4

THE MULTI-LAYER PERCEPTRON

In Chapter 3, we discussed the properties of networks having a single layer of adaptive weights. Such networks have a number of important limitations in terms of the range of functions which they can represent. To allow for more general mappings we might consider successive transformations corresponding to networks having several layers of adaptive weights. In fact we shall see that networks with just two layers of weights are capable of approximating any continuous functional mapping. More generally we can consider arbitrary network diagrams (not necessarily having a simple layered structure) since any network diagram can be converted into its corresponding mapping function. The only restriction is that the diagram must be *feed-forward*, so that it contains no feedback loops. This ensures that the network outputs can be calculated as explicit functions of the inputs and the weights.

We begin this chapter by reviewing the representational capabilities of multi-layered networks having either threshold or sigmoidal activation functions. Such networks are generally called *multi-layer perceptrons*, even when the activation functions are sigmoidal. For networks having differentiable activation functions, there exists a powerful and computationally efficient method, called *error back-propagation*, for finding the derivatives of an error function with respect to the weights and biases in the network. This is an important feature of such networks since these derivatives play a central role in the majority of training algorithms for multi-layered networks, and we therefore discuss back-propagation at some length. We also consider a variety of techniques for evaluating and approximating the second derivatives of an error function. These derivatives form the elements of the Hessian matrix, which has a variety of different applications in the context of neural networks.

4.1 Feed-forward network mappings

In the first three sections of this chapter we consider a variety of different kinds of feed-forward network, and explore the limitations which exist on the mappings which they can generate. We are only concerned in this discussion with finding fundamental restrictions on the capabilities of the networks, and so we shall for instance assume that arbitrarily large networks can be constructed if needed. In practice, we must deal with networks of a finite size, and this raises a number of important issues which are discussed in later chapters.

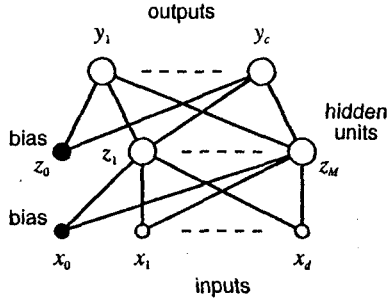

Figure 4.1. An example of a feed-forward network having two layers of adaptive weights. The bias parameters in the first layer are shown as weights from an extra input having a fixed value of $x_0 = 1$. Similarly, the bias parameters in the second layer are shown as weights from an extra hidden unit, with activation again fixed at $z_0 = 1$.

We shall view feed-forward neural networks as providing a general framework for representing non-linear functional mappings between a set of input variables and a set of output variables. This is achieved by representing the non-linear function of many variables in terms of compositions of non-linear functions of a single variable, called activation functions. Each multivariate function can be represented in terms of a network diagram such that there is a one-to-one correspondence between components of the function and the elements of the diagram. Equally, any topology of network diagram, provided it is feed-forward, can be translated into the corresponding mapping function. We can therefore categorize different network functions by considering the structure of the corresponding network diagrams.

4.1.1 *Layered networks*

We begin by looking at networks consisting of successive layers of adaptive weights. As discussed in Chapter 3, single-layer networks are based on a linear combination of the input variables which is transformed by a non-linear activation function. We can construct more general functions by considering networks having successive layers of processing units, with connections running from every unit in one layer to every unit in the next layer, but with no other connections permitted. Such layered networks are easier to analyse theoretically than more general topologies, and can often be implemented more efficiently in a software simulation.

An example of a layered network is shown in Figure 4.1. Note that units which are not treated as output units are called *hidden* units. In this network there are d inputs, M hidden units and c output units. We can write down the analytic function corresponding to Figure 4.1 as follows. The output of the jth

hidden unit is obtained by first forming a weighted linear combination of the d input values, and adding a bias, to give

$$a_j = \sum_{i=1}^{d} w_{ji}^{(1)} x_i + w_{j0}^{(1)}. \tag{4.1}$$

Here $w_{ji}^{(1)}$ denotes a weight in the first layer, going *from* input i *to* hidden unit j, and $w_{j0}^{(1)}$ denotes the bias for hidden unit j. As with the single-layer networks of Chapter 3, we have made the bias terms for the hidden units explicit in the diagram of Figure 4.1 by the inclusion of an extra input variable x_0 whose value is permanently set at $x_0 = 1$. This can be represented analytically by rewriting (4.1) in the form

$$a_j = \sum_{i=0}^{d} w_{ji}^{(1)} x_i. \tag{4.2}$$

The activation of hidden unit j is then obtained by transforming the linear sum in (4.2) using an activation function $g(\cdot)$ to give

$$z_j = g(a_j). \tag{4.3}$$

In this chapter we shall consider two principal forms of activation function given respectively by the Heaviside step function, and by continuous sigmoidal functions, as introduced already in the context of single-layer networks in Section 3.1.3.

The outputs of the network are obtained by transforming the activations of the hidden units using a second layer of processing elements. Thus, for each output unit k, we construct a linear combination of the outputs of the hidden units of the form

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)}. \tag{4.4}$$

Again, we can absorb the bias into the weights to give

$$a_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j \tag{4.5}$$

which can be represented diagrammatically by including an extra hidden unit with activation $z_0 = 1$ as shown in Figure 4.1. The activation of the kth output unit is then obtained by transforming this linear combination using a non-linear

activation function, to give

$$y_k = \widetilde{g}(a_k). \tag{4.6}$$

Here we have used the notation $\widetilde{g}(\cdot)$ for the activation function of the output units to emphasize that this need not be the same function as used for the hidden units.

If we combine (4.2), (4.3), (4.5) and (4.6) we obtain an explicit expression for the complete function represented by the network diagram in Figure 4.1 in the form

$$y_k = \widetilde{g}\left(\sum_{j=0}^{M} w_{kj}^{(2)} g\left(\sum_{i=0}^{d} w_{ji}^{(1)} x_i\right)\right) \tag{4.7}$$

We note that, if the activation functions for the output units are taken to be linear, so that $\widetilde{g}(a) = a$, this functional form becomes a special case of the generalized linear discriminant function discussed in Section 3.3, in which the basis functions are given by the particular functions z_j defined by (4.2) and (4.3). The crucial difference is that here we shall regard the weight parameters appearing in the first layer of the network, as well as those in the second layer, as being adaptive, so that their values can be changed during the process of network training.

The network of Figure 4.1 corresponds to a transformation of the input variables by two successive single-layer networks. It is clear that we can extend this class of networks by considering further successive transformations of the same general kind, corresponding to networks with extra layers of weights. Throughout this book, when we use the term L-layer network we shall be referring to a network with L layers of adaptive weights. Thus we shall call the network of Figure 4.1 a two-layer network, while the networks of Chapter 3 are called single-layer networks. It should be noted, however, that an alternative convention is sometimes also found in the literature. This counts layers of units rather than layers of weights, and regards the inputs as separate units. According to this convention the networks of Chapter 3 would be called two-layer networks, and the network in Figure 4.1 would be said to have three layers. We do not recommend this convention, however, since it is the layers of adaptive weights which are crucial in determining the properties of the network function. Furthermore, the circles representing inputs in a network diagram are not true processing units since their sole purpose is to represent the values of the input variables.

A useful technique for visualization of the weight values in a neural network is the *Hinton diagram*, illustrated in Figure 4.2. Each square in the diagram corresponds to one of the weight or bias parameters in the network, and the squares are grouped into blocks corresponding to the parameters associated with each unit. The size of a square is proportional to the magnitude of the corresponding

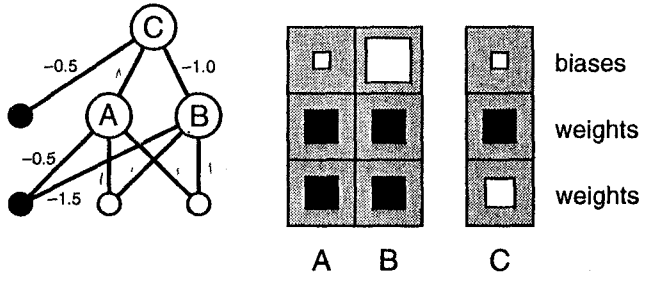

Figure 4.2. Example of a two-layer network which solves the XOR problem, showing the corresponding Hinton diagram. The weights in the network have the value 1.0 unless indicated otherwise.

parameter, and the square is black or white according to whether the parameter is positive or negative.

4.1.2 *General topologies*

Since there is a direct correspondence between a network diagram and its mathematical function, we can develop more general network mappings by considering more complex network diagrams. We shall, however, restrict our attention to the case of *feed-forward* networks. These have the property that there are no feedback loops in the network. In general we say that a network is feed-forward if it is possible to attach successive numbers to the inputs and to all of the hidden and output units such that each unit only receives connections from inputs or units having a smaller number. An example of a general feed-forward network is shown in Figure 4.3. Such networks have the property that the outputs can be expressed as deterministic functions of the inputs, and so the whole network represents a multivariate non-linear functional mapping.

The procedure for translating a network diagram into the corresponding mathematical function follows from a straightforward extension of the ideas already discussed. Thus, the output of unit k is obtained by transforming a weighted linear sum with a non-linear activation function to give

$$z_k = g\left(\sum_j w_{kj} z_j\right) \tag{4.8}$$

where the sum runs over all inputs and units which send connections to unit k (and a bias parameter is included in the summation). For a given set of values applied to the inputs of the network, successive use of (4.8) allows the activations of all units in the network to be evaluated including those of the output units. This process can be regarded as a *forward propagation* of signals through the

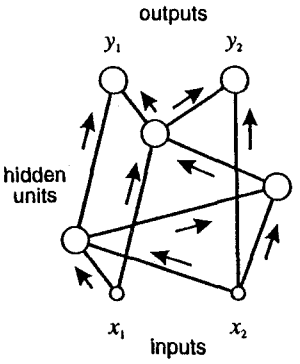

Figure 4.3. An example of a neural network having a general feed-forward topology. Note that each unit has an associated bias parameter, which has been omitted from the diagram for clarity.

network. In practice, there is little call to consider random networks, but there is often considerable advantage in building a lot of structure into the network. An example involving multiple layers of processing units, with highly restricted and structured interconnections between the layers, is discussed in Section 8.7.3.

Note that, if the activation functions of all the hidden units in a network are taken to be linear, then for any such network we can always find an equivalent network without hidden units. This follows from the fact that the composition of successive linear transformations is itself a linear transformation. Note, however, that if the number of hidden units is smaller than either the number of input or output units, then the linear transformation which the network generates is not the most general possible since information is lost in the dimensionality reduction at the hidden units. In Section 8.6.2 it is shown that such networks can be related to conventional data processing techniques such as principal component analysis. In general, however, there is little interest in multi-layer linear networks, and we shall therefore mainly consider networks for which the hidden unit activation functions are non-linear.

4.2 Threshold units

There are many possible choices for the non-linear activation functions in a multi-layered network, and the choice of activation functions for the hidden units may often be different from that for the output units. This is because hidden and output units perform different roles, as is discussed at length in Sections 6.6.1 and 6.7.1. However, we begin by considering networks in which all units have Heaviside, or step, activation functions of the form

$$g(a) = \begin{cases} 0 & \text{when } a < 0 \\ 1 & \text{when } a \geq 0. \end{cases} \tag{4.9}$$

Such units are also known as *threshold* units. We consider separately the cases in which the inputs consist of binary and continuous variables.

4.2.1 *Binary inputs*

Consider first the case of binary inputs, so that $x_i = 0$ or 1. Since the network outputs are also 0 or 1, the network is computing a Boolean function. We can easily show that a two-layer network of the form shown in Figure 4.1 can generate any Boolean function, provided the number M of hidden units is sufficiently large (McCulloch and Pitts, 1943). This can be seen by constructing a specific network which computes a particular (arbitrary) Boolean function. We first note that for d inputs the total possible number of binary patterns which we have to consider is 2^d. A Boolean function is therefore completely specified once we have given the output (0 or 1) corresponding to each of the 2^d possible input patterns. To construct the required network we take one hidden unit for every input pattern which has an output target of 1. We then arrange for each hidden unit to respond just to the corresponding pattern. This can be achieved by setting the weight from an input to a given hidden unit to $+1$ if the corresponding pattern has a 1 for that input, and setting the weight to -1 if the pattern has a 0 for that input. The bias for the hidden unit is set to $1 - b$ where b is the number of non-zero inputs for that pattern. Thus, for any given hidden unit, presentation of the corresponding pattern will generate a summed input of b and the unit will give an output of 1, while any other pattern (including any of the patterns with target 0) will give a summed input of at most $b - 2$ and the unit will have an output of 0. It is now a simple matter to connect each hidden unit to the output unit with a weight $+1$. An output bias of -1 then ensures that the output of the network is correct for all patterns.

This construction is of little practical value, since it merely stores a set of binary relations and has no capability to generalize to new patterns outside the training set (since the training set was exhaustive). It does, however, illustrate the concept of a *template*. Each hidden unit acts as a template for the corresponding input pattern and only generates an output when the input pattern matches the template pattern.

