac{1}{2}\|\boldsymbol{\epsilon}(\mathbf{w}_{\text{old}}) + \mathbf{Z}(\mathbf{w}_{\text{new}} - \mathbf{w}_{\text{old}})\|^2 + \lambda\|\mathbf{w}_{\text{new}} - \mathbf{w}_{\text{old}}\|^2 \tag{7.111}$$

where the parameter λ governs the step size. For large values of λ the value of $\|\mathbf{w}_{\text{new}} - \mathbf{w}_{\text{old}}\|^2$ will tend to be small. If we minimize the modified error (7.111) with respect to $\mathbf{w}_{\text{new}}$, we obtain

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - (\mathbf{Z}^{\mathrm{T}}\mathbf{Z} + \lambda\mathbf{I})^{-1}\mathbf{Z}^{\mathrm{T}}\boldsymbol{\epsilon}(\mathbf{w}_{\text{old}}) \tag{7.112}$$

where $\mathbf{I}$ is the unit matrix. For very small values of the parameter λ we recover the Newton formula, while for large values of λ we recover standard gradient

descent. In this latter case the step length is determined by λ^{-1}, so that it is clear that, for sufficiently large values of λ, the error will necessarily decrease since (7.112) then generates a very small step in the direction of the negative gradient. The Levenberg–Marquardt algorithm is an example of a *model trust region* approach in which the model (in this case the linearized approximation for the error function) is trusted only within some region around the current search point. The size of this region is governed by the value of λ.

In practice a value must be chosen for λ and this value should vary appropriately during the minimization process. One common approach for setting λ is to begin with some arbitrary value such as $\lambda = 0.1$, and at each step monitor the change in error E. If the error decreases after taking the step predicted by (7.112) the new weight vector is retained, the value of λ is decreased by a factor of 10, and the process repeated. If, however, the error increases, then λ is increased by a factor of 10, the old weight vector is restored, and a new weight update computed. This is repeated until a decrease in E is obtained. Comparisons of the Levenberg–Marquardt algorithm with other methods for training multi-layer perceptrons are given in Webb *et al.* (1988).

Exercises

7.1 ($\star$) Show that the stationary point $\mathbf{w}^*$ of quadratic error surface of the form (7.10) is a unique global minimum if, and only if, the Hessian matrix is positive definite, so that all of its eigenvalues are positive.

7.2 ($\star\star$) Consider a quadratic error error function in two-dimensions of the form

$$E = \frac{1}{2}\lambda_1 w_1^2 + \frac{1}{2}\lambda_2 w_2^2 \tag{7.113}$$

Verify that λ_1 and λ_2 are the eigenvalues of the Hessian matrix. Write a numerical implementation of the gradient descent algorithm, and apply it to the minimization of this error function for the case where the ratio of the eigenvalues λ_2/λ_1 is large (say 10:1). Explore the convergence properties of the algorithm for various values of the learning rate parameter, and verify that the largest value of η which still leads to a reduction in E is determined by the ratio of the two eigenvalues, as discussed in Section 7.5.1. Now include a momentum term and explore the convergence behaviour as a function of both the learning rate and momentum parameters. For each experiment, plot trajectories of the evolution of the weight vector in the two-dimensional weight space, superimposed on contours of constant error.

7.3 ($\star$) Take the continuous-time limit of (7.33) and show that leads to the following equation of motion

$$m\frac{d^2\mathbf{w}}{d\tau^2} + \nu\frac{d\mathbf{w}}{d\tau} = -\nabla E \tag{7.114}$$

where

$$m = \frac{\mu\Delta^2}{\eta^2}, \qquad \nu = \frac{(1-\mu)\Delta}{\eta} \tag{7.115}$$

and τ is the continuous time variable. The equation of motion (7.114) corresponds to the motion of a massive particle (i.e. one having inertia) with mass m moving downhill under a force $-\nabla E$, subject to viscous drag with viscosity coefficient ν. This is the origin of the term 'momentum' in (7.33).

7.4 (⋆) In (7.35) we considered the effect of a momentum term on gradient descent through a region of weight space in which the error function gradient could be taken to be approximately constant. This was based on summing an arithmetic series after an infinite number of steps. Repeat this analysis more carefully for a finite number L of steps, by expressing the resulting finite series as the difference of two infinite series. Hence obtain an expression for the weight vector $\mathbf{w}^{(L)}$ in terms of the initial weight vector $\mathbf{w}^{(0)}$, the error gradient ∇E (assumed constant) and the parameters η and μ. Show that (7.35) is obtained in the limit $L \to \infty$.

