11 Backpropagation

Objectives

In this chapter we continue our discussion of performance learning, which we began in Chapter 8, by presenting a generalization of the LMS algorithm of Chapter 10. This generalization, called backpropagation, can be used to train multilayer networks. As with the LMS learning law, backpropagation is an approximate steepest descent algorithm, in which the performance index is mean square error. The difference between the LMS algorithm and backpropagation is only in the way in which the derivatives are calculated. For a single-layer linear network the error is an explicit linear function of the network weights, and its derivatives with respect to the weights can be easily computed. In multilayer networks with nonlinear transfer functions, the relationship between the network weights and the error is more complex. In order to calculate the derivatives, we need to use the chain rule of calculus. In fact, this chapter is in large part a demonstration of how to use the chain rule.

Theory and Examples

The perceptron learning rule of Frank Rosenblatt and the LMS algorithm of Bernard Widrow and Marcian Hoff were designed to train single-layer perceptron-like networks. As we have discussed in previous chapters, these single-layer networks suffer from the disadvantage that they are only able to solve linearly separable classification problems. Both Rosenblatt and Widrow were aware of these limitations and proposed multilayer networks that could overcome them, but they were not able to generalize their algorithms to train these more powerful networks.

Apparently the first description of an algorithm to train multilayer networks was contained in the thesis of Paul Werbos in 1974 [Werbo74]. This thesis presented the algorithm in the context of general networks, with neural networks as a special case, and was not disseminated in the neural network community. It was not until the mid 1980s that the backpropagation algorithm was rediscovered and widely publicized. It was rediscovered independently by David Rumelhart, Geoffrey Hinton and Ronald Williams [RuHi86], David Parker [Park85], and Yann Le Cun [LeCu85]. The algorithm was popularized by its inclusion in the book *Parallel Distributed Processing* [RuMc86], which described the work of the Parallel Distributed Processing Group led by psychologists David Rumelhart and James McClelland. The publication of this book spurred a torrent of research in neural networks. The multilayer perceptron, trained by the backpropagation algorithm, is currently the most widely used neural network.

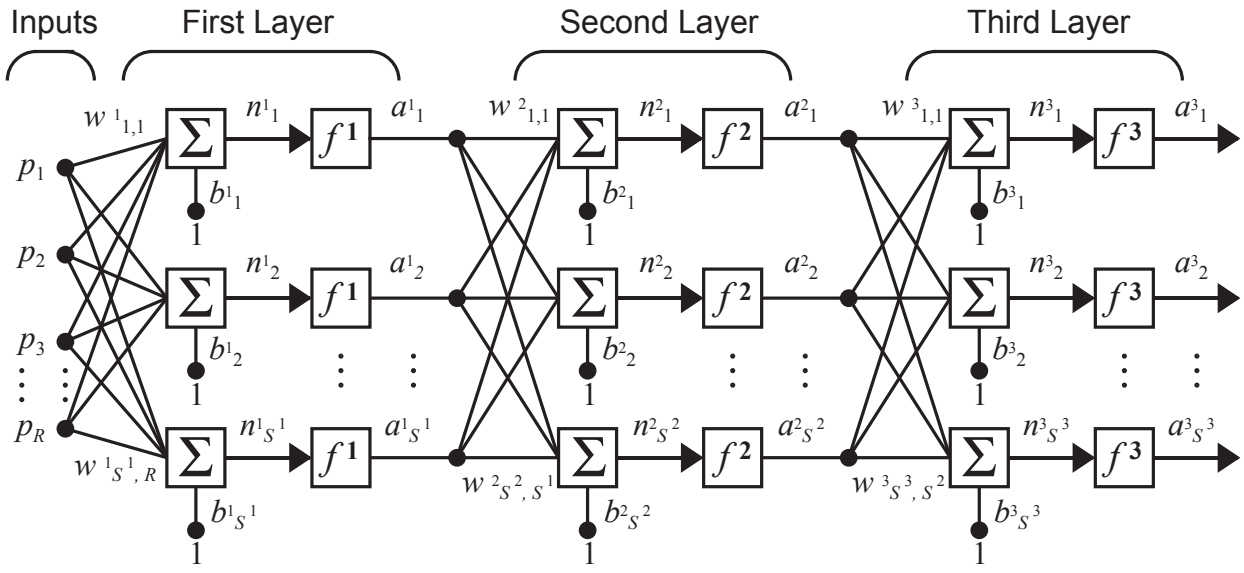
In this chapter we will first investigate the capabilities of multilayer networks and then present the backpropagation algorithm.

Multilayer Perceptrons

We first introduced the notation for multilayer networks in Chapter 2. For ease of reference we have reproduced the diagram of the three-layer perceptron in Figure 11.1. Note that we have simply cascaded three perceptron networks. The output of the first network is the input to the second network, and the output of the second network is the input to the third network. Each layer may have a different number of neurons, and even a different transfer function. Recall from Chapter 2 that we are using superscripts to identify the layer number. Thus, the weight matrix for the first layer is written as $\mathbf{W}^1$ and the weight matrix for the second layer is written $\mathbf{W}^2$.