4.2.2 *Continuous inputs*

We now discuss the case of continuous input variables, again for units with threshold activation functions, and we consider the possible decision boundaries which can be produced by networks having various numbers of layers (Lippmann, 1987; Lonstaff and Cross, 1987). In Section 3.1 it was shown that a network with a single layer of weights, and a threshold output unit, has a decision boundary which is a hyperplane. This is illustrated for a two-dimensional input space in Figure 4.4 (a). Now consider networks with two layers of weights. Again, each hidden units divides the input space with a hyperplane, so that it has activation

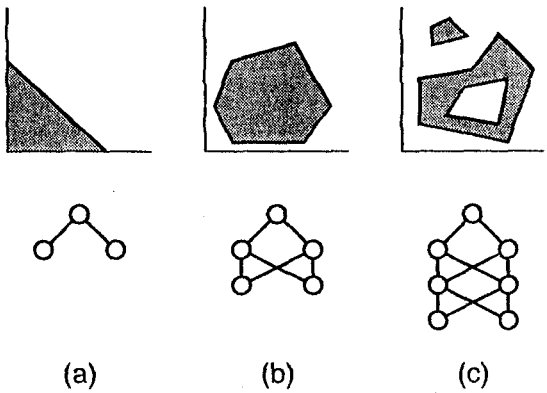

Figure 4.4. Illustration of some possible decision boundaries which can be generated by networks having threshold activation functions and various numbers of layers. Note that, for the two-layer network in (b), a single convex region of the form shown is not the most general possible.

$z = 1$ on one side of the hyperplane, and $z = 0$ on the other side. If there are M hidden units and the bias on the output unit is set to $-M$, then the output unit computes a logical AND of the outputs of the hidden units. In other words, the output unit has an output of 1 only if all of the hidden units have an output of 1. Such a network can generate decision boundaries which surround a single convex region of the input space, whose boundary consists of segments of hyperplanes, as illustrated in Figure 4.4 (b). A convex region is defined to be one for which any line joining two points on the boundary of the region passes only through points which lie inside the region. These are not, however, the most general regions which can be generated by a two-layer network of threshold units, as we shall see shortly.

Networks having three layers of weights can generate arbitrary decision regions, which may be non-convex and disjoint, as illustrated in Figure 4.4 (c). A simple demonstration of this last property can be given as follows (Lippmann, 1987). Consider a particular network architecture in which, instead of having full connectivity between adjacent layers as considered so far, the hidden units are arranged into groups of $2d$ units, where d denotes the number of inputs. The topology of the network is illustrated in Figure 4.5. The units in each group send their outputs to a unit in the second hidden layer associated with that group. Each second-layer unit then sends a connection to the output unit. Suppose the input space is divided into a fine grid of hypercubes, each of which is labelled as class C_1 or C_2. By making the input-space grid sufficiently fine we can approximate an arbitrarily shaped decision boundary as closely as we wish. One group of first-layer units is assigned to each hypercube which corresponds to class C_1,

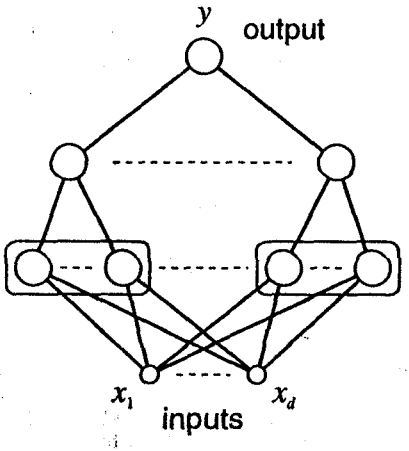

Figure 4.5. Topology of a neural network to demonstrate that networks with three layers of threshold units can generate arbitrarily complex decision boundaries. Biases have been omitted for clarity.

and there are no units corresponding to class C_2. Using the 'AND' construction for two-layer networks discussed above, we now arrange that each second-layer hidden unit generates a 1 only for inputs lying in the corresponding hypercube. This can be done by arranging for the hyperplanes associated with the first-layer units in the block to be aligned with the sides of the hypercube. Finally, the output unit has a bias which is set to -1 so that it computes a logical 'OR' of the outputs of the second-layer hidden units. In other words the output unit generates a 1 whenever one (or more) of the second-layer hidden units does so. If the output unit activation is 1, this is interpreted as class C_1, otherwise it is interpreted as class C_2. The resulting decision boundary then reflects the (arbitrary) assignment of hypercubes to classes C_1 and C_2.

The above existence proof demonstrates that feed-forward neural networks with threshold units can generate arbitrarily complex decision boundaries. The proof is of little practical interest, however, since it requires the decision boundary to be specified in advance, and also it will typically lead to very large networks. Although it is 'constructive' in that it provides a set of weights and thresholds which generate a given decision boundary, it does not answer the more practical question of how to choose an appropriate set of weights and biases for a particular problem when we are given only a set of training examples and we do not know in advance what the optimal decision boundary will be.

Returning to networks with two layers of weights, we have already seen how the AND construction for the output unit allows such a network to generate an arbitrary simply-connected convex decision region. However, by relaxing the

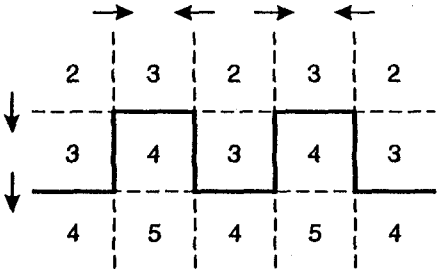

Figure 4.6. Example of a non-convex decision boundary generated by a network having two layers of threshold units. The dashed lines show the hyperplanes corresponding to the hidden units, and the arrows show the direction in which the hidden unit activations make the transition from 0 to 1. The second-layer weights are all set to 1, and so the numbers represent the value of the linear sum presented to the output unit. By setting the output unit bias to −3.5, the decision boundary represented by the solid curve is generated.

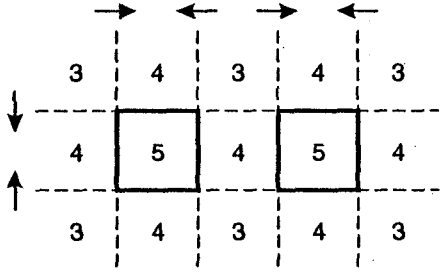

Figure 4.7. As in Figure 4.6, but showing how a disjoint decision region can be produced. In this case the bias on the output unit is set to −4.5.

restriction of an AND output unit, more general decision boundaries can be constructed (Wieland and Leighton, 1987; Huang and Lippmann, 1988). Figure 4.6 shows an example of a non-convex decision boundary, and Figure 4.7 shows a decision region which is disjoint. Huang and Lippmann (1988) give some examples of very complex decision boundaries for networks having a two layers of threshold units.

This would seem to suggest that a network with just two layers of weights could generate arbitrary decision boundaries. This is not in fact the case (Gibson and Cowan, 1990; Blum and Li, 1991) and Figure 4.8 shows an example of a decision region which cannot be produced by a network having just two layers of

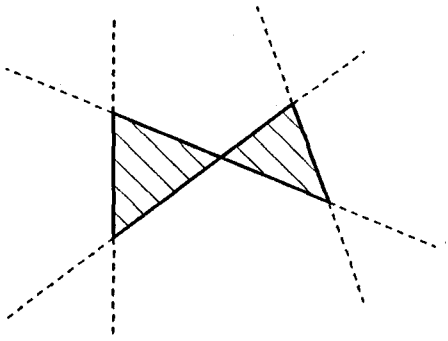

Figure 4.8. An example of a decision boundary which cannot be produced by a network having two layers of threshold units (Gibson and Cowan, 1990).

weights. Note, however, that any given decision boundary can be approximated arbitrarily closely by a two-layer network having sigmoidal activation functions, as discussed in Section 4.3.2.

So far we have discussed procedures for generating particular forms of decision boundary. A distinct, though related, issue whether a network can classify correctly a given set of data points which have been labelled as belonging to one of two classes (a dichotomy). In Chapter 3 it is shown that a network having a single layer of threshold units could classify a set of points perfectly if they were linearly separable. This would always be the case if the number of data points was at most equal to $d + 1$ where d is the dimensionality of the input space. Nilsson (1965) showed that, for a set of N data points, a two-layer network of threshold units with $N - 1$ units in the hidden layer could exactly separate an arbitrary dichotomy. Baum (1988) improved this result by showing that for N points in general position (i.e. excluding exact degeneracies) in d-dimensional space, a network with $\lceil N/d \rceil$ hidden units in a single hidden layer could separate them correctly into two classes. Here $\lceil N/d \rceil$ denotes the smallest integer which is greater than or equal to N/d.

4.3 Sigmoidal units

We turn now to a consideration of multi-layer networks having differentiable activation functions, and to the problem of representing smooth mappings between continuous variables. In Section 3.1.3 we introduced the logistic sigmoid activation function, whose outputs lie in the range $(0, 1)$, given by

$$g(a) \equiv \frac{1}{1 + \exp(-a)} \tag{4.10}$$

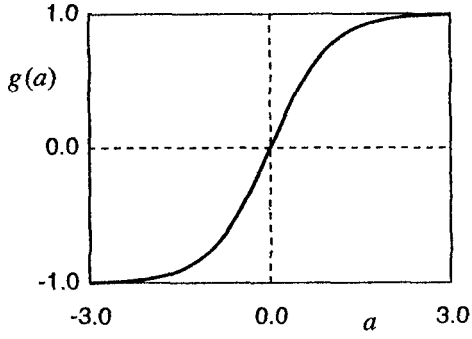

Figure 4.9. Plot of the 'tanh' activation function given by (4.11).

which is plotted in Figure 3.5. We discuss the motivation for this form of activation function in Sections 3.1.3 and 6.7.1, where we show that the use of such activation functions on the network outputs plays an important role in allowing the outputs to be given a probabilistic interpretation.

The logistic sigmoid (4.10) is often used for the hidden units of a multi-layer network. However, there may be some small practical advantage in using a 'tanh' activation function of the form

$$g(a) \equiv \tanh(a) \equiv \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4.11}$$

which is plotted in Figure 4.9. Note that (4.11) differs from the logistic function in (4.10) only through a linear transformation. Specifically, an activation function $\widetilde{g}(\widetilde{a}) = \tanh(\widetilde{a})$ is equivalent to an activation function $g(a) = 1/(1 + e^{-a})$ if we apply a linear transformation $\widetilde{a} = a/2$ to the input and a linear transformation $\widetilde{g} = 2g - 1$ to the output. Thus a neural network whose hidden units use the activation function in (4.11) is equivalent to one with hidden units using (4.10) but having different values for the weights and biases. Empirically, it is often found that 'tanh' activation functions give rise to faster convergence of training algorithms than logistic functions.

In this section we shall consider networks with linear output units. As we shall see, this does not restrict the class of functions which such networks can approximate. The use of sigmoid units at the outputs would limit the range of possible outputs to the range attainable by the sigmoid, and in some cases this would be undesirable. Even if the desired output always lay within the range of the sigmoid we note that the sigmoid function $g(\cdot)$ is monotonic, and hence is invertible, and so a desired output of y for a network with sigmoidal output units is equivalent to a desired output of $g^{-1}(y)$ for a network with linear output

units. Note, however, that there are other reasons why we might wish to use non-linear activation functions at the output units, as discussed in Chapter 6.

A sigmoidal hidden unit can approximate a linear hidden unit arbitrarily accurately. This can be achieved by arranging for all of the weights feeding into the unit, as well as the bias, to be very small, so that the summed input lies on the linear part of the sigmoid curve near the origin. The weights on the outputs of the unit leading to the next layer of units can then be made correspondingly large to re-scale the activations (with a suitable offset to the biases if necessary). Similarly, a sigmoidal hidden unit can be made to approximate a step function by setting the weights and the bias feeding into that unit to very large values.

As we shall see shortly, essentially any continuous functional mapping can be represented to arbitrary accuracy by a network having two layers of weights with sigmoidal hidden units. We therefore know that networks with extra layers of processing units also have general approximation capabilities since they contain the two-layer network as a special case. This follows from the fact that the remaining layers can be arranged to perform linear transformations as discussed above, and the identity transformation is a special case of a linear transformation (provided there is a sufficient number of hidden units so that no reduction in dimensionality occurs). Nevertheless, it is instructive to begin with a discussion of networks having three layers of weights.

4.3.1 *Three-layer networks*

In Section 4.2 we gave a heuristic proof that a three-layer network with threshold activation functions could represent an arbitrary decision boundary to arbitrary accuracy. In the same spirit we can give an analogous proof that a network with three layers of weights and sigmoidal activation functions can approximate, to arbitrary accuracy, any smooth mapping (Lapedes and Farber, 1988). The required network topology has the same form as in Figure 4.5, with each group of units in the first hidden layer again containing $2d$ units, where d is the dimensionality of the input space. As we did for threshold units, we try to arrange for each group to provide a non-zero output only when the input vector lies within a small region of the input space. For this purpose it is convenient to consider the logistic sigmoid activation function given by (4.10).

