SIX
Multi-Layer Networks

The limitations of a simple perceptron do not apply to feed-forward networks with intermediate or "hidden" layers between the input and output layer. In fact, as we will see later, a network with just one hidden layer can represent any Boolean function (including for example XOR). Although the greater power of multi-layer networks was realized long ago, it was only recently shown how to make them *learn* a particular function, using "back-propagation" or other methods. This absence of a learning rule—together with the demonstration by Minsky and Papert [1969] that only linearly separable functions could be represented by simple perceptrons—led to a waning of interest in layered networks until recently.

Throughout this chapter, like the previous one, we consider only *feed-forward* networks. More general networks are discussed in the next chapter.

6.1 Back-Propagation

The back-propagation algorithm is central to much current work on learning in neural networks. It was invented independently several times, by Bryson and Ho [1969], Werbos [1974], Parker [1985] and Rumelhart, Hinton, and Williams [1986a, b]. A closely related approach was proposed by Le Cun [1985]. The algorithm gives a prescription for changing the weights w_{pq} in any feed-forward network to learn a training set of input-output pairs $\{\xi_k^\mu, \zeta_i^\mu\}$. The basis is simply gradient descent, as described in Sections 5.4 (linear) and 5.5 (nonlinear) for a simple perceptron.

We consider first a two-layer network such as that illustrated by Fig. 6.1. Our notational conventions are shown in the figure; output units are denoted by O_i, hidden units by V_j, and input terminals by ξ_k. There are connections w_{jk} from the

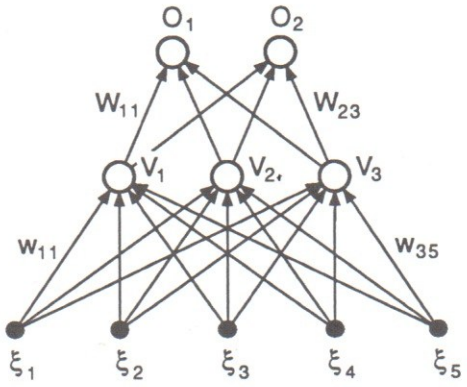

FIGURE 6.1 A two layer feed-forward network, showing the notation for units and weights.

inputs to the hidden units, and W_{ij} from the hidden units to the output units. Note that the index i always refers to an output unit, j to a hidden one, and k to an input terminal.

The inputs are always clamped to particular values. As in previous chapters, we label different patterns by a superscript μ, so input k is set to ξ_k^μ when pattern μ is being presented. The ξ_k^μ's can be binary (0/1, or ± 1) or continuous-valued. We use N for the number of *input* units and p, as before, for the number of input patterns ($\mu = 1, 2, \ldots, p$).

Given pattern μ, hidden unit j receives a net input

$$h_j^\mu = \sum_k w_{jk}\xi_k^\mu \tag{6.1}$$

and produces output

$$V_j^\mu = g(h_j^\mu) = g\Big(\sum_k w_{jk}\xi_k^\mu\Big). \tag{6.2}$$

Output unit i thus receives

$$h_i^\mu = \sum_j W_{ij}V_j^\mu = \sum_j W_{ij}g\Big(\sum_k w_{jk}\xi_k^\mu\Big) \tag{6.3}$$

and produces for the final output

$$O_i^\mu = g(h_i^\mu) = g\Big(\sum_j W_{ij}V_j^\mu\Big) = g\Big(\sum_j W_{ij}g\Big(\sum_k w_{jk}\xi_k^\mu\Big)\Big). \tag{6.4}$$

As in the previous chapter we have omitted the thresholds; they can be taken care of as usual by an extra input unit clamped to -1 and connected to all units in the network.

Our usual error measure or cost function

$$E[\mathbf{w}] = \frac{1}{2}\sum_{\mu i}[\zeta_i^\mu - O_i^\mu]^2 \tag{6.5}$$

now becomes

$$E[\mathbf{w}] = \frac{1}{2}\sum_{\mu i}\Big[\zeta_i^\mu - g\Big(\sum_j W_{ij}\, g\Big(\sum_k w_{jk}\xi_k^\mu\Big)\Big)\Big]^2 . \tag{6.6}$$

This is clearly a continuous differentiable function of every weight, so we can use a gradient descent algorithm to learn appropriate weights. In one sense this is all there is to back-propagation, but there is great practical importance in the form of the resulting update rules.