To identify the structure of a multilayer network, we will sometimes use the following shorthand notation, where the number of inputs is followed by the number of neurons in each layer:

$$R - S^1 - S^2 - S^3 . \tag{11.1}$$



$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

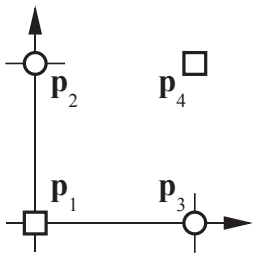
Figure 11.1 Three-Layer Network

Let's now investigate the capabilities of these multilayer perceptron networks. First we will look at the use of multilayer networks for pattern classification, and then we will discuss their application to function approximation.

Pattern Classification

To illustrate the capabilities of the multilayer perceptron for pattern classification, consider the classic exclusive-or (XOR) problem. The input/target pairs for the XOR gate are

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\}\left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$



This problem, which is illustrated graphically in the figure to the left, was used by Minsky and Papert in 1969 to demonstrate the limitations of the single-layer perceptron. Because the two categories are not linearly separable, a single-layer perceptron cannot perform the classification.

A two-layer network can solve the XOR problem. In fact, there are many different multilayer solutions. One solution is to use two neurons in the first layer to create two decision boundaries. The first boundary separates $\mathbf{p}_1$ from the other patterns, and the second boundary separates $\mathbf{p}_4$. Then the second layer is used to combine the two boundaries together using an

AND operation. The decision boundaries for each first-layer neuron are shown in Figure 11.2.

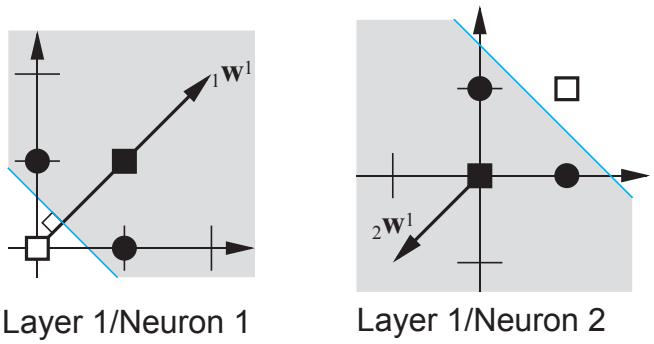
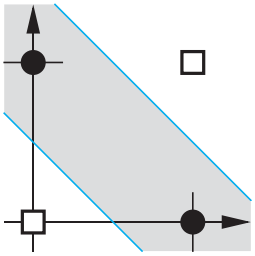


Figure 11.2 Decision Boundaries for XOR Network



The resulting two-layer, 2-2-1 network is shown in Figure 11.3. The overall decision regions for this network are shown in the figure in the left margin. The shaded region indicates those inputs that will produce a network output of 1.

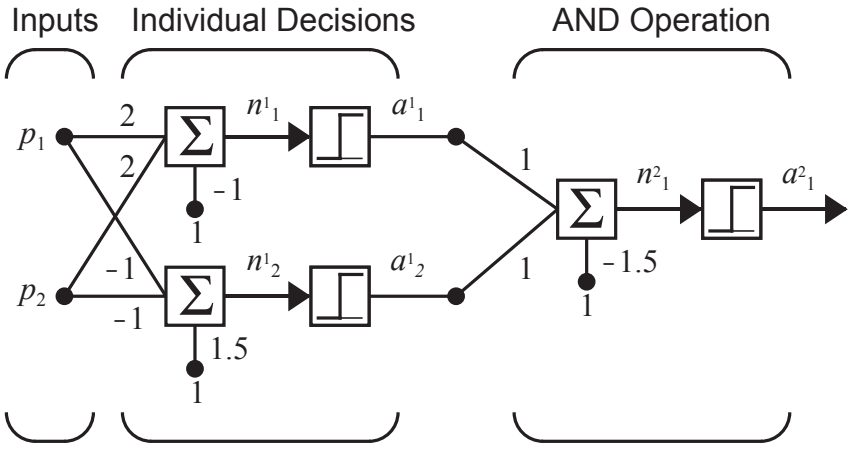


Figure 11.3 Two-Layer XOR Network

See Problems P11.1 and P11.2 for more on the use of multilayer networks for pattern classification.

Function Approximation

Up to this point in the text we have viewed neural networks mainly in the context of pattern classification. It is also instructive to view networks as function approximators. In control systems, for example, the objective is to find an appropriate feedback function that maps from measured outputs to control inputs. In adaptive filtering (Chapter 10) the objective is to find a function that maps from delayed values of an input signal to an appropriate output signal. The following example will illustrate the flexibility of the multilayer perceptron for implementing functions.