We can illustrate the construction of the network by considering a two-dimensional input space. In Figure 4.10 (a) we show the output from a single unit in the first hidden layer, given by

$$z = g(\mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0). \tag{4.12}$$

From the discussion in Section 3.1, we see that the orientation of the sigmoid is determined by the direction of $\mathbf{w}$, its location is determined by the bias w_0, and the steepness of the sigmoid slope is determined by $\|\mathbf{w}\|$. Units in the second hidden layer form linear combinations of these sigmoidal surfaces. Consider the combination of two such surfaces in which we choose the second sigmoid to have the same orientation as the first but displaced from it by a short distance. By

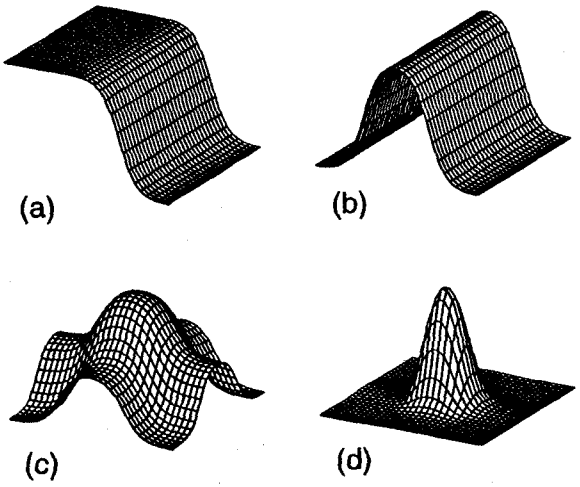

Figure 4.10. Demonstration that a network with three layers of weights, and sigmoidal hidden units, can approximate a smooth multivariate mapping to arbitrary accuracy. In (a) we see the output of a single sigmoidal unit as a function of two input variables. Adding the outputs from two such units can produce a ridge-like function (b), and adding two ridges can give a function with a maximum (c). Transforming this function with another sigmoid gives a localized response (d). By taking linear combinations of these localized functions, we can approximate any smooth functional mapping.

adding the two sigmoids together we obtain a ridge-like function as shown in Figure 4.10 (b). We next construct d of these ridges with orthogonal orientations and add them together to give a bump-like structure as shown in Figure 4.10 (c). Although this has a central peak there are also many other ridges present which stretch out to infinity. These are removed by the action of the sigmoids of the second-layer units which effectively provide a form of soft threshold to isolate the central bump, as shown in Figure 4.10 (d). We now appeal to the intuitive idea (discussed more formally in Section 5.2) that any reasonable function can be approximated to arbitrary accuracy by a linear superposition of a sufficiently large number of localized 'bump' functions, provided the coefficients in the linear combination are appropriately chosen. This superposition is performed by the output unit, which has a linear activation function.

Once again, although this is a constructive algorithm it is of little relevance to practical applications and serves mainly as an existence proof. However, the idea of representing a function as a linear superposition of localized bump functions suggests that we might consider two-layer networks in which each hidden unit generates a bump-like function directly. Such networks are called local basis

function networks, and will be considered in detail in Chapter 5.

4.3.2 *Two-layer networks*

We turn next to the question of the capabilities of networks having two layers of weights and sigmoidal hidden units. This has proven to be an important class of network for practical applications. The general topology is shown in Figure 4.1, and the network function was given explicitly in (4.7). We shall see that such networks can approximate arbitrarily well any functional (one-one or many-one) continuous mapping from one finite-dimensional space to another, provided the number M of hidden units is sufficiently large.

A considerable number of papers have appeared in the literature discussing this property including Funahashi (1989), Hecht-Nielsen (1989), Cybenko (1989), Hornik *et al.* (1989), Stinchecombe and White (1989), Cotter (1990), Ito (1991), Hornik (1991) and Kreinovich (1991). An important corollary of this result is that, in the context of a classification problem, networks with sigmoidal non-linearities and two layers of weights can approximate any decision boundary to arbitrary accuracy. Thus, such networks also provide universal non-linear discriminant functions. More generally, the capability of such networks to approximate general smooth functions allows them to model posterior probabilities of class membership.

Here we outline a simple proof of the universality property (Jones, 1990; Blum and Li, 1991). Consider the case of two input variables x_1 and x_2, and a single output variable y (the extension to larger numbers of input or output variables is straightforward). We know that, for any given value of x_1, the desired function $y(x_1, x_2)$ can be approximated to within any given (sum-of-squares) error by a Fourier decomposition in the variable x_2, giving rise to terms of the form

$$y(x_1, x_2) \simeq \sum_s A_s(x_1) \cos(s x_2) \tag{4.13}$$

where the coefficients A_s are functions of x_1. Similarly, the coefficients themselves can be expressed in terms of a Fourier series giving

$$y(x_1, x_2) \simeq \sum_s \sum_l A_{sl} \cos(l x_1) \cos(s x_2) \tag{4.14}$$

We can now use the standard trigonometric identity $\cos\alpha \cos\beta = \frac{1}{2}\cos(\alpha + \beta) + \frac{1}{2}\cos(\alpha - \beta)$ to write this as a linear combination of terms of the form $\cos(z_{sl})$ and $\cos(z'_{sl})$ where $z_{sl} = l x_1 + s x_2$ and $z'_{sl} = l x_1 - s x_2$. Finally, we note that the function $\cos(z)$ can be approximated to arbitrary accuracy by a linear combination of threshold step functions. This can be seen by making an explicit construction, illustrated in Figure 4.11, for a function $f(z)$ in terms of a piecewise constant function, of the form

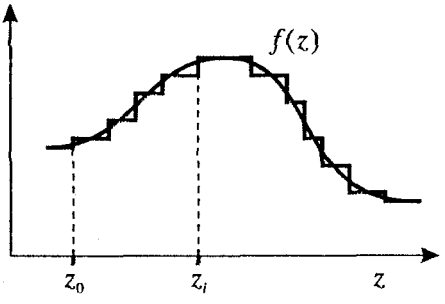

Figure 4.11. Approximation of a continuous function $f(z)$ by a linear superposition of threshold step functions. This forms the basis of a simple proof that a two-layer network having sigmoidal hidden units and linear output units can approximate a continuous function to arbitrary accuracy.

$$f(z) \simeq f_0 + \sum_{i=0}^{N} \{f_{i+1} - f_i\} H(z - z_i) \tag{4.15}$$

where $H(z)$ is the Heaviside step function. Thus we see that the function $y(x_1, x_2)$ can be expressed as a linear combination of step functions whose arguments are linear combinations of x_1 and x_2. In other words the function $y(x_1, x_2)$ can be approximated by a two-layer network with threshold hidden units and linear output units. Finally, we recall that threshold activation functions can be approximated arbitrarily well by sigmoidal functions, simply by scaling the weights and biases.

Note that this proof does not indicate whether the network can simultaneously approximate the derivatives of the function, since our approximation in (4.15) has zero derivative except at discrete points at which the derivative is undefined. A proof that two-layer networks having sigmoidal hidden units can simultaneously approximate both a function and its derivatives was given by Hornik *et al.* (1990).

As a simple illustration of the capabilities of two-layer networks with sigmoidal hidden units we consider mappings from a single input x to a single output y. In Figure 4.12 we show the result of training a network with five hidden units having 'tanh' activation functions given by (4.11). The data sets each consist of 50 data points generated by a variety of functions, and the network has a single linear output unit and was trained for 1000 epochs using the BFGS quasi-Newton algorithm described in Section 7.10. We see that the same network can generate a wide variety of different functions simply by choosing different values for the weights and biases.

The above proofs were concerned with demonstrating that a network with a

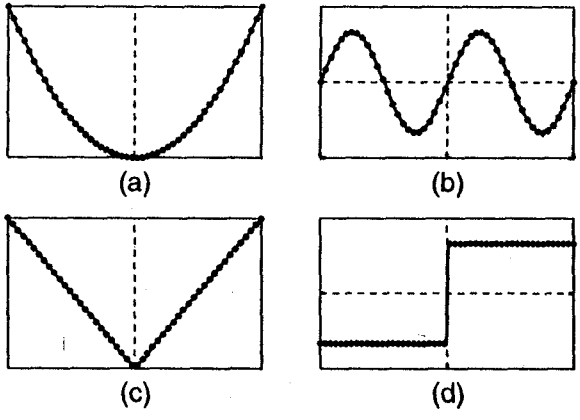

Figure 4.12. Examples of sets of data points (circles) together with the corresponding functions represented by a multi-layer perceptron network which has been trained using the data. The data sets were generated by sampling the following functions: (a) x^2, (b) $\sin(2\pi x)$ (c) $|x|$ which is continuous but with a discontinuous first derivative, and (d) the step function $\theta(x) \equiv \text{sign}(x)$, which is discontinuous.

sufficiently large number of hidden units could approximate a particular mapping. White (1990) and Gallant and White (1992) considered the conditions under which a network will actually learn a given mapping from a finite data set, showing how the number of hidden units must grow as the size of the data set grows.

If we try to approximate a given function $h(\mathbf{x})$ with a network having a finite number M of hidden units, then there will be a residual error. Jones (1992) and Barron (1993) have shown that this error decreases as $\mathcal{O}(1/M)$ as the number M of hidden units is increased.

Since we know that, with a single hidden layer, we can approximate any mapping to arbitrary accuracy we might wonder if there is anything to be gained by using any other network topology, for instance one having several hidden layers. One possibility is that by using extra layers we might find more efficient approximations in the sense of achieving the same level of accuracy with fewer weights and biases in total. Very little is currently known about this issue. However, later chapters discuss situations in which there are other good reasons to consider networks with more complex topologies, including networks with several hidden layers, and networks with only partial connectivity between layers.

4.4 Weight-space symmetries

Consider a two-layer network having M hidden units, with 'tanh' activation functions given by (4.11), and full connectivity in both layers. If we change the sign of all of the weights and the bias feeding into a particular hidden unit, then, for a given input pattern, the sign of the activation of the hidden unit will be reversed, since (4.11) is an odd function. This can be compensated by changing the sign of all of the weights leading out of that hidden unit. Thus, by changing the signs of a particular group of weights (and a bias), the input–output mapping function represented by the network is unchanged, and so we have found two different weight vectors which give rise to the same mapping function. For M hidden units, there will be M such 'sign-flip' symmetries, and thus any given weight vector will be one of a set 2^M equivalent weight vectors (Chen *et al.*, 1993).

Similarly, imagine that we interchange the values of all of the weights (and the bias) leading into and out of a particular hidden unit with the corresponding values of the weights (and bias) associated with a different hidden unit. Again, this clearly leaves the network input–output mapping function unchanged, but it corresponds to a different choice of weight vector. For M hidden units, any given weight vector will have $M!$ equivalent weight vectors associated with this interchange symmetry, corresponding to the $M!$ different orderings of the hidden units (Chen *et al.*, 1993). The network will therefore have an overall weight-space symmetry factor of $M!2^M$. For networks with more than two layers of weights, the total level of symmetry will be given by the product of such factors, one for each layer of hidden units.

It turns out that these factors account for all of the symmetries in weight space (except for possible accidental symmetries due to specific choices for the weight values). Furthermore, the existence of these symmetries is not a particular property of the 'tanh' function, but applies to a wide range of activation functions (Sussmann, 1992; Chen *et al.*, 1993; Albertini and Sontag, 1993; Kůrková and Kainen, 1994). In many cases, these symmetries in weight space are of little practical consequence. However, we shall encounter an example in Section 10.6 where we need to take them into account.

4.5 Higher-order networks

So far in this chapter we have considered units for which the output is given by a non-linear activation function acting on a *linear* combination of the inputs of the form

$$a_j = \sum_i w_{ji} x_i + w_{j0}. \tag{4.16}$$

We have seen that networks composed of such units can in principle approximate any functional mapping to arbitrary accuracy, and therefore constitute a universal class of parametrized multivariate non-linear mappings. Nevertheless, there is still considerable interest in studying other forms of processing unit. Chapter 5

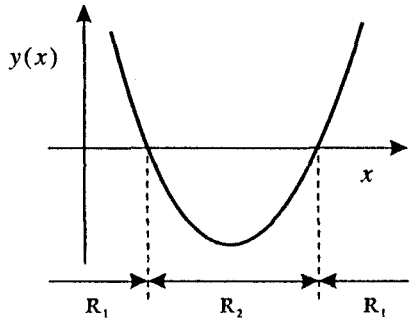

Figure 4.13. A one-dimensional input space x with decision regions $\mathcal{R}_1$ (which is disjoint) and $\mathcal{R}_2$. A linear discriminant function cannot generate the required decision boundaries, but a quadratic discriminant $y(x)$, shown by the solid curve, can. The required decision rule then assigns an input x to class $\mathcal{C}_1$ if $y(x) > 0$ and to class $\mathcal{C}_2$ otherwise.

for instance is devoted to a study of networks containing units whose activations depend on the *distance* of an input vector from the weight vector. Here we consider some extensions of the linear expression in (4.16) which therefore contain (4.16) as a special case.