7.5 (⋆) Consider an arbitrary vector $\mathbf{v}$ and suppose that we first normalize $\mathbf{v}$ so that $\|\mathbf{v}\| = 1$ and then multiply the resulting vector by a real symmetric matrix $\mathbf{H}$. Show that, if this process of normalization and multiplication by $\mathbf{H}$ is repeated many times, the resulting vector will converge towards $\lambda_{\max}\mathbf{u}_{\max}$ where $\lambda_{\max}$ is the largest eigenvalue of $\mathbf{H}$ and $\mathbf{u}_{\max}$ is the corresponding eigenvector. (Assume that the initial vector $\mathbf{v}$ is not orthogonal to $\mathbf{u}_{\max}$).

7.6 (⋆) Consider a single-layer network having a mapping function given by

$$y_k = \sum_i w_{ki} x_i \tag{7.116}$$

and a sum-of-squares error function of the form

$$E = \frac{1}{2}\sum_n \sum_k (y_k^n - t_k^n)^2 \tag{7.117}$$

with n labels the patterns, and k labels the output units. Suppose the weights are updated by a gradient descent rule in which each weight w_{ki} has its own learning rate parameter η_{ki}, so that the value of w_{ki} at time step τ is given by

$$w_{ki}^{(\tau)} = w_{ki}^{(\tau-1)} - \eta_{ki}^{(\tau)} \frac{\partial E}{\partial w_{ki}^{(\tau-1)}}. \tag{7.118}$$

Use the above equations to find an expression for the error at step τ in terms of the weight values at step $\tau - 1$ and the learning rate parameters $\eta_{ki}^{(\tau)}$. Show that the derivative of the error function with respect to $\eta_{ki}^{(\tau)}$ is given by the delta-delta expression

$$\frac{\partial E}{\partial \eta_{ki}^{(\tau)}} = -g_{ki}^{(\tau)} g_{ki}^{(\tau-1)} \tag{7.119}$$

where

$$g_{ki}^{(\tau)} \equiv \frac{\partial E}{\partial w_{ki}^{(\tau)}}. \tag{7.120}$$

7.7 (⋆) Derive the quickprop weight update formula (7.42) by following the discussion given in the text.

7.8 (⋆) Consider a symmetric, positive-definite $W \times W$ matrix $\mathbf{H}$, and suppose there exists a set of W mutually conjugate directions $\mathbf{d}_i$ satisfying

$$\mathbf{d}_j^{\mathrm{T}} \mathbf{H} \mathbf{d}_i = 0, \qquad j \neq i. \tag{7.121}$$

Show that the vectors $\mathbf{d}_i$ must be linearly independent (i.e. that $\mathbf{d}_i$ cannot be expressed as a linear combination of $\{\mathbf{d}_j\}$ where $j = 1, \ldots, W$ with $j \neq i$).

7.9 (⋆) The purpose of this exercise is to show by induction that if successive search directions are constructed from (7.67) using the conjugacy condition (7.68), that the first W such directions will all be mutually conjugate. We know by construction that $\mathbf{d}_2^{\mathrm{T}} \mathbf{H} \mathbf{d}_1 = 0$. Now suppose that $\mathbf{d}_j^{\mathrm{T}} \mathbf{H} \mathbf{d}_i = 0$ for some given $j < W$ and for all i satisfying $i < j$. Since $\mathbf{d}_{j+1}^{\mathrm{T}} \mathbf{H} \mathbf{d}_j = 0$ by construction, we need to show that $\mathbf{d}_{j+1}^{\mathrm{T}} \mathbf{H} \mathbf{d}_i = 0$ for all $i < j + 1$. Using (7.67) we have

$$\mathbf{d}_{j+1}^{\mathrm{T}} \mathbf{H} \mathbf{d}_i = -\mathbf{g}_{j+1}^{\mathrm{T}} \mathbf{H} \mathbf{d}_i + \beta_j \mathbf{d}_j^{\mathrm{T}} \mathbf{H} \mathbf{d}_i. \tag{7.122}$$

The second term in (7.122) vanishes by assumption. Show that the first term also vanishes, by making use of (7.63) and (7.71). This completes the proof.

7.10 (⋆) Verify by direct substitution that the BFGS update formula (7.96) satisfies the Newton condition (7.95).

7.11 (⋆) Verify that replacement of the approximate inverse Hessian matrix $\mathbf{G}^{(\tau)}$ by the unit matrix $\mathbf{I}$ in the BFGS formula (7.96) leads to a Newton step $-\mathbf{G}^{(\tau+1)}\mathbf{g}$ given by the limited memory BFGS expression (7.101).