Consider the two-layer, 1-2-1 network shown in Figure 11.4. For this example the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. In other words,

$$f^1(n) = \frac{1}{1+e^{-n}} \text{ and } f^2(n) = n. \tag{11.2}$$

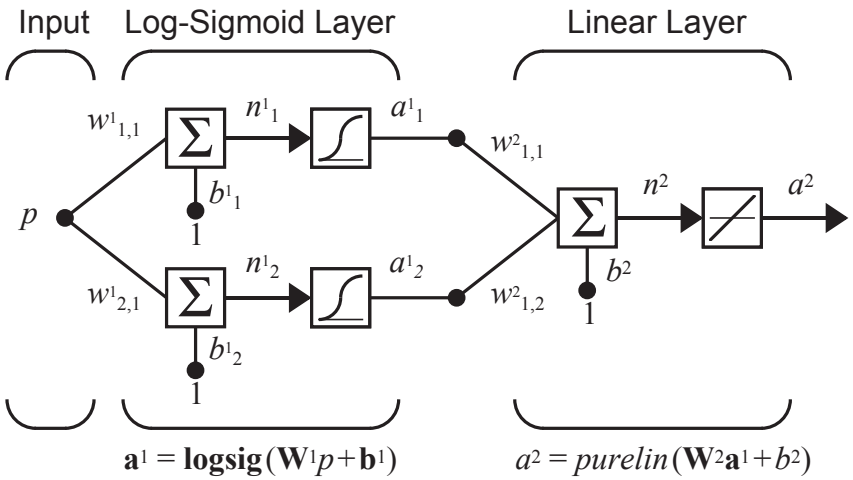


Figure 11.4 Example Function Approximation Network

Suppose that the nominal values of the weights and biases for this network are

$$w^1_{1,1} = 10,\ w^1_{2,1} = 10,\ b^1_1 = -10,\ b^1_2 = 10,$$

$$w^2_{1,1} = 1,\ w^2_{1,2} = 1,\ b^2 = 0.$$

The network response for these parameters is shown in Figure 11.5, which plots the network output a^2 as the input p is varied over the range $[-2, 2]$.

Notice that the response consists of two steps, one for each of the log-sigmoid neurons in the first layer. By adjusting the network parameters we can change the shape and location of each step, as we will see in the following discussion.

The centers of the steps occur where the net input to a neuron in the first layer is zero:

$$n^1_1 = w^1_{1,1}p + b^1_1 = 0 \quad \Rightarrow \quad p = -\frac{b^1_1}{w^1_{1,1}} = -\frac{-10}{10} = 1, \tag{11.3}$$

$$n_2^1 = w_{2,1}^1 p + b_2^1 = 0 \quad \Rightarrow \quad p = -\frac{b_2^1}{w_{2,1}^1} = -\frac{10}{10} = -1 . \tag{11.4}$$

The steepness of each step can be adjusted by changing the network weights.

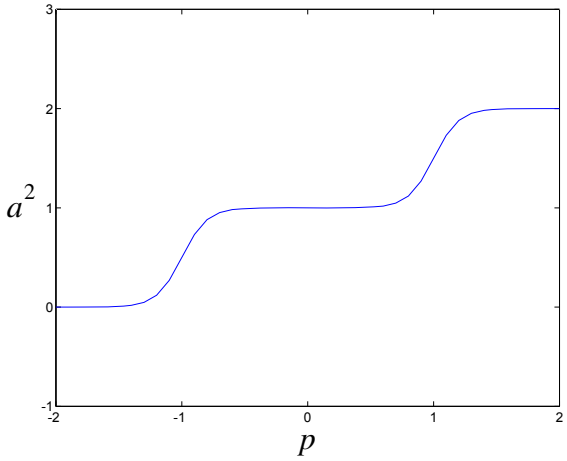


Figure 11.5 Nominal Response of Network of Figure 11.4

Figure 11.6 illustrates the effects of parameter changes on the network response. The blue curve is the nominal response. The other curves correspond to the network response when one parameter at a time is varied over the following ranges:

$$-1 \le w_{1,1}^2 \le 1 , \; -1 \le w_{1,2}^2 \le 1 , \; 0 \le b_2^1 \le 20 , \; -1 \le b^2 \le 1 . \tag{11.5}$$

Figure 11.6 (a) shows how the network biases in the first (hidden) layer can be used to locate the position of the steps. Figure 11.6 (b) illustrates how the weights determine the slope of the steps. The bias in the second (output) layer shifts the entire network response up or down, as can be seen in Figure 11.6 (d).

From this example we can see how flexible the multilayer network is. It would appear that we could use such networks to approximate almost any function, if we had a sufficient number of neurons in the hidden layer. In fact, it has been shown that two-layer networks, with sigmoid transfer functions in the hidden layer and linear transfer functions in the output layer, can approximate virtually any function of interest to any degree of accuracy, provided sufficiently many hidden units are available (see [HoSt89]).

To experiment with the response of this two-layer network, use the MATLAB® Neural Network Design Demonstration Network Function (`nnd11nf`).