As discussed in Chapter 3, a network consisting of a single layer of units of the form (4.16) can only produce decision boundaries which take the form of piecewise hyperplanes in the input space. Such a network is therefore incapable of generating decision regions which are concave or which are multiply connected. Consider the one-dimensional input space x illustrated in Figure 4.13. We wish to find a discriminant function which will divide the space into the decision regions $\mathcal{R}_1$ and $\mathcal{R}_2$ as shown. A linear discriminant function is not sufficient since the region $\mathcal{R}_1$ is disjoint. However, the required decision boundaries can be generated by a quadratic discriminant of the form

$$y(x) = w_2 x^2 + w_1 x + w_0 \tag{4.17}$$

provided the weights w_2, w_1 and w_0 are chosen appropriately.

We can generalize this idea to higher orders than just quadratic, and to several input variables (Ivakhnenko, 1971; Barron and Barron, 1988). This leads to *higher-order* processing units (Giles and Maxwell, 1987; Ghosh and Shin, 1992), also known as *sigma-pi* units (Rumelhart *et al.*, 1986). For second-order units the generalization of (4.16) takes the form

$$a_j = w_j^{(0)} + \sum_{i_1=1}^{d} w_{ji_1}^{(1)} x_{i_1} + \sum_{i_1=1}^{d} \sum_{i_2=1}^{d} w_{ji_1i_2}^{(2)} x_{i_1} x_{i_2} \tag{4.18}$$

where the sums run over all inputs, or units, which send connections to unit j. As before, this sum is then transformed using a non-linear activation function to give $z_j = g(a_j)$. If terms up to degree M are retained, this will be known as an Mth-order unit. Clearly (4.18) includes the conventional linear (first-order) unit (4.16) as a special case. The similarity to the higher-order polynomials discussed in Section 1.7 is clear. Note that the summations in (4.18) can be constrained to allow for the permutation symmetry of the higher-order terms. For instance, the term $x_{i_1} x_{i_2}$ is equivalent to the term $x_{i_2} x_{i_1}$ and so we need only retain one of these in the summation. The total number of independent parameters in a higher-order expression such as (4.18) is discussed in Exercises 1.6–1.8.

If we introduce an extra input $z_0 = +1$ then, for an Mth-order unit we can absorb all of the terms up to the Mth-order within the Mth-order term. For instance, if we consider second-order units we can write (4.18) in the equivalent form

$$a_j = \sum_{i_1=0}^{d} \sum_{i_2=0}^{d} w_{ji_1i_2}^{(2)} z_{i_1} z_{i_2} \tag{4.19}$$

with similar generalizations to higher orders.

We see that there will typically be many more weight parameters in a higher-order unit than there are in a first-order unit. For example, if we consider an input dimensionality of $d = 10$ then a first-order unit will have 11 weight parameters (including the bias), a second-order unit will have 66 independent weights, and a third-order unit will have 572 independent weights. This explosion in the number of parameters is the principal difficulty with such higher-order units. The compensating benefit is that it is possible to arrange for the response of the unit to be invariant to various transformations of the input. In Section 8.7.4 it is shown how a third-order unit can be simultaneously invariant to translations, rotations and scalings of the input patterns when these are drawn from pixels in a two-dimensional image. This is achieved by imposing constraints on the weights, which also greatly reduce the number of independent parameters, and thereby makes the use of such units a tractable proposition. Higher-order units are generally used only in the first layer of a network, with subsequent layers being composed of conventional first-order units.

4.6 Projection pursuit regression and other conventional techniques

Statisticians have developed a variety of techniques for classification and regression which can be regarded as complementary to the multi-layer perceptron. Here we give a brief overview of the most prominent of these approaches, and indicate their relation to neural networks. One of the most closely related is that of

projection pursuit regression (Friedman and Stuetzle, 1981; Huber, 1985). For a single output variable, the projection pursuit regression mapping can be written in the form

$$y = \sum_{j=1}^{M} w_j \phi_j(\mathbf{u}_j^{\mathrm{T}}\mathbf{x} + u_{j0}) + w_0 \tag{4.20}$$

which is remarkably similar to a two-layer feed-forward neural network. The parameters $\mathbf{u}_j$ and u_{j0} define the projection of the input vector $\mathbf{x}$ onto a set of planes labelled by $j = 1, \ldots, M$, as in the multi-layer perceptron. These projections are transformed by non-linear 'activation functions' ϕ_j and these in turn are linearly combined to form the output variable y. Determination of the parameters in the model is done by minimizing a sum-of-squares error function. One important difference is that each 'hidden unit' in projection pursuit regression is allowed a different activation function, and these functions are not prescribed in advance, but are determined from the data as part of the training procedure.

Another difference is that typically all of the parameters in a neural network are optimized simultaneously, while those in projection pursuit regression are optimized cyclically in groups. Specifically, training in the projection pursuit regression network takes place for one hidden unit at a time, and for each hidden unit the second-layer weights are optimized first, followed by the activation function, followed by the first-layer weights. The process is repeated for each hidden unit in turn, until a sufficiently small value for the error function is achieved, or until some other stopping criterion is satisfied. Since the output y in (4.20) depends linearly on the second-layer parameters, these can be optimized by linear least-squares techniques, as discussed in Section 3.4. Optimization of the activation functions ϕ_j represents a problem in one-dimensional curve-fitting for which a variety of techniques can be used, such as cubic splines (Press *et al.*, 1992). Finally, the optimization of the first-layer weights requires non-linear techniques of the kind discussed in Chapter 7.

Several generalizations to more than one output variable are possible (Ripley, 1994) depending on whether the outputs share common basis functions ϕ_j, and if not, whether the separate basis functions ϕ_{jk} (where k labels the outputs) share common projection directions. In terms of representational capability, we can regard projection pursuit regression as a generalization of the multi-layer perceptron, in that the activation functions are more flexible. It is therefore not surprising that projection pursuit regression should have the same 'universal' approximation capabilities as multi-layer perceptrons (Diaconis and Shahshahani, 1984; Jones, 1987). Projection pursuit regression is compared with multi-layer perceptron networks in Hwang *et al.* (1994).

Another framework for non-linear regression is the class of *generalized additive models* (Hastie and Tibshirani, 1990) which take the form

$$y = g\left(\sum_{i=1}^{d} \phi_i(x_i) + w_0\right) \tag{4.21}$$

where the $\phi_i(\cdot)$ are non-linear functions and $g(\cdot)$ represents the logistic sigmoid function (4.10). This is actually a very restrictive class of models, since it does not allow for interactions between the input variables. Thus a function of the form $x_1 x_2$, for example, cannot be modelled. They do, however, have an advantage in terms of the interpretation of the trained model, since the individual univariate functions $\phi_i(\cdot)$ can be plotted.

An extension of the additive models which allows for interactions is given by the technique of *multivariate adaptive regression splines* (MARS) (Friedman, 1991) for which the mapping function can be written

$$y = \sum_{j=1}^{M} w_j \prod_{k=1}^{K_j} \phi_{jk}(x_{\nu(k,j)}) + w_0 \tag{4.22}$$

where the jth basis function is given by a product of some number K_j of one-dimensional spline functions ϕ_{jk} (Press *et al.*, 1992) each of which depends on one of the input variables x_ν, where the particular input variable used in each case is governed by a label $\nu(k, j)$. The basis functions are adaptive in that the number of factors K_j, the labels $\nu(k, j)$, and the knots for the one-dimensional spline functions are all determined from the data. Basis functions are added incrementally during learning, using the technique of sequential forward selection discussed in Section 8.5.3.

An alternative framework for learning non-linear multivariate mappings involves partitioning the input space into regions, and fitting a different mapping within each region. In many such algorithms, the partitions are formed from hyperplanes which are parallel to the input variable axes, as indicated in Figure 4.14. In the simplest case the output variable is taken to be constant within each region. A common technique is to form a binary partition in which the input space is divided into two regions, and then each of these is divided in turn, and so on. This form of partitioning can then be described by a binary tree structure, in which each leaf represents one of the regions. Successive branches can be added to the tree during learning, with the locations of the hyperplanes being determined by the data. Procedures are often also devised for pruning the tree structure as a way of controlling the effective complexity of the model. Two of the best known algorithms of this kind are *classification and regression trees* (CART) (Breiman *et al.*, 1984) and ID3 (Quinlan, 1986). A detailed discussion of these algorithms would, however, take us too far afield.

4.7 Kolmogorov's theorem

There is a theorem due to Kolmogorov (1957) which, although of no direct practical significance, does have an interesting relation to neural networks. The theo-

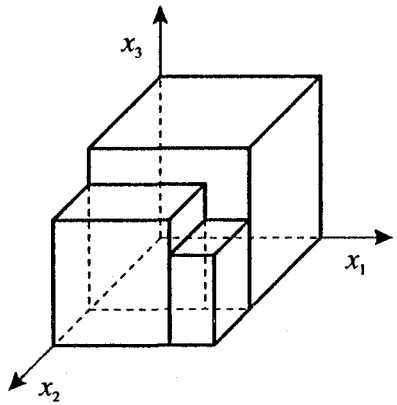

Figure 4.14. An example of the partitioning of a space by hyperplanes which are parallel to the coordinate axes. Such partitions form the basis of a number of algorithms for solving classification and regression problems.

rem has its origins at the end of the nineteenth century when the mathematician Hilbert compiled a list of 23 unsolved problems as a challenge for twentieth century mathematicians (Hilbert, 1900). Hilbert's thirteenth problem concerns the issue of whether functions of several variables can be represented in terms of superpositions of functions of fewer variables. He conjectured that there exist continuous functions of three variables which cannot be represented as superpositions of functions of two variables. The conjecture was disproved by Arnold (1957). However, a much more general result was obtained by Kolmogorov (1957) who showed that every continuous function of several variables (for a closed and bounded input domain) can be represented as the superposition of a small number of functions of one variable. Improved versions of Kolmogorov's theorem have been given by Sprecher (1965), Kahane (1975) and Lorentz (1976). In neural network terms this theorem says that any continuous mapping $y(\mathbf{x})$ from d input variables x_i to an output variable y can be represented exactly by a three-layer neural network having $d(2d+1)$ units in the first hidden layer and $(2d+1)$ units in the second hidden layer. The network topology is illustrated, for the case of a single output, in Figure 4.15. Each unit in the first hidden layer computes a function of one of the input variables x_i given by $h_j(x_i)$ where $j = 1, \ldots, 2d+1$ and the h_j are strictly monotonic functions. The activation of the jth unit in the second hidden layer is given by

$$z_j = \sum_{i=1}^{d} \lambda_i h_j(x_i) \tag{4.23}$$

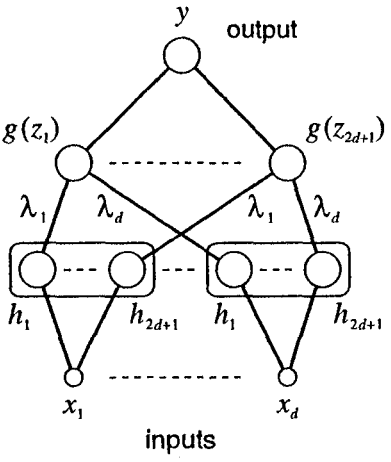

Figure 4.15. Network topology to implement Kolmogorov's theorem.

where $0 < \lambda_i < 1$ are constants. The output y of the network is then given by

$$y = \sum_{j=1}^{2d+1} g(z_j) \tag{4.24}$$

where the function g is real and continuous. Note that the function g depends on the particular function $y(\mathbf{x})$ which is to be represented, while the functions h_j do not. This expression can be extended to a network with more that one output unit simply by modifying (4.24) to give

$$y_k = \sum_{j=1}^{2d+1} g_k(z_j). \tag{4.25}$$

Note that the theorem only guarantees the existence of a suitable network. No actual examples of functions h_j or g are known, and there is no known constructive technique for finding them.

While Kolmogorov's theorem is remarkable, its relevance to practical neural computing is at best limited (Girosi and Poggio, 1989; Kůrková, 1991; Kůrková, 1992). There are two reasons for this. First, the functions h_j are far from being smooth. Indeed, it has been shown that if the functions h_j are required to be smooth then the theorem breaks down (Vitushkin, 1954). The presence of non-smooth functions in a network would lead to problems of extreme sensitivity

to the input variables. Smoothness of the network mapping is an important property in connection with the generalization performance of a network, as is discussed in greater detail in Section 9.2. The second reason is that the function g depends on the particular function $y(\mathbf{x})$ which we wish to represent. This is the converse of the situation which we generally encounter with neural networks. Usually, we consider fixed activation functions, and then adjust the number of hidden units, and the values of the weights and biases, to give a sufficiently close representation of the desired mapping. In Kolmogorov's theorem the number of hidden units is fixed, while the activation functions depend on the mapping. In general, if we are trying to represent an arbitrary continuous function then we cannot hope to do this exactly with a finite number of fixed activation functions since the finite number of adjustable parameters represents a finite number of degrees of freedom, and a general continuous function has effectively infinitely many degrees of freedom.

4.8 Error back-propagation

So far in this chapter we have concentrated on the representational capabilities of multi-layer networks. We next consider how such a network can learn a suitable mapping from a given data set. As in previous chapters, learning will be based on the definition of a suitable error function, which is then minimized with respect to the weights and biases in the network.