For the hidden-to-output connections the gradient descent rule gives

$$\begin{aligned}\Delta W_{ij} &= -\eta\frac{\partial E}{\partial W_{ij}} = \eta\sum_\mu[\zeta_i^\mu - O_i^\mu]g'(h_i^\mu)V_j^\mu \\ &= \eta\sum_\mu \delta_i^\mu V_j^\mu\end{aligned} \tag{6.7}$$

where we have defined

$$\delta_i^\mu = g'(h_i^\mu)[\zeta_i^\mu - O_i^\mu] . \tag{6.8}$$

The result is of course identical to that obtained earlier (equations (5.50) and (5.51)) for a single layer perceptron, with the output V_j^μ of the hidden units now playing the role of the perceptron input.

For the input-to-hidden connections Δw_{jk} we must differentiate with respect to the w_{jk}'s, which are more deeply embedded in (6.6). Using the chain rule, we obtain

$$\begin{aligned}\Delta w_{jk} &= -\eta\frac{\partial E}{\partial w_{jk}} = -\eta\sum_\mu \frac{\partial E}{\partial V_j^\mu}\frac{\partial V_j^\mu}{\partial w_{jk}} \\ &= \eta\sum_{\mu i}[\zeta_i^\mu - O_i^\mu]g'(h_i^\mu)W_{ij}g'(h_j^\mu)\xi_k^\mu \\ &= \eta\sum_{\mu i}\delta_i^\mu W_{ij}g'(h_j^\mu)\xi_k^\mu \\ &= \eta\sum_\mu \delta_j^\mu\xi_k^\mu\end{aligned} \tag{6.9}$$

with

$$\delta_j^\mu = g'(h_j^\mu)\sum_i W_{ij}\delta_i^\mu . \tag{6.10}$$

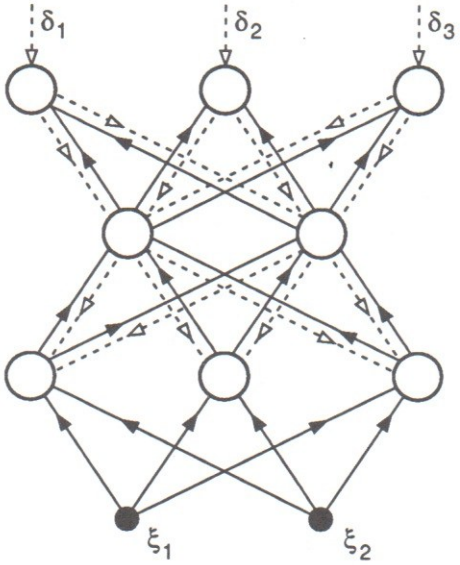

FIGURE 6.2 Back-propagation in a three-layer network. The solid lines show the forward propagation of signals and the dashed lines show the backward propagation of errors (δ's).

Note that (6.9) has the same form as (6.7), but with a different definition of the δ's. In general, with an arbitrary number of layers, the back-propagation update rule always has the form

$$\Delta w_{pq} = \eta \sum_{\text{patterns}} \delta_{\text{output}} \times V_{\text{input}} \tag{6.11}$$

where *output* and *input* refer to the two ends p and q of the connection concerned, and V stands for the appropriate input-end activation from a hidden unit or a real input. The meaning of δ depends on the layer concerned; for the last layer of connections it is given by (6.8), while for all other layers it is given by an equation like (6.10). It is easy to derive this generalized multi-layer result (6.11), simply by further application of the chain rule.

Equation (6.10) allows us to determine the δ for a given hidden unit V_j in terms of the δ's of the units O_i that it feeds. The coefficients are just the usual "forward" W_{ij}'s, but here they are propagating errors (δ's) backwards instead of signals forwards: hence the name **error back-propagation** or just **back-propagation**. We can therefore use the same network—or rather a bidirectional version of it—to compute both the output values and the δ's. Figure 6.2 illustrates this idea for a three-layer network.

Although we have written the update rules (6.7) and (6.9) as sums over all patterns μ, they are usually used incrementally: a pattern μ is presented at the input and then all weights are updated before the next pattern is considered. This clearly decreases the cost function (for small enough η) at each step, and lets successive steps adapt to the local gradient. If the patterns are chosen in random order it also

makes the path through weight-space *stochastic*, allowing wider exploration of the cost surface. The alternative **batch mode**—taking (6.7) and (6.9) literally and only updating after all patterns have been presented—requires additional local storage for each connection. The relative effectiveness of the two approaches depends on the problem, but the incremental approach seems superior in most cases, especially for very regular or redundant training sets.