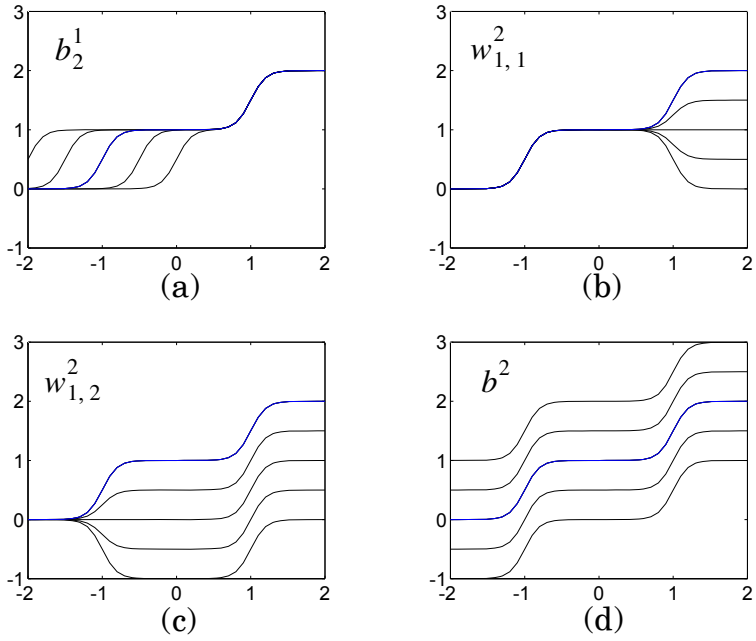


Figure 11.6 Effect of Parameter Changes on Network Response

Now that we have some idea of the power of multilayer perceptron networks for pattern recognition and function approximation, the next step is to develop an algorithm to train such networks.

The Backpropagation Algorithm

It will simplify our development of the backpropagation algorithm if we use the abbreviated notation for the multilayer network, which we introduced in Chapter 2. The three-layer network in abbreviated notation is shown in Figure 11.7.

As we discussed earlier, for multilayer networks the output of one layer becomes the input to the following layer. The equations that describe this operation are

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1, \qquad (11.6)$$

where M is the number of layers in the network. The neurons in the first layer receive external inputs:

$$\mathbf{a}^0 = \mathbf{p}, \qquad (11.7)$$

which provides the starting point for Eq. (11.6). The outputs of the neurons in the last layer are considered the network outputs:

$$\mathbf{a} = \mathbf{a}^M. \qquad (11.8)$$

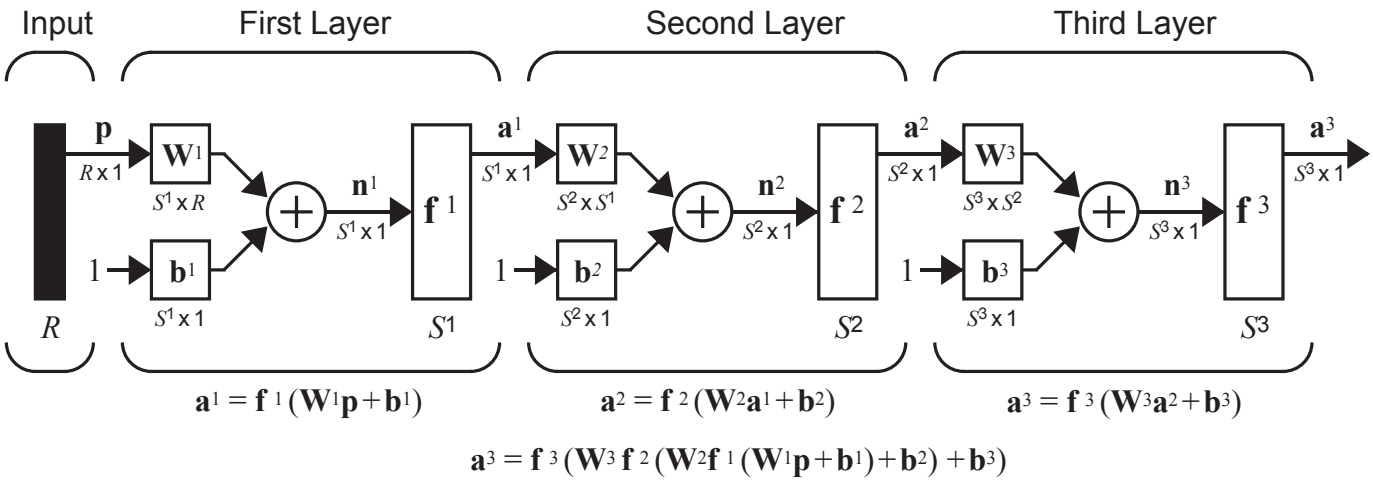


Figure 11.7 Three-Layer Network, Abbreviated Notation

Performance Index

The backpropagation algorithm for multilayer networks is a generalization of the LMS algorithm of Chapter 10, and both algorithms use the same performance index: *mean square error*. The algorithm is provided with a set of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{11.9}$$

where $\mathbf{p}_q$ is an input to the network, and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The algorithm should adjust the network parameters in order to minimize the mean square error:

$$F(\mathbf{x}) = E[e^2] = E[(t-a)^2]. \tag{11.10}$$

where $\mathbf{x}$ is the vector of network weights and biases (as in Chapter 10). If the network has multiple outputs this generalizes to

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]. \tag{11.11}$$

As with the LMS algorithm, we will approximate the mean square error by

$$\hat{F}(\mathbf{x}) = (\mathbf{t}(k)-\mathbf{a}(k))^T(\mathbf{t}(k)-\mathbf{a}(k)) = \mathbf{e}^T(k)\mathbf{e}(k), \tag{11.12}$$

where the expectation of the squared error has been replaced by the squared error at iteration k.

The steepest descent algorithm for the approximate mean square error (stochastic gradient descent) is

$$w_{i,j}^{m}(k+1) = w_{i,j}^{m}(k) - \alpha \frac{\partial \hat{F}}{\partial w_{i,j}^{m}}, \tag{11.13}$$

$$b_{i}^{m}(k+1) = b_{i}^{m}(k) - \alpha \frac{\partial \hat{F}}{\partial b_{i}^{m}}, \tag{11.14}$$

where α is the learning rate.