Consider first the case of networks of threshold units. The final layer of weights in the network can be regarded as a perceptron with inputs given by the outputs of the last layer of hidden units. These weights could therefore be chosen using the perceptron learning rule introduced in Chapter 3. Such an approach cannot, however, be used to determine the weights in earlier layers of the network. Although such layers could in principle be regarded as being like single-layer perceptrons, we have no procedure for assigning target values to their outputs, and so the perceptron procedure cannot be applied. This is known as the *credit assignment problem*. If an output unit produces an incorrect response when the network is presented with an input vector we have no way of determining which of the hidden units should be regarded as responsible for generating the error, so there is no way of determining which weights to adjust or by how much.

The solution to this credit assignment problem is relatively simple. If we consider a network with differentiable activation functions, then the activations of the output units become differentiable functions of both the input variables, and of the weights and biases. If we define an error function, such as the sum-of-squares error introduced in Chapter 1, which is a differentiable function of the network outputs, then this error is itself a differentiable function of the weights. We can therefore evaluate the derivatives of the error with respect to the weights, and these derivatives can then be used to find weight values which minimize the error function, by using either gradient descent or one of the more powerful optimization methods discussed in Chapter 7. The algorithm for evaluating the derivatives of the error function is known as *back-propagation* since, as we shall

see, it corresponds to a propagation of errors backwards through the network. The technique of back-propagation was popularized in a paper by Rumelhart, Hinton and Williams (1986). However, similar ideas had been developed earlier by a number of researchers including Werbos (1974) and Parker (1985).

It should be noted that the term back-propagation is used in the neural computing literature to mean a variety of different things. For instance, the multi-layer perceptron architecture is sometimes called a back-propagation network. The term back-propagation is also used to describe the training of a multi-layer perceptron using gradient descent applied to a sum-of-squares error function. In order to clarify the terminology it is useful to consider the nature of the training process more carefully. Most training algorithms involve an iterative procedure for minimization of an error function, with adjustments to the weights being made in a sequence of steps. At each such step we can distinguish between two distinct stages. In the first stage, the derivatives of the error function with respect to the weights must be evaluated. As we shall see, the important contribution of the back-propagation technique is in providing a computationally efficient method for evaluating such derivatives. Since it is at this stage that errors are propagated backwards through the network, we shall use the term back-propagation specifically to describe the evaluation of derivatives. In the second stage, the derivatives are then used to compute the adjustments to be made to the weights. The simplest such technique, and the one originally considered by Rumelhart *et al.* (1986), involves gradient descent. It is important to recognize that the two stages are distinct. Thus, the first stage process, namely the propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multi-layer perceptron. It can also be applied to error functions other that just the simple sum-of-squares, and to the evaluation of other derivatives such as the Jacobian and Hessian matrices, as we shall see later in this chapter. Similarly, the second stage of weight adjustment using the calculated derivatives can be tackled using a variety of optimization schemes (discussed at length in Chapter 7), many of which are substantially more powerful than simple gradient descent.

4.8.1 *Evaluation of error function derivatives*

We now derive the back-propagation algorithm for a general network having arbitrary feed-forward topology, and arbitrary differentiable non-linear activation functions, for the case of an arbitrary differentiable error function. The resulting formulae will then be illustrated using a simple layered network structure having a single layer of sigmoidal hidden units and a sum-of-squares error.

In a general feed-forward network, each unit computes a weighted sum of its inputs of the form

$$a_j = \sum_i w_{ji} z_i \tag{4.26}$$

where z_i is the activation of a unit, or input, which sends a connection to unit

j, and w_{ji} is the weight associated with that connection. The summation runs over all units which send connections to unit j. In Section 4.1 we showed that biases can be included in this sum by introducing an extra unit, or input, with activation fixed at $+1$. We therefore do not need to deal with biases explicitly. The sum in (4.26) is transformed by a non-linear activation function $g(\cdot)$ to give the activation z_j of unit j in the form

$$z_j = g(a_j). \tag{4.27}$$

Note that one or more of the variables z_i in the sum in (4.26) could be an input, in which case we shall denote it by x_i. Similarly, the unit j in (4.27) could be an output unit, in which case we denote its activation by y_k.

As before, we shall seek to determine suitable values for the weights in the network by minimization of an appropriate error function. Here we shall consider error functions which can be written as a sum, over all patterns in the training set, of an error defined for each pattern separately

$$E = \sum_n E^n \tag{4.28}$$

where n labels the patterns. Nearly all error functions of practical interest take this form, for reasons which are explained in Chapter 6. We shall also suppose that the error E^n can be expressed as a differentiable function of the network output variables so that

$$E^n = E^n(y_1, \ldots, y_c). \tag{4.29}$$

Our goal is to find a procedure for evaluating the derivatives of the error function E with respect to the weights and biases in the network. Using (4.28) we can express these derivatives as sums over the training set patterns of the derivatives for each pattern separately. From now on we shall therefore consider one pattern at a time.

For each pattern we shall suppose that we have supplied the corresponding input vector to the network and calculated the activations of all of the hidden and output units in the network by successive application of (4.26) and (4.27). This process is often called *forward propagation* since it can be regarded as a forward flow of information through the network.

Now consider the evaluation of the derivative of E^n with respect to some weight w_{ji}. The outputs of the various units will depend on the particular input pattern n. However, in order to keep the notation uncluttered, we shall omit the superscript n from the input and activation variables. First we note that E^n depends on the weight w_{ji} only via the summed input a_j to unit j. We can therefore apply the chain rule for partial derivatives to give

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{\partial E^n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}. \tag{4.30}$$

We now introduce a useful notation

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} \tag{4.31}$$

where the δ's are often referred to as *errors* for reasons we shall see shortly. Using (4.26) we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i. \tag{4.32}$$

Substituting (4.31) and (4.32) into (4.30) we then obtain

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j z_i. \tag{4.33}$$

Note that this has the same general form as obtained for single-layer networks in Section 3.4. Equation (4.33) tells us that the required derivative is obtained simply by multiplying the value of δ for the unit at the output end of the weight by the value of z for the unit at the input end of the weight (where $z = 1$ in the case of a bias). Thus, in order to evaluate the derivatives, we need only to calculate the value of δ_j for each hidden and output unit in the network, and then apply (4.33).

For the output units the evaluation of δ_k is straightforward. From the definition (4.31) we have

$$\delta_k \equiv \frac{\partial E^n}{\partial a_k} = g'(a_k)\frac{\partial E^n}{\partial y_k} \tag{4.34}$$

where we have used (4.27) with z_k denoted by y_k. In order to evaluate (4.34) we substitute appropriate expressions for $g'(a)$ and $\partial E^n/\partial y$. This will be illustrated with a simple example shortly.

To evaluate the δ's for hidden units we again make use of the chain rule for partial derivatives,

$$\delta_j \equiv \frac{\partial E^n}{\partial a_j} = \sum_k \frac{\partial E^n}{\partial a_k}\frac{\partial a_k}{\partial a_j} \tag{4.35}$$

where the sum runs over all units k to which unit j sends connections. The arrangement of units and weights is illustrated in Figure 4.16. Note that the units labelled k could include other hidden units and/or output units. In writing

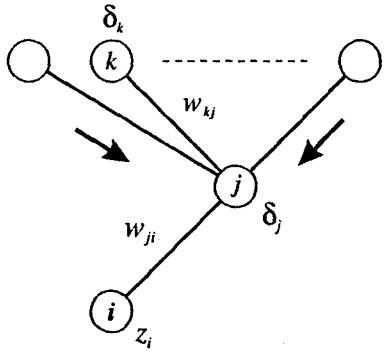

Figure 4.16. Illustration of the calculation of δ_j for hidden unit j by back-propagation of the δ's from those units k to which unit j sends connections.

down (4.35) we are making use of the fact that variations in a_j give rise to variations in the error function only through variations in the variables a_k. If we now substitute the definition of δ given by (4.31) into (4.35), and make use of (4.26) and (4.27), we obtain the following *back-propagation* formula

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k \tag{4.36}$$

which tells us that the value of δ for a particular hidden unit can be obtained by propagating the δ's backwards from units higher up in the network, as illustrated in Figure 4.16. Since we already know the values of the δ's for the output units, it follows that by recursively applying (4.36) we can evaluate the δ's for all of the hidden units in a feed-forward network, regardless of its topology.

We can summarize the back-propagation procedure for evaluating the derivatives of the error E^n with respect to the weights in four steps:

1. Apply an input vector $\mathbf{x}^n$ to the network and forward propagate through the network using (4.26) and (4.27) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (4.34).
3. Back-propagate the δ's using (4.36) to obtain δ_j for each hidden unit in the network.
4. Use (4.33) to evaluate the required derivatives.

The derivative of the total error E can then be obtained by repeating the above steps for each pattern in the training set, and then summing over all patterns:

$$\frac{\partial E}{\partial w_{ji}} = \sum_n \frac{\partial E^n}{\partial w_{ji}}. \tag{4.37}$$

In the above derivation we have implicitly assumed that each hidden or output unit in the network has the same activation function $g(\cdot)$. The derivation is easily generalized, however, to allow different units to have individual activation functions, simply by keeping track of which form of $g(\cdot)$ goes with which unit.

4.8.2 *A simple example*

The above derivation of the back-propagation procedure allowed for general forms for the error function, the activation functions and the network topology. In order to illustrate the application of this algorithm, we shall consider a particular example. This is chosen both for its simplicity and for its practical importance, since many applications of neural networks reported in the literature make use of this type of network. Specifically, we shall consider a two-layer network of the form illustrated in Figure 4.1, together with a sum-of-squares error. The output units have linear activation functions while the hidden units have logistic sigmoid activation functions given by (4.10), and repeated here:

$$g(a) \equiv \frac{1}{1 + \exp(-a)}. \tag{4.38}$$

A useful feature of this function is that its derivative can be expressed in a particularly simple form:

$$g'(a) = g(a)(1 - g(a)). \tag{4.39}$$

In a software implementation of the network algorithm, (4.39) represents a convenient property since the derivative of the activation can be obtained efficiently from the activation itself using two arithmetic operations.

For the standard sum-of-squares error function, the error for pattern n is given by

$$E^n = \frac{1}{2} \sum_{k=1}^{c} (y_k - t_k)^2 \tag{4.40}$$

where y_k is the response of output unit k, and t_k is the corresponding target, for a particular input pattern $\mathbf{x}^n$.

Using the expressions derived above for back-propagation in a general network, together with (4.39) and (4.40), we obtain the following results. For the output units, the δ's are given by

$$\delta_k = y_k - t_k \tag{4.41}$$

while for units in the hidden layer the δ's are found using

$$\delta_j = z_j(1 - z_j)\sum_{k=1}^{c} w_{kj}\delta_k \tag{4.42}$$

where the sum runs over all output units. The derivatives with respect to the first-layer and second-layer weights are then given by

$$\frac{\partial E^n}{\partial w_{ji}} = \delta_j x_i, \qquad \frac{\partial E^n}{\partial w_{kj}} = \delta_k z_j. \tag{4.43}$$

So far we have discussed the evaluation of the derivatives of the error function with respect to the weights and biases in the network. In order to turn this into a learning algorithm we need some method for updating the weights based on these derivatives. In Chapter 7 we discuss several such parameter optimization strategies in some detail. For the moment, we consider the fixed-step gradient descent technique introduced in Section 3.4. We have the choice of updating the weights either after presentation of each pattern (on-line learning) or after first summing the derivatives over all the patterns in the training set (batch learning). In the former case the weights in the first layer are updated using

$$\Delta w_{ji} = -\eta\delta_j x_i \tag{4.44}$$

while in the case of batch learning the first-layer weights are updated using

$$\Delta w_{ji} = -\eta\sum_n \delta_j^n x_i^n \tag{4.45}$$

with analogous expressions for the second-layer weights.

4.8.3 *Efficiency of back-propagation*

One of the most important aspects of back-propagation is its computational efficiency. To understand this, let us examine how the number of computer operations required to evaluate the derivatives of the error function scales with the size of the network. Let W be the total number of weights and biases. Then a single evaluation of the error function (for a given input pattern) would require $\mathcal{O}(W)$ operations, for sufficiently large W. This follows from the fact that, except for a network with very sparse connections, the number of weights is typically much greater than the number of units. Thus, the bulk of the computational effort in forward propagation is concerned with evaluating the sums in (4.26), with the evaluation of the activation functions representing a small overhead. Each term in the sum in (4.26) requires one multiplication and one addition, leading to an overall computational cost which is $\mathcal{O}(W)$.

For W weights in total there are W such derivatives to evaluate. If we simply

took the expression for the error function and wrote down explicit formulae for the derivatives and then evaluated them numerically by forward propagation, we would have to evaluate W such terms (one for each weight or bias) each requiring $\mathcal{O}(W)$ operations. Thus, the total computational effort required to evaluate all the derivatives would scale as $\mathcal{O}(W^2)$. By comparison, back-propagation allows the derivatives to be evaluated in $\mathcal{O}(W)$ operations. This follows from the fact that both the forward and the backward propagation phases are $\mathcal{O}(W)$, and the evaluation of the derivative using (4.33) also requires $\mathcal{O}(W)$ operations. Thus back-propagation has reduced the computational complexity from $\mathcal{O}(W^2)$ to $\mathcal{O}(W)$ for each input vector. Since the training of MLP networks, even using back-propagation, can be very time consuming, this gain in efficiency is crucial. For a total of N training patterns, the number of computational steps required to evaluate the complete error function for the whole data set is N times larger than for one pattern.