The fact that the appropriate cost function derivatives can be calculated by back-propagating errors is clearly attractive. But it also has two important consequences:

- The update rule (6.11) is *local*. To compute the weight change for a given connection we only need quantities available (after back-propagation of the δ's) at the two ends of that connection. This makes the back-propagation rule appropriate for parallel computation. It may even have some indirect relevance for neurobiology.[1]
- The computational complexity is less than might have been expected. If we have n connections in all, computation of the cost function (6.6) takes of order n operations, so calculating n derivatives directly would take order n^2 operations. In contrast the back-propagation scheme lets us calculate *all* the derivatives in order n operations.

It is normal to use a sigmoid function for the activation function $g(h)$. The function clearly *must* be differentiable, and we normally want it to saturate at both extremes. Either a 0/1 or a ± 1 range can be used, with

$$g(h) = f_\beta(h) = \frac{1}{1+\exp(-2\beta h)} \tag{6.12}$$

and

$$g(h) = \tanh \beta h \tag{6.13}$$

respectively for the activation function. The steepness parameter β is often set to 1, or 1/2 for (6.12). As we noted in Chapter 5, the derivatives of these functions are readily expressed in terms of the functions themselves as $g'(h) = 2\beta g(1-g)$ for (6.12) and $g'(h) = \beta(1-g^2)$ for (6.13). Thus one often sees (6.8), for example, written as

$$\delta_i^\mu = O_i^\mu(1-O_i^\mu)(\zeta_i^\mu - O_i^\mu) \tag{6.14}$$

for 0/1 units with $\beta = 1/2$.

Because back-propagation is so important, we summarize the result in terms of a step-by-step procedure, taking one pattern μ at a time (i.e., incremental updates). We consider a network with M layers $m = 1, 2, \ldots, M$ and use V_i^m for the output

[1] Locality is necessary for biological implementation, but not sufficient. Bidirectional bifunctional connections are not biologically reasonable [Grossberg, 1987b], but can be avoided, allowing hypothetical neurophysiological implementations [Hecht-Nielsen, 1989]. Nevertheless back-propagation seems rather far-fetched as a biological learning mechanism [Crick, 1989].

of the ith unit in the mth layer. V_i^0 will be a synonym for ξ_i, the ith input. Note that superscript m's label layers, not patterns. We let w_{ij}^m mean the connection from V_j^{m-1} to V_i^m. Then the back-propagation procedure is:

1. Initialize the weights to small random values.
2. Choose a pattern ξ_k^μ and apply it to the input layer ($m = 0$) so that

$$V_k^0 = \xi_k^\mu \qquad \text{for all } k\,. \tag{6.15}$$

3. Propagate the signal forwards through the network using

$$V_i^m = g(h_i^m) = g\Big(\sum_j w_{ij}^m V_j^{m-1}\Big) \tag{6.16}$$

for each i and m until the final outputs V_i^M have all been calculated.

4. Compute the deltas for the output layer

$$\delta_i^M = g'(h_i^M)[\zeta_i^\mu - V_i^M] \tag{6.17}$$

by comparing the actual outputs V_i^M with the desired ones ζ_i^μ for the pattern μ being considered.

5. Compute the deltas for the preceding layers by propagating the errors backwards

$$\delta_i^{m-1} = g'(h_i^{m-1}) \sum_j w_{ji}^m \delta_j^m \tag{6.18}$$

for $m = M, M-1, \ldots, 2$ until a delta has been calculated for every unit.

6. Use

$$\Delta w_{ij}^m = \eta \delta_i^m V_j^{m-1} \tag{6.19}$$

to update all connections according to $w_{ij}^{\text{new}} = w_{ij}^{\text{old}} + \Delta w_{ij}$.

7. Go back to step 2 and repeat for the next pattern.

It is straightforward to generalize back-propagation to other kinds of networks where connections jump over one or more layers, such as the direct input-to-output connections in Fig. 6.5(b). This produces the same kind of error propagation scheme as long as the network is *feed-forward*, without any backward or lateral connections.