So far, this development is identical to that for the LMS algorithm. Now we come to the difficult part – the computation of the partial derivatives.

Chain Rule

For a single-layer linear network (the ADALINE) these partial derivatives are conveniently computed using Eq. (10.33) and Eq. (10.34). For the multilayer network the error is not an explicit function of the weights in the hidden layers, therefore these derivatives are not computed so easily.

Because the error is an indirect function of the weights in the hidden layers, we will use the chain rule of calculus to calculate the derivatives. To review the chain rule, suppose that we have a function f that is an explicit function only of the variable n. We want to take the derivative of f with respect to a third variable w. The chain rule is then:

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw}. \tag{11.15}$$

For example, if

$$f(n) = e^{n} \text{ and } n = 2w, \text{ so that } f(n(w)) = e^{2w}, \tag{11.16}$$

then

$$\frac{df(n(w))}{dw} = \frac{df(n)}{dn} \times \frac{dn(w)}{dw} = (e^{n})(2). \tag{11.17}$$

We will use this concept to find the derivatives in Eq. (11.13) and Eq. (11.14):

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial w_{i,j}^{m}}, \tag{11.18}$$

$$\frac{\partial \hat{F}}{\partial b_{i}^{m}} = \frac{\partial \hat{F}}{\partial n_{i}^{m}} \times \frac{\partial n_{i}^{m}}{\partial b_{i}^{m}}. \tag{11.19}$$

The second term in each of these equations can be easily computed, since the net input to layer m is an explicit function of the weights and bias in that layer:

$$n_i^m = \sum_{j=1}^{S^{m-1}} w_{i,j}^m a_j^{m-1} + b_i^m . \tag{11.20}$$

Therefore

$$\frac{\partial n_i^m}{\partial w_{i,j}^m} = a_j^{m-1}, \quad \frac{\partial n_i^m}{\partial b_i^m} = 1 . \tag{11.21}$$

If we now define

$$s_i^m \equiv \frac{\partial \hat{F}}{\partial n_i^m}, \tag{11.22}$$

Sensitivity

(the *sensitivity* of $\hat{F}$ to changes in the ith element of the net input at layer m), then Eq. (11.18) and Eq. (11.19) can be simplified to

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = s_i^m a_j^{m-1}, \tag{11.23}$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = s_i^m . \tag{11.24}$$

We can now express the approximate steepest descent algorithm as

$$w_{i,j}^m(k+1) = w_{i,j}^m(k) - \alpha s_i^m a_j^{m-1}, \tag{11.25}$$

$$b_i^m(k+1) = b_i^m(k) - \alpha s_i^m . \tag{11.26}$$

In matrix form this becomes:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \tag{11.27}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m, \tag{11.28}$$

where

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}. \tag{11.29}$$

(Note the close relationship between this algorithm and the LMS algorithm of Eq. (10.33) and Eq. (10.34)).

Backpropagating the Sensitivities

It now remains for us to compute the sensitivities $\mathbf{s}^m$, which requires another application of the chain rule. It is this process that gives us the term *backpropagation*, because it describes a recurrence relationship in which the sensitivity at layer m is computed from the sensitivity at layer $m+1$.

To derive the recurrence relationship for the sensitivities, we will use the following Jacobian matrix:

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} \equiv \begin{bmatrix} \frac{\partial n_1^{m+1}}{\partial n_1^m} & \frac{\partial n_1^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_1^{m+1}}{\partial n_{S^m}^m} \\ \frac{\partial n_2^{m+1}}{\partial n_1^m} & \frac{\partial n_2^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_2^{m+1}}{\partial n_{S^m}^m} \\ \vdots & \vdots & & \vdots \\ \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_1^m} & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_2^m} & \cdots & \frac{\partial n_{S^{m+1}}^{m+1}}{\partial n_{S^m}^m} \end{bmatrix}. \tag{11.30}$$

Next we want to find an expression for this matrix. Consider the i,j element of the matrix:

$$\frac{\partial n_i^{m+1}}{\partial n_j^m} = \frac{\partial\left(\sum_{l=1}^{S^m} w_{i,l}^{m+1} a_l^m + b_i^{m+1}\right)}{\partial n_j^m} = w_{i,j}^{m+1}\frac{\partial a_j^m}{\partial n_j^m}$$

$$= w_{i,j}^{m+1}\frac{\partial f^m(n_j^m)}{\partial n_j^m} = w_{i,j}^{m+1}\dot{f}^m(n_j^m)\,, \tag{11.31}$$

where

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}\,. \tag{11.32}$$

Therefore the Jacobian matrix can be written

$$\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m} = \mathbf{W}^{m+1}\dot{\mathbf{F}}^m(\mathbf{n}^m)\,, \tag{11.33}$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \dots & 0 \\ 0 & \dot{f}^m(n_2^m) & \dots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & \dots & \dot{f}^m(n_{S^m}^m) \end{bmatrix}. \tag{11.34}$$

We can now write out the recurrence relation for the sensitivity by using the chain rule in matrix form:

$$\mathbf{s}^m = \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \left(\frac{\partial \mathbf{n}^{m+1}}{\partial \mathbf{n}^m}\right)^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}} = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \frac{\partial \hat{F}}{\partial \mathbf{n}^{m+1}}$$

$$= \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T \mathbf{s}^{m+1}\,. \tag{11.35}$$

Now we can see where the backpropagation algorithm derives its name. The sensitivities are propagated backward through the network from the last layer to the first layer:

$$\mathbf{s}^M \rightarrow \mathbf{s}^{M-1} \rightarrow \dots \rightarrow \mathbf{s}^2 \rightarrow \mathbf{s}^1\,. \tag{11.36}$$

At this point it is worth emphasizing that the backpropagation algorithm uses the same approximate steepest descent technique that we used in the LMS algorithm. The only complication is that in order to compute the gradient we need to first backpropagate the sensitivities. The beauty of backpropagation is that we have a very efficient implementation of the chain rule.