The practical importance of the $\mathcal{O}(W)$ scaling of back-propagation is analogous in some respects to that of the fast Fourier transform (FFT) algorithm (Brigham, 1974; Press *et al.*, 1992) which reduces the computational complexity of evaluating an L-point Fourier transform from $\mathcal{O}(L^2)$ to $\mathcal{O}(L \log_2 L)$. The discovery of this algorithm led to the widespread use of Fourier transforms in a large range of practical applications.

4.8.4 *Numerical differentiation*

An alternative approach to back-propagation for computing the derivatives of the error function is to use finite differences. This can be done by perturbing each weight in turn, and approximating the derivatives by the expression

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{E^n(w_{ji} + \epsilon) - E^n(w_{ji})}{\epsilon} + \mathcal{O}(\epsilon) \tag{4.46}$$

where $\epsilon \ll 1$ is a small quantity. In a software simulation, the accuracy of the approximation to the derivatives can be improved by making ϵ smaller, until numerical roundoff problems arise. The main problem with this approach is that the highly desirable $\mathcal{O}(W)$ scaling has been lost. Each forward propagation requires $\mathcal{O}(W)$ steps, and there are W weights in the network each of which must be perturbed individually, so that the overall scaling is $\mathcal{O}(W^2)$. However, finite differences play an important role in practice, since a numerical comparison of the derivatives calculated by back-propagation with those obtained using finite differences provides a very powerful check on the correctness of any software implementation of the back-propagation algorithm.

The accuracy of the finite differences method can be improved significantly by using symmetrical *central differences* of the form

$$\frac{\partial E^n}{\partial w_{ji}} = \frac{E^n(w_{ji} + \epsilon) - E^n(w_{ji} - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{4.47}$$

In this case the $\mathcal{O}(\epsilon)$ corrections cancel, as is easily verified by Taylor expansion on the right-hand side of (4.47), and so the residual corrections are $\mathcal{O}(\epsilon^2)$. The number of computational steps is, however, roughly doubled compared with (4.46).

We have seen that the derivatives of an error function with respect to the weights in a network can be expressed efficiently through the relation

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial a_j} z_i. \tag{4.48}$$

Instead of using the technique of central differences to evaluate the derivatives $\partial E^n/\partial w_{ji}$ directly, we can use it to estimate $\partial E^n/\partial a_j$ since

$$\frac{\partial E^n}{\partial a_j} = \frac{E^n(a_j + \epsilon) - E^n(a_j - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2) \tag{4.49}$$

We can then make use of (4.48) to evaluate the required derivatives. Because the derivatives with respect to the weights are found from (4.48) this approach is still relatively efficient. Back-propagation requires one forward and one backward propagation through the network, each taking $\mathcal{O}(W)$ steps, in order to evaluate all of the $\partial E/\partial a_i$. By comparison, (4.49) requires $2M$ forward propagations, where M is the number of hidden and output nodes. The overall scaling is therefore proportional to MW, which is typically much less than the $\mathcal{O}(W^2)$ scaling of (4.47), but more than the $\mathcal{O}(W)$ scaling of back-propagation. This technique is called *node perturbation* (Jabri and Flower, 1991), and is closely related to the *madeline III* learning rule (Widrow and Lehr, 1990).

In a software implementation, derivatives should be evaluated using back-propagation, since this gives the greatest accuracy and numerical efficiency. However, the results should be compared with numerical differentiation using (4.47) for a few test cases in order to check the correctness of the implementation.

4.9 The Jacobian matrix

We have seen how the derivatives of an error function with respect to the weights can be obtained by the propagation of errors backwards through the network. The technique of back-propagation can also be applied to the calculation of other derivatives. Here we consider the evaluation of the *Jacobian* matrix, whose elements are given by the derivatives of the network outputs with respect to the inputs

$$J_{ki} \equiv \frac{\partial y_k}{\partial x_i} \tag{4.50}$$

where each such derivative is evaluated with all other inputs held fixed. Note that the term Jacobian matrix is also sometimes used to describe the derivatives

of the error function with respect to the network weights, as calculated earlier using back-propagation. The Jacobian matrix provides a measure of the local sensitivity of the outputs to changes in each of the input variables, and is useful in several contexts in the application of neural networks. For instance, if there are known errors associated with the input variables, then the Jacobian matrix allows these to be propagated through the trained network in order to estimate their contribution to the errors at the outputs. Thus, we have

$$\Delta y_k \simeq \sum_i \frac{\partial y_k}{\partial x_i} \Delta x_i. \tag{4.51}$$

In general, the network mapping represented by a trained neural network will be non-linear, and so the elements of the Jacobian matrix will not be constants but will depend on the particular input vector used. Thus (4.51) is valid only for small perturbations of the inputs, and the Jacobian itself must be re-evaluated for each new input vector.

The Jacobian matrix can be evaluated using a back-propagation procedure which is very similar to the one derived earlier for evaluating the derivatives of an error function with respect to the weights. We start by writing the element J_{ki} in the form

$$\begin{aligned} J_{ki} = \frac{\partial y_k}{\partial x_i} &= \sum_j \frac{\partial y_k}{\partial a_j} \frac{\partial a_j}{\partial x_i} \\ &= \sum_j w_{ji} \frac{\partial y_k}{\partial a_j} \end{aligned} \tag{4.52}$$

where we have made use of (4.26). The sum in (4.52) runs over all units j to which the input unit i sends connections (for example, over all units in the first hidden layer in the layered topology considered earlier). We now write down a recursive back-propagation formula to determine the derivatives $\partial y_k / \partial a_j$

$$\begin{aligned} \frac{\partial y_k}{\partial a_j} &= \sum_l \frac{\partial y_k}{\partial a_l} \frac{\partial a_l}{\partial a_j} \\ &= g'(a_j) \sum_l w_{lj} \frac{\partial y_k}{\partial a_l} \end{aligned} \tag{4.53}$$

where the sum runs over all units l to which unit j sends connections. Again, we have made use of (4.26) and (4.27). This back-propagation starts at the output units for which, using (4.27), we have

$$\frac{\partial y_k}{\partial a_{k'}} = g'(a_k)\delta_{kk'} \tag{4.54}$$

where $\delta_{kk'}$ is the Kronecker delta symbol, and equals 1 if $k = k'$ and 0 otherwise. We can therefore summarize the procedure for evaluating the Jacobian matrix as follows. Apply the input vector corresponding to the point in input space at which the Jacobian matrix is to be found, and forward propagate in the usual way to obtain the activations of all of the hidden and output units in the network. Next, for each row k of the Jacobian matrix, corresponding to the output unit k, back-propagate using the recursive relation (4.53), starting with (4.54), for all of the hidden units in the network. Finally, use (4.52) to do the back-propagation to the inputs. The second and third steps are then repeated for each value of k, corresponding to each row of the Jacobian matrix.

The Jacobian can also be evaluated using an alternative *forward* propagation formalism which can be derived in an analogous way to the back-propagation approach given here (Exercise 4.6). Again, the implementation of such algorithms can be checked by using numerical differentiation in the form

$$\frac{\partial y_k}{\partial x_i} = \frac{y_k(x_i + \epsilon) - y_k(x_i - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2). \tag{4.55}$$

4.10 The Hessian matrix

We have shown how the technique of back-propagation can be used to obtain the first derivatives of an error function with respect to the weights in the network. Back-propagation can also be used to evaluate the second derivatives of the error, given by

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}}. \tag{4.56}$$

These derivatives form the elements of the *Hessian* matrix, which plays an important role in many aspects of neural computing, including the following:

1. Several non-linear optimization algorithms used for training neural networks are based on considerations of the second-order properties of the error surface, which are controlled by the Hessian matrix (Chapter 7).
2. The Hessian forms the basis of a fast procedure for re-training a feed-forward network following a small change in the training data (Bishop, 1991a).
3. The inverse of the Hessian has been used to identify the least significant weights in a network as part of network 'pruning' algorithms (Section 9.5.3).
4. The inverse of the Hessian can also be used to assign error bars to the predictions made by a trained network (Section 10.2).

5. Suitable values for regularization parameters can be determined from the eigenvalues of the Hessian (Section 10.4).
6. The determinant of the Hessian can be used to compare the relative probabilities of different network models (Section 10.6).

For many of these applications, various approximation schemes have been used to evaluate the Hessian matrix. However, the Hessian can also be calculated exactly using an extension of the back-propagation technique for evaluating the first derivatives of the error function.

An important consideration for many applications of the Hessian is the efficiency with which it can be evaluated. If there are W parameters (weights and biases) in the network then the Hessian matrix has dimensions $W \times W$ and so the computational effort needed to evaluate the Hessian must scale at least like $\mathcal{O}(W^2)$ for each pattern in the data set. As we shall see, there are efficient methods for evaluating the Hessian whose scaling is indeed $\mathcal{O}(W^2)$.

4.10.1 *Diagonal approximation*

Some of the applications for the Hessian matrix discussed above require the inverse of the Hessian, rather than the Hessian itself. For this reason there has been some interest in using a diagonal approximation to the Hessian, since its inverse is trivial to evaluate. We again shall assume, as is generally the case, that the error function consists of a sum of terms, one for each pattern in the data set, so that $E = \sum_n E^n$. The Hessian can then be obtained by considering one pattern at a time, and then summing the results over all patterns. From (4.26) the diagonal elements of the Hessian, for pattern n, can be written

$$\frac{\partial^2 E^n}{\partial w_{ji}^2} = \frac{\partial^2 E^n}{\partial a_j^2} z_i^2. \tag{4.57}$$

Using (4.26) and (4.27), the second derivatives on the right-hand side of (4.57) can be found recursively using the chain rule of differential calculus, to give a back-propagation equation of the form

$$\frac{\partial^2 E^n}{\partial a_j^2} = g'(a_j)^2 \sum_k \sum_{k'} w_{kj} w_{k'j} \frac{\partial^2 E^n}{\partial a_k \partial a_{k'}} + g''(a_j) \sum_k w_{kj} \frac{\partial E^n}{\partial a_k}. \tag{4.58}$$

If we now neglect off-diagonal elements in the second derivative terms we obtain (Becker and Le Cun, 1989; Le Cun *et al.*, 1990)

$$\frac{\partial^2 E^n}{\partial a_j^2} = g'(a_j)^2 \sum_k w_{kj}^2 \frac{\partial^2 E^n}{\partial a_k^2} + g''(a_j) \sum_k w_{kj} \frac{\partial E^n}{\partial a_k}. \tag{4.59}$$

Due to the neglect of off-diagonal terms on the right-hand side of (4.59), this approach only gives an approximation to the diagonal terms of the Hessian.

However, the number of computational steps is reduced from $\mathcal{O}(W^2)$ to $\mathcal{O}(W)$.

Ricotti *et al.* (1988) also used the diagonal approximation to the Hessian, but they retained all terms in the evaluation of $\partial^2 E^n / \partial a_j^2$ and so obtained exact expressions for the diagonal terms. Note that this no longer has $\mathcal{O}(W)$ scaling. The major problem with diagonal approximations, however, is that in practice the Hessian is typically found to be strongly non-diagonal, and so these approximations, which are driven mainly be computational convenience, must be treated with great care.

4.10.2 *Outer product approximation*

When neural networks are applied to regression problems, it is common to use a sum-of-squares error function of the form

$$E = \frac{1}{2} \sum_n (y^n - t^n)^2 \tag{4.60}$$

where we have considered the case of a single output in order to keep the notation simple (the extension to several outputs is straightforward). We can then write the elements of the Hessian in the form

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}} = \sum_n \frac{\partial y^n}{\partial w_{ji}} \frac{\partial y^n}{\partial w_{lk}} + \sum_n (y^n - t^n) \frac{\partial^2 y^n}{\partial w_{ji} \partial w_{lk}}. \tag{4.61}$$

If the network has been trained on the data set and its outputs y^n happen to be very close to the target values t^n then the second term in (4.61) will be small and can be neglected. If the data are noisy, however, such a network mapping is severely over-fitted to the data, and is not the kind of mapping we seek in order to achieve good generalization (see Chapters 1 and 9). Instead we want to find a mapping which averages over the noise in the data. It turns out that for such a solution we may still be able to neglect the second term in (4.61). This follows from the fact that the quantity $(y^n - t^n)$ is a random variable with zero mean, which is uncorrelated with the value of the second derivative term on the right-hand side of (4.61). This whole term will therefore tend to average to zero in the summation over n (Hassibi and Stork, 1993). A more formal derivation of this result is given in Section 6.1.4.

By neglecting the second term in (4.61) we arrive at the Levenberg–Marquardt approximation (Levenberg, 1944; Marquardt, 1963) or *outer product* approximation (since the Hessian matrix is built up from a sum of outer products of vectors), given by

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{lk}} = \sum_n \frac{\partial y^n}{\partial w_{ji}} \frac{\partial y^n}{\partial w_{lk}}. \tag{4.62}$$

Its evaluation is straightforward as it only involves first derivatives of the error

function, which can be evaluated efficiently in $\mathcal{O}(W)$ steps using standard back-propagation. The elements of the matrix can then be found in $\mathcal{O}(W^2)$ steps by simple multiplication. It is important to emphasize that this approximation is only likely to be valid for a network which has been trained correctly on the same data set used to evaluate the Hessian, or on one with the same statistical properties. For a general network mapping, the second derivative terms on the right-hand side of (4.61) will typically not be negligible.

4.10.3 *Inverse Hessian*

Hassibi and Stork (1993) have used the outer product approximation to develop a computationally efficient procedure for approximating the inverse of the Hessian. We first write the outer product approximation in matrix notation as

$$\mathbf{H}_N = \sum_{n=1}^{N} \mathbf{g}^n (\mathbf{g}^n)^{\mathrm{T}} \tag{4.63}$$

where N is the number of patterns in the data set, and the vector $\mathbf{g} \equiv \nabla_{\mathbf{w}} E$ is the gradient of the error function. This leads to a sequential procedure for building up the Hessian, obtained by separating off the contribution from data point $N+1$ to give

$$\mathbf{H}_{N+1} = \mathbf{H}_N + \mathbf{g}^{N+1} (\mathbf{g}^{N+1})^{\mathrm{T}}. \tag{4.64}$$

In order to evaluate the inverse of the Hessian we now consider the matrix identity (Kailath, 1980)

$$(\mathbf{A} + \mathbf{BC})^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{B}(\mathbf{I} + \mathbf{CA}^{-1}\mathbf{B})^{-1}\mathbf{CA}^{-1} \tag{4.65}$$

where $\mathbf{I}$ is the unit matrix. If we now identify $\mathbf{H}_N$ with $\mathbf{A}$, $\mathbf{g}^{N+1}$ with $\mathbf{B}$, and $(\mathbf{g}^{N+1})^{\mathrm{T}}$ with $\mathbf{C}$, then we can apply (4.65) to (4.64) to obtain

$$\mathbf{H}_{N+1}^{-1} = \mathbf{H}_N^{-1} - \frac{\mathbf{H}_N^{-1}\mathbf{g}^{N+1}(\mathbf{g}^{N+1})^{\mathrm{T}}\mathbf{H}_N^{-1}}{1 + (\mathbf{g}^{N+1})^{\mathrm{T}}\mathbf{H}_N^{-1}\mathbf{g}^{N+1}}. \tag{4.66}$$

This represents a procedure for evaluating the inverse of the Hessian using a single pass through the data set. The initial matrix $\mathbf{H}_0$ is chosen to be $\alpha\mathbf{I}$, where α is a small quantity, so that the algorithm actually finds the inverse of $\mathbf{H} + \alpha\mathbf{I}$. The results are not particularly sensitive to the precise value of α. Extension of this algorithm to networks having more than one output is straightforward (Exercise 4.9).

We note here that the Hessian matrix can sometimes be calculated indirectly as part of the network training algorithm. In particular, quasi-Newton non-linear optimization algorithms gradually build up an approximation to the inverse of the Hessian during training. Such algorithms are discussed in detail in

Section 7.10.

4.10.4 *Finite differences*

As with first derivatives of the error function, we can find the second derivatives by using finite differences, with accuracy limited by the numerical precision of our computer. If we perturb each possible pair of weights in turn, we obtain

$$\frac{\partial^2 E}{\partial w_{ji}\partial w_{lk}} = \frac{1}{4\epsilon^2}\{E(w_{ji}+\epsilon, w_{lk}+\epsilon) - E(w_{ji}+\epsilon, w_{lk}-\epsilon)$$

$$-E(w_{ji}-\epsilon, w_{lk}+\epsilon) + E(w_{ji}-\epsilon, w_{lk}-\epsilon)\} + \mathcal{O}(\epsilon^2). \quad (4.67)$$

Again, by using a symmetrical central differences formulation, we ensure that the residual errors are $\mathcal{O}(\epsilon^2)$ rather than $\mathcal{O}(\epsilon)$. Since there are W^2 elements in the Hessian matrix, and since the evaluation of each element requires four forward propagations each needing $\mathcal{O}(W)$ operations (per pattern), we see that this approach will require $\mathcal{O}(W^3)$ operations to evaluate the complete Hessian. It therefore has very poor scaling properties, although in practice it is very useful as a check on the software implementation of back-propagation methods.

A more efficient version of numerical differentiation can be found by applying central differences to the first derivatives of the error function, which are themselves calculated using back-propagation. This gives

$$\frac{\partial^2 E}{\partial w_{ji}\partial w_{lk}} = \frac{1}{2\epsilon}\left\{\frac{\partial E}{\partial w_{ji}}(w_{lk}+\epsilon) - \frac{\partial E}{\partial w_{ji}}(w_{lk}-\epsilon)\right\} + \mathcal{O}(\epsilon^2). \quad (4.68)$$

Since there are now only W weights to be perturbed, and since the gradients can be evaluated in $\mathcal{O}(W)$ steps, we see that this method gives the Hessian in $\mathcal{O}(W^2)$ operations.

4.10.5 *Exact evaluation of the Hessian*

So far we have considered various approximation schemes for evaluating the Hessian matrix. We now describe an algorithm for evaluating the Hessian exactly, which is valid for a network of arbitrary feed-forward topology, of the kind illustrated schematically in Figure 4.3 (Bishop, 1991a, 1992). The algorithm is based on an extension of the technique of back-propagation used to evaluate first derivatives, and shares many of its desirable features including computational efficiency. It can be applied to any differentiable error function which can be expressed as a function of the network outputs, and to networks having arbitrary differentiable activation functions. The number of computational steps needed to evaluate the Hessian scales like $\mathcal{O}(W^2)$. Similar algorithms have also been considered by Buntine and Weigend (1993). As before, we shall consider one pattern at a time. The complete Hessian is then obtained by summing over all patterns.

Consider the general expression (4.33) for the derivative of the error function with respect to an arbitrary weight w_{lk}, which we reproduce here for convenience

$$\frac{\partial E^n}{\partial w_{lk}} = \delta_l z_k. \tag{4.69}$$

Differentiating this with respect to some other weight w_{ji} we obtain

$$\frac{\partial^2 E^n}{\partial w_{ji} \partial w_{lk}} = \frac{\partial a_j}{\partial w_{ji}} \frac{\partial}{\partial a_j} \left(\frac{\partial E^n}{\partial w_{lk}} \right) = z_i \frac{\partial}{\partial a_j} \left(\frac{\partial E^n}{\partial w_{lk}} \right) \tag{4.70}$$

where we have used (4.26). Here we have assumed that the weight w_{ji} does not occur on any forward propagation path connecting unit l to the outputs of the network. We shall return to this point shortly.

Making use of (4.69), together with the relation $z_k = g(a_k)$, we can write (4.70) in the form

$$\frac{\partial^2 E^n}{\partial w_{ji} \partial w_{lk}} = z_i \delta_l g'(a_k) h_{kj} + z_i z_k b_{lj} \tag{4.71}$$

where we have defined the quantities

$$h_{kj} \equiv \frac{\partial a_k}{\partial a_j} \tag{4.72}$$

$$b_{lj} \equiv \frac{\partial \delta_l}{\partial a_j}. \tag{4.73}$$

The quantities $\{h_{kj}\}$ can be evaluated by forward propagation as follows. Using the chain rule for partial derivatives we have

$$h_{kj} = \sum_r \frac{\partial a_k}{\partial a_r} \frac{\partial a_r}{\partial a_j} \tag{4.74}$$

where the sum runs over all units r which send connections to unit k. In fact, contributions only arise from units which lie on paths connecting unit j to unit k. From (4.26) and (4.27) we then obtain the forward propagation equation

$$h_{kj} = \sum_r g'(a_r) w_{kr} h_{rj}. \tag{4.75}$$

The initial conditions for evaluating the $\{h_{kj}\}$ follow from the definition (4.72), and can be stated as follows. For each unit j in the network, (except for input units, for which the corresponding $\{h_{kj}\}$ are not required), set $h_{jj} = 1$ and set

$h_{kj} = 0$ for all units $k \neq j$ which do not lie on any forward propagation path starting from unit j. The remaining elements of h_{kj} can then be found by forward propagation using (4.75).

Similarly, we can derive a back-propagation equation which allows the $\{b_{lj}\}$ to be evaluated. We have already seen that the quantities δ_l can be found by back-propagation

$$\delta_l = g'(a_l) \sum_s w_{sl} \delta_s. \tag{4.76}$$

Substituting this into the definition of b_{lj} in (4.73) we obtain

$$b_{lj} = \frac{\partial}{\partial a_j} \left\{ g'(a_l) \sum_s w_{sl} \delta_s \right\} \tag{4.77}$$

which gives

$$b_{lj} = g''(a_l) h_{lj} \sum_s w_{sl} \delta_s + g'(a_l) \sum_s w_{sl} b_{sj} \tag{4.78}$$

where the sums run over all units s to which unit l sends connections. Note that, in a software implementation, the first summation in (4.78) will already have been computed in evaluating the $\{\delta_l\}$ in (4.76).

There is one subtlety which needs to be considered. The derivative $\partial/\partial a_j$ which appears in (4.77) arose from the derivative $\partial/\partial w_{ji}$ in (4.70). This transformation, from w_{ji} to a_j, is valid provided w_{ji} does not appear explicitly within the brackets on the right-hand side of (4.77). In other words, the weight w_{ji} should not lie on any of the forward-propagation paths from unit l to the outputs of the network, since these are also the paths used to evaluate δ_l by back-propagation. In practice the problem is easily avoided as follows. If w_{ji} does occur in the sequence of back-propagations needed to evaluate δ_l, then we simply consider instead the diagonally opposite element of the Hessian matrix for which this problem will not arise (since the network has a feed-forward topology). We then make use of the fact that the Hessian is a symmetric matrix.

The initial conditions for the back-propagation in (4.78) follow from (4.72) and (4.73), together with the initial conditions (4.34) for the δ's, to give

$$b_{kj} = \sum_{k'} H_{kk'} h_{k'j} \tag{4.79}$$

where we have defined

$$H_{kk'} \equiv \frac{\partial^2 E^n}{\partial a_k \partial a'_k}. \tag{4.80}$$

This algorithm represents a straightforward extension of the usual forward and backward propagation procedures used to find the first derivatives of the error function. We can summarize the algorithm in five steps:

1. Evaluate the activations of all of the hidden and output units, for a given input pattern, by using the usual forward propagation equations. Similarly, compute the initial conditions for the h_{kj} and forward propagate through the network using (4.75) to find the remaining non-zero elements of h_{kj}.
2. Evaluate δ_k for the output units in the usual way. Similarly, evaluate the H_k for all the output units using (4.80).
3. Use the standard back-propagation equations to find δ_j for all hidden units in the network. Similarly, back-propagate to find the $\{b_{lj}\}$ by using (4.78) with initial conditions given by (4.79).
4. Evaluate the elements of the Hessian for this input pattern using (4.71).
5. Repeat the above steps for each pattern in the training set, and then sum to obtain the full Hessian.

In a practical implementation, we substitute appropriate expressions for the error function and the activation functions. For the sum-of-squares error function and linear output units, for example, we have

$$\delta_k = y_k - t_k, \qquad H_{kk'} = \delta_{kk'} \tag{4.81}$$

where $\delta_{kk'}$ is the Kronecker delta symbol.

4.10.6 *Exact Hessian for two-layer network*

As an illustration of the above algorithm, we consider the specific case of a layered network having two layers of weights. We can then use the results obtained above to write down explicit expressions for the elements of the Hessian matrix. We shall use indices i and i' to denote inputs, indices j and j' to denoted units in the hidden layer, and indices k and k' to denote outputs. Using the previous results, the Hessian matrix for this network can then be considered in three separate blocks as follows.

1. Both weights in the second layer:

$$\frac{\partial^2 E^n}{\partial w_{kj} \partial w_{k'j'}} = z_j z_{j'} \delta_{kk'} H_{kk}. \tag{4.82}$$

2. Both weights in the first layer:

$$\frac{\partial^2 E^n}{\partial w_{ji} \partial w_{j'i'}} = x_i x_{i'} g''(a_{j'}) \delta_{jj'} \sum_k w_{kj'} \delta_k + x_i x_{i'} g'(a_{j'}) g'(a_j) \sum_k w_{kj'} w_{kj} H_{kk}. \tag{4.83}$$

3. One weight in each layer:

$$\frac{\partial^2 E^n}{\partial w_{ji} \partial w_{kj'}} = x_i g'(a_j) \{\delta_k \delta_{jj'} + z_{j'} w_{kj} H_{kk}\}. \tag{4.84}$$

If one or both of the weights is a bias term, then the corresponding expressions are obtained simply by setting the appropriate activation(s) to 1.