We still have one more step to make in order to complete the backpropagation algorithm. We need the starting point, $\mathbf{s}^M$, for the recurrence relation of Eq. (11.35). This is obtained at the final layer:

$$s_i^M = \frac{\partial \hat{F}}{\partial n_i^M} = \frac{\partial (\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})}{\partial n_i^M} = \frac{\partial \sum_{j=1}^{S^M}(t_j - a_j)^2}{\partial n_i^M} = -2(t_i - a_i)\frac{\partial a_i}{\partial n_i^M}. \quad (11.37)$$

Now, since

$$\frac{\partial a_i}{\partial n_i^M} = \frac{\partial a_i^M}{\partial n_i^M} = \frac{\partial f^M(n_i^M)}{\partial n_i^M} = \dot{f}^M(n_i^M), \quad (11.38)$$

we can write

$$s_i^M = -2(t_i - a_i)\dot{f}^M(n_i^M). \quad (11.39)$$

This can be expressed in matrix form as

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}). \quad (11.40)$$

Summary

Let's summarize the backpropagation algorithm. The first step is to propagate the input forward through the network:

$$\mathbf{a}^0 = \mathbf{p}, \quad (11.41)$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1, \quad (11.42)$$

$$\mathbf{a} = \mathbf{a}^M. \quad (11.43)$$

The next step is to propagate the sensitivities backward through the network:

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t}-\mathbf{a}), \quad (11.44)$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \ldots, 2, 1. \tag{11.45}$$

Finally, the weights and biases are updated using the approximate steepest descent rule:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T, \tag{11.46}$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m. \tag{11.47}$$

Example

To illustrate the backpropagation algorithm, let's choose a network and apply it to a particular problem. To begin, we will use the 1-2-1 network that we discussed earlier in this chapter. For convenience we have reproduced the network in Figure 11.8.

Next we want to define a problem for the network to solve. Suppose that we want to use the network to approximate the function

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{11.48}$$

To obtain our training set we will evaluate this function at several values of p.

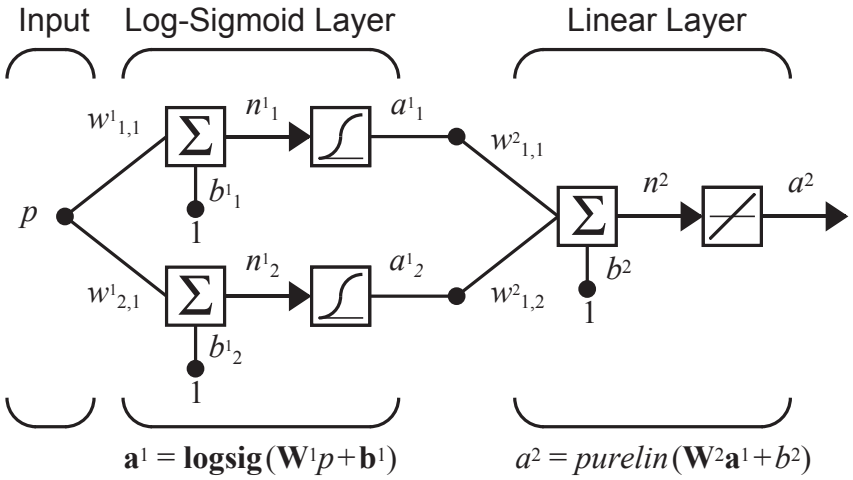


Figure 11.8 Example Function Approximation Network

Before we begin the backpropagation algorithm we need to choose some initial values for the network weights and biases. Generally these are chosen to be small random values. In the next chapter we will discuss some reasons for this. For now let's choose the values

$$\mathbf{W}^1(0) = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 0.48 \end{bmatrix}.$$

The response of the network for these initial values is illustrated in Figure 11.9, along with the sine function we wish to approximate.

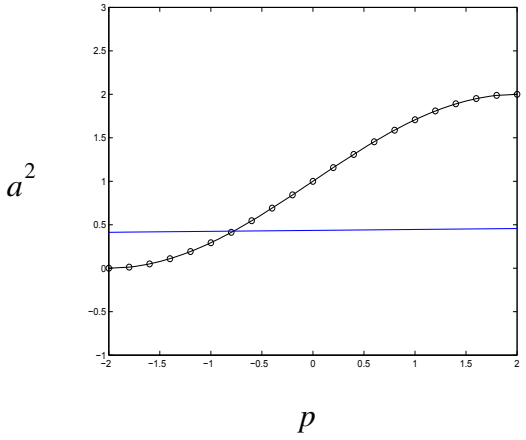


Figure 11.9 Initial Network Response

Next, we need to select a training set $\{p_1, t_1\}, \{p_2, t_2\}, \ldots, \{p_Q, t_Q\}$. In this case, we will sample the function at 21 points in the range [-2,2] at equally spaced intervals of 0.2. The training points are indicated by the circles in Figure 11.9.