4.10.7 *Fast multiplication by the Hessian*

In some applications of the Hessian, the quantity of interest is not the Hessian matrix $\mathbf{H}$ itself, but the product of $\mathbf{H}$ with some vector $\mathbf{v}$. We have seen that the evaluation of the Hessian takes $\mathcal{O}(W^2)$ operations, and it also requires storage which is $\mathcal{O}(W^2)$. The vector $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ which we wish to calculate itself only has W elements, so instead of computing the Hessian as an intermediate step, we can instead try to find an efficient approach to evaluating $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ directly, which requires only $\mathcal{O}(W)$ operations.

We first note that

$$\mathbf{v}^{\mathrm{T}}\mathbf{H} \equiv \mathbf{v}^{\mathrm{T}} \nabla (\nabla E) \tag{4.85}$$

where ∇ denotes the gradient operator in weight space. We can then estimate the right-hand side of (4.85) using finite differences to give

$$\mathbf{v}^{\mathrm{T}} \nabla (\nabla E) = \frac{\nabla E(\mathbf{w} + \epsilon \mathbf{v}) - \nabla E(\mathbf{w})}{\epsilon} + \mathcal{O}(\epsilon). \tag{4.86}$$

Thus, the quantity $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ can be found by forward propagating first with the original weights, and then with the weights perturbed by the small vector $\epsilon\mathbf{v}$. This procedure therefore takes $\mathcal{O}(W)$ operations. It was used by Le Cun *et al.* (1993) as part of a technique for on-line estimation of the learning rate parameter in gradient descent.

Note that the residual error in (4.86) can again be reduced from $\mathcal{O}(\epsilon)$ to $\mathcal{O}(\epsilon^2)$ by using central differences of the form

$$\mathbf{v}^{\mathrm{T}} \nabla (\nabla E) = \frac{\nabla E(\mathbf{w} + \epsilon \mathbf{v}) - \nabla E(\mathbf{w} - \epsilon \mathbf{v})}{2\epsilon} + \mathcal{O}(\epsilon^2) \tag{4.87}$$

which again scales as $\mathcal{O}(W)$.

The problem with a finite-difference approach is one of numerical inaccuracies. This can be resolved by adopting an analytic approach (Møller, 1993a; Pearlmutter, 1994). Suppose we write down standard forward-propagation and back-propagation equations for the evaluation of ∇E. We can then apply (4.85) to these equations to give a set of forward-propagation and back-propagation equations for the evaluation of $\mathbf{v}^{\mathrm{T}}\mathbf{H}$. This corresponds to acting on the original forward-propagation and back-propagation equations with a differential operator

$\mathbf{v}^{\mathrm{T}}\nabla$. Pearlmutter (1994) used the notation $\mathcal{R}\{\cdot\}$ to denote the operator $\mathbf{v}^{\mathrm{T}}\nabla$ and we shall follow this notation. The analysis is straightforward, and makes use of the usual rules of differential calculus, together with the result

$$\mathcal{R}\{\mathbf{w}\} = \mathbf{v}. \tag{4.88}$$

The technique is best illustrated with a simple example, and again we choose a two-layer network with linear output units and a sum-of-squares error function. As before, we consider the contribution to the error function from one pattern in the data set. The required vector is then obtained as usual by summing over the contributions from each of the patterns separately. For the two-layer network, the forward-propagation equations are given by

$$a_j = \sum_i w_{ji} x_i \tag{4.89}$$

$$z_j = g(a_j) \tag{4.90}$$

$$y_k = \sum_j w_{kj} z_j. \tag{4.91}$$

We now act on these equations using the $\mathcal{R}\{\cdot\}$ operator to obtain a set of forward propagation equations in the form

$$\mathcal{R}\{a_j\} = \sum_i v_{ji} x_i \tag{4.92}$$

$$\mathcal{R}\{z_j\} = g'(a_j)\mathcal{R}\{a_j\} \tag{4.93}$$

$$\mathcal{R}\{y_k\} = \sum_j w_{kj}\mathcal{R}\{z_j\} + \sum_j v_{kj} z_j \tag{4.94}$$

where v_{ji} is the element of the vector $\mathbf{v}$ which corresponds to the weight w_{ji}. Quantities of the form $\mathcal{R}\{z_j\}$, $\mathcal{R}\{a_j\}$ and $\mathcal{R}\{y_k\}$ are to be regarded as new variables whose values are found using the above equations.

Since we are considering a sum-of-squares error function, we have the following standard back-propagation expressions:

$$\delta_k = y_k - t_k \tag{4.95}$$

$$\delta_j = g'(a_j) \sum_k w_{kj} \delta_k. \tag{4.96}$$

Again we act on these equations with the $\mathcal{R}\{\cdot\}$ operator to obtain a set of back-propagation equations in the form

$$\mathcal{R}\{\delta_k\} = \mathcal{R}\{y_k\} \tag{4.97}$$

$$\begin{aligned}\mathcal{R}\{\delta_j\} = {} & g''(a_j)\mathcal{R}\{a_j\}\sum_k w_{kj}\delta_k \\ & + g'(a_j)\sum_k v_{kj}\delta_k \\ & + g'(a_j)\sum_k w_{kj}\mathcal{R}\{\delta_k\}.\end{aligned} \tag{4.98}$$

Finally, we have the usual equations for the first derivatives of the error

$$\frac{\partial E}{\partial w_{kj}} = \delta_k z_j \tag{4.99}$$

$$\frac{\partial E}{\partial w_{ji}} = \delta_j x_i \tag{4.100}$$

and acting on these with the $\mathcal{R}\{\cdot\}$ operator we obtain expressions for the elements of the vector $\mathbf{v}^{\mathrm{T}}\mathbf{H}$:

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{kj}}\right\} = \mathcal{R}\{\delta_k\}z_j + \delta_k\mathcal{R}\{z_j\} \tag{4.101}$$

$$\mathcal{R}\left\{\frac{\partial E}{\partial w_{ji}}\right\} = x_i\mathcal{R}\{\delta_j\}. \tag{4.102}$$

The implementation of this algorithm involves the introduction of additional variables $\mathcal{R}\{a_j\}$, $\mathcal{R}\{z_j\}$ and $\mathcal{R}\{\delta_j\}$ for the hidden units, and $\mathcal{R}\{\delta_k\}$ and $\mathcal{R}\{y_k\}$ for the output units. For each input pattern, the values of these quantities can be found using the above results, and the elements of $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ are then given by (4.101) and (4.102). An elegant aspect of this technique is that the structure of the equations for evaluating $\mathbf{v}^{\mathrm{T}}\mathbf{H}$ mirror closely those for standard forward and backward propagation, and so software implementation is straightforward.

If desired, the technique can be used to evaluate the full Hessian matrix by choosing the vector $\mathbf{v}$ to be given successively by a series of unit vectors of the form $(0, 0, \ldots, 1, \ldots, 0)$ each of which picks out one column of the Hessian. This leads to a formalism which is analytically equivalent to the back-propagation procedure of Bishop (1992), as described in Section 4.10.5, though with some loss of efficiency in a software implementation due to redundant calculations.

Exercises

4.1 ($\star$) In Section 4.4 we showed that, for networks with 'tanh' hidden unit activation functions, the network mapping is invariant if all of the weights and the bias feeding into and out of a unit have their signs changed. Demonstrate the corresponding symmetry for hidden units with logistic sigmoidal activation functions.

4.2 ($\star$) Consider a second-order network unit of the form (4.19). Use the symmetry properties of this term, together with the results of Exercises 1.7 and 1.8, to find an expression for the number of independent weight parameters and show that this is the same result as that obtained by applying symmetry considerations to the equivalent form (4.18).

4.3 ($\star$) Show, for a feed-forward network with 'tanh' hidden unit activation functions, and a sum-of-squares error function, that the origin in weight space is a stationary point of the error function.

4.4 ($\star$) Consider a layered network with d inputs, M hidden units and c output units. Write down an expression for the total number of weights and biases in the network. Consider the derivatives of the error function with respect to the weights for one input pattern only. Using the fact that these derivatives are given by equations of the form $\partial E^n / \partial w_{kj} = \delta_k z_j$, write down an expression for the number of independent derivatives.

4.5 ($\star$) Consider a layered network having second-order units of the form (4.19) in the first layer and conventional units in the remaining layers. Derive a back-propagation formalism for evaluating the derivatives of the error function with respect to any weight or bias in the network. Extend the result to general Mth-order units in the first layer.

4.6 ($\star$) In Section 4.9, a formalism was developed for evaluating the Jacobian matrix by a process of back-propagation. Derive an alternative formalism for obtaining the Jacobian matrix using *forward* propagation equations.

4.7 ($\star$) Consider a two-layer network having 20 inputs, 10 hidden units, and 5 outputs, together with a training set of 2000 patterns. Calculate roughly how long it would take to perform one evaluation of the Hessian matrix using (a) central differences based on direct error function evaluations; (b) central differences based on gradient evaluations using back-propagation; (c) the analytic expressions given in (4.82), (4.83) and (4.84). Assume that the workstation can perform 5×10^7 floating point operations per second, and that the time taken to evaluate an activation function or its derivatives can be neglected.

4.8 ($\star$) Verify the identity (4.65) by pre- and post-multiplying both sides by $\mathbf{A} + \mathbf{BC}$.

4.9 ($\star$) Extend the expression (4.63) for the outer product approximation of the Hessian to the case of $c > 1$ output units. Hence derive a recursive expression analogous to (4.64) for incrementing the number N of patterns, and a similar expression for incrementing the number c of outputs. Use these results, together with the identity (4.65), to find sequential update

expressions analogous (4.66) for finding the inverse of the Hessian by incrementally including both extra patterns and extra outputs.

4.10 ($\star\star$) Verify that the results (4.82), (4.83) and (4.84) for the Hessian matrix of a two-layer network follow from the general expressions for calculating the Hessian matrix for a network of arbitrary topology given in Section 4.10.5.

4.11 ($\star\star$) Derive the results (4.82), (4.83) and (4.84) for the exact evaluation of the Hessian matrix for a two-layer network by direct differentiation of the forward-propagation and back-propagation equations.

4.12 ($\star\star\star$) Write a software implementation of the forward and backward propagation equations for a two-layer network with 'tanh' hidden unit activation function and linear output units. Generate a data set of random input and target vectors, and set the network weights to random values. For the case of a sum-of-squares error function, evaluate the derivatives of the error with respect to the weights and biases in the network by using the central differences expression (4.47). Compare the results with those obtained using the back-propagation algorithm. Experiment with different values of ϵ, and show numerically that, for values of ϵ in an appropriate range, the two approaches give almost identical results. Plot graphs of the logarithm of the evaluation times for these two algorithms versus the logarithm of the number W of weights in the network, for networks having a range of different sizes (including networks with relatively large values of W). Hence verify the scalings with W discussed in Section 4.8.

4.13 ($\star\star\star$) Extend the software implementation of the previous exercise to include the forward and backward propagation equations for the $\mathcal{R}\{\cdot\}$ variables, described in Section 4.10.7. Use this implementation to evaluate the complete Hessian matrix by setting the vector $\mathbf{v}$ in the $\mathcal{R}\{\cdot\}$ operator to successive unit vectors of the form $(0, 0, \ldots, 1, \ldots, 0)$ each of which picks out one column of the Hessian. Also implement the central differences approach for evaluation of the Hessian given by (4.67). Show that the results from the $\mathcal{R}\{\cdot\}$ operator and central difference methods agree closely, provided ϵ is chosen appropriately. Again, plot graphs of the logarithm of the evaluation time versus the logarithm of the number of weights in the network, for networks having a range of different sizes, for both of these approaches to evaluation of the Hessian, and verify the scalings with W of the two algorithms, as discussed in the text.

4.14 ($\star\star\star$) Extend further the software implementation of Exercise 4.12 by implementing equations (4.82), (4.83) and (4.84) for computing the elements of the Hessian matrix. Show that the results agree with those from the $\mathcal{R}\{\cdot\}$-operator approach, and extend the graph of the previous exercise to include the logarithm of the computation times for this algorithm.

4.15 ($\star\star$) Consider a feed-forward network which has been trained to a minimum of some error function E, corresponding to a set of weights $\{w_j\}$, where for convenience we have labelled all of the weights and biases in the

network with a single index j. Suppose that all of the input values x_i^n and target values t_k^n in the training set are perturbed by small amounts Δx_i^n and Δt_k^n respectively. This causes the minimum of the error function to change to a new set of weight values given by $\{w_j + \Delta w_j\}$. Write down the Taylor expansion of the new error function $E(\{w_j+\Delta w_j\}, \{x_i^n+\Delta x_i^n\}, \{t_k^n+\Delta t_k^n\})$ to second order in the Δ's. By minimizing this expression with respect to the $\{\Delta w_j\}$, show that the new set of weights which minimizes the error function can be calculated from the original set of weights by adding corrections Δw_j which are given by solutions of the following equation

$$\sum_j H_{lj} \Delta w_j = -\Delta T_l, \tag{4.103}$$

where H_{lj} are the elements of the Hessian matrix, and we have defined

$$\Delta T_l \equiv \sum_n \sum_i \frac{\partial^2 E^n}{\partial x_i^n \partial w_l} \Delta x_i^n + \sum_n \sum_k \frac{\partial^2 E^n}{\partial w_l \partial t_k^n} \Delta t_k^n. \tag{4.104}$$