Now we are ready to start the algorithm. The training points can be presented in any order, but they are often chosen randomly. For our initial input we will choose $p = 1$, which is the 16th training point:

$$a^0 = p = 1.$$

The output of the first layer is then

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{a}^0 + \mathbf{b}^1) = \mathbf{logsig}\left(\begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix}\begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix}\right) = \mathbf{logsig}\left(\begin{bmatrix} -0.75 \\ -0.54 \end{bmatrix}\right)$$

$$= \begin{bmatrix} \frac{1}{1+e^{0.75}} \\ \frac{1}{1+e^{0.54}} \end{bmatrix} = \begin{bmatrix} 0.321 \\ 0.368 \end{bmatrix}.$$

The second layer output is

$$a^2 = f^2(\mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2) = purelin\left(\begin{bmatrix}0.09 & -0.17\end{bmatrix}\begin{bmatrix}0.321\\0.368\end{bmatrix} + \begin{bmatrix}0.48\end{bmatrix}\right) = \begin{bmatrix}0.446\end{bmatrix}.$$

The error would then be

$$e = t - a = \left\{1 + \sin\left(\frac{\pi}{4}p\right)\right\} - a^2 = \left\{1 + \sin\left(\frac{\pi}{4}1\right)\right\} - 0.446 = 1.261.$$

The next stage of the algorithm is to backpropagate the sensitivities. Before we begin the backpropagation, recall that we will need the derivatives of the transfer functions, $\dot{f}^1(n)$ and $\dot{f}^2(n)$. For the first layer

$$\dot{f}^1(n) = \frac{d}{dn}\left(\frac{1}{1+e^{-n}}\right) = \frac{e^{-n}}{(1+e^{-n})^2} = \left(1 - \frac{1}{1+e^{-n}}\right)\left(\frac{1}{1+e^{-n}}\right) = (1-a^1)(a^1).$$

For the second layer we have

$$\dot{f}^2(n) = \frac{d}{dn}(n) = 1.$$

We can now perform the backpropagation. The starting point is found at the second layer, using Eq. (11.44):

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2\left[\dot{f}^2(n^2)\right](1.261) = -2\begin{bmatrix}1\end{bmatrix}(1.261) = -2.522.$$

The first layer sensitivity is then computed by backpropagating the sensitivity from the second layer, using Eq. (11.45):

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = \begin{bmatrix}(1-a_1^1)(a_1^1) & 0\\ 0 & (1-a_2^1)(a_2^1)\end{bmatrix}\begin{bmatrix}0.09\\-0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$= \begin{bmatrix}(1-0.321)(0.321) & 0\\ 0 & (1-0.368)(0.368)\end{bmatrix}\begin{bmatrix}0.09\\-0.17\end{bmatrix}\begin{bmatrix}-2.522\end{bmatrix}$$

$$= \begin{bmatrix}0.218 & 0\\ 0 & 0.233\end{bmatrix}\begin{bmatrix}-0.227\\0.429\end{bmatrix} = \begin{bmatrix}-0.0495\\0.0997\end{bmatrix}.$$

The final stage of the algorithm is to update the weights. For simplicity, we will use a learning rate $\alpha = 0.1$. (In Chapter 12 the choice of learning rate will be discussed in more detail.) From Eq. (11.46) and Eq. (11.47) we have

$$\mathbf{W}^2(1) = \mathbf{W}^2(0) - \alpha \mathbf{s}^2 (\mathbf{a}^1)^T = \begin{bmatrix} 0.09 & -0.17 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} \begin{bmatrix} 0.321 & 0.368 \end{bmatrix}$$

$$= \begin{bmatrix} 0.171 & -0.0772 \end{bmatrix},$$

$$\mathbf{b}^2(1) = \mathbf{b}^2(0) - \alpha \mathbf{s}^2 = \begin{bmatrix} 0.48 \end{bmatrix} - 0.1 \begin{bmatrix} -2.522 \end{bmatrix} = \begin{bmatrix} 0.732 \end{bmatrix},$$

$$\mathbf{W}^1(1) = \mathbf{W}^1(0) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T = \begin{bmatrix} -0.27 \\ -0.41 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} = \begin{bmatrix} -0.265 \\ -0.420 \end{bmatrix},$$

$$\mathbf{b}^1(1) = \mathbf{b}^1(0) - \alpha \mathbf{s}^1 = \begin{bmatrix} -0.48 \\ -0.13 \end{bmatrix} - 0.1 \begin{bmatrix} -0.0495 \\ 0.0997 \end{bmatrix} = \begin{bmatrix} -0.475 \\ -0.140 \end{bmatrix}.$$

This completes the first iteration of the backpropagation algorithm. We next proceed to randomly choose another input from the training set and perform another iteration of the algorithm. We continue to iterate until the difference between the network response and the target function reaches some acceptable level. (Note that this will generally take many passes through the entire training set.) We will discuss convergence criteria in more detail in Chapter 12.

To experiment with the backpropagation calculation for this two-layer network, use the MATLAB® Neural Network Design Demonstration Backpropagation Calculation (**nnd11bc**).

Batch vs. Incremental Training

Incremental Training

Batch Training

The algorithm described above is the stochastic gradient descent algorithm, which involves "on-line" or *incremental training*, in which the network weights and biases are updated after each input is presented (as with the LMS algorithm of Chapter 10). It is also possible to perform *batch training*, in which the complete gradient is computed (after all inputs are applied to the network) before the weights and biases are updated. For example, if each input occurs with equal probability, the mean square error performance index can be written

$$F(\mathbf{x}) = E[\mathbf{e}^T \mathbf{e}] = E[(\mathbf{t} - \mathbf{a})^T (\mathbf{t} - \mathbf{a})] = \frac{1}{Q} \sum_{q=1}^{Q} (\mathbf{t}_q - \mathbf{a}_q)^T (\mathbf{t}_q - \mathbf{a}_q). \quad (11.49)$$

The total gradient of this performance index is

$$\nabla F(\mathbf{x}) = \nabla\left\{\frac{1}{Q}\sum_{q=1}^{Q}(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\right\} = \frac{1}{Q}\sum_{q=1}^{Q}\nabla\{(\mathbf{t}_q - \mathbf{a}_q)^T(\mathbf{t}_q - \mathbf{a}_q)\}. \quad (11.50)$$

Therefore, the total gradient of the mean square error is the mean of the gradients of the individual squared errors. Therefore, to implement a batch version of the backpropagation algorithm, we would step through Eq. (11.41) through Eq. (11.45) for all of the inputs in the training set. Then, the individual gradients would be averaged to get the total gradient. The update equations for the batch steepest descent algorithm would then be

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m(\mathbf{a}_q^{m-1})^T, \quad (11.51)$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \frac{\alpha}{Q}\sum_{q=1}^{Q}\mathbf{s}_q^m. \quad (11.52)$$

Using Backpropagation

In this section we will present some issues relating to the practical implementation of backpropagation. We will discuss the choice of network architecture, and problems with network convergence and generalization. (We will discuss implementation issues again in Chapter 12, which investigates procedures for improving the algorithm.)

Choice of Network Architecture

As we discussed earlier in this chapter, multilayer networks can be used to approximate almost any function, if we have enough neurons in the hidden layers. However, we cannot say, in general, how many layers or how many neurons are necessary for adequate performance. In this section we want to use a few examples to provide some insight into this problem.

For our first example let's assume that we want to approximate the following functions:

$$g(p) = 1 + \sin\left(\frac{i\pi}{4}p\right) \text{ for } -2 \le p \le 2, \quad (11.53)$$

where i takes on the values 1, 2, 4 and 8. As i is increased, the function becomes more complex, because we will have more periods of the sine wave over the interval $-2 \le p \le 2$. It will be more difficult for a neural network with a fixed number of neurons in the hidden layers to approximate $g(p)$ as i is increased.

For this first example we will use a 1-3-1 network, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear. Recall from our example on page 11-5 that this type of two-layer network can produce a response that is a sum of three log-sigmoid functions (or as many log-sigmoids as there are neurons in the hidden layer). Clearly there is a limit to how complex a function this network can implement. Figure 11.10 illustrates the response of the network after it has been trained to approximate $g(p)$ for $i = 1, 2, 4, 8$. The final network responses are shown by the blue lines.

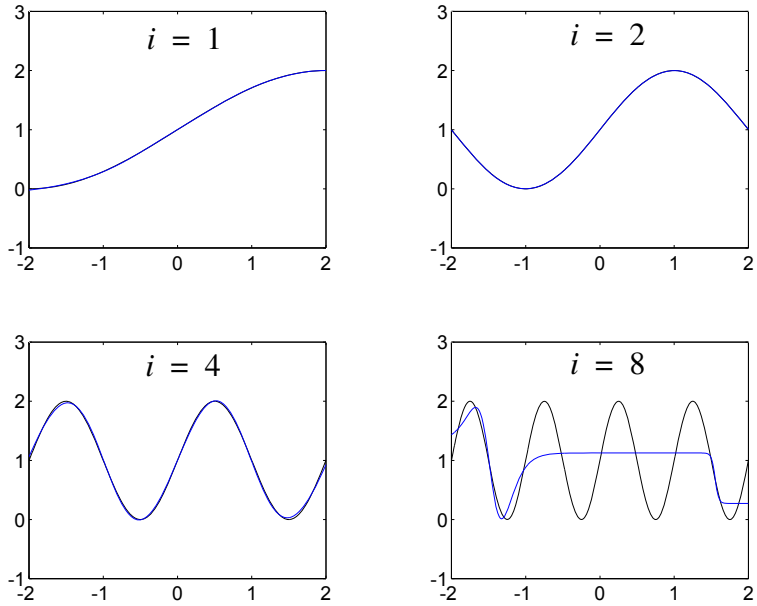


Figure 11.10 Function Approximation Using a 1-3-1 Network

We can see that for $i = 4$ the 1-3-1 network reaches its maximum capability. When $i > 4$ the network is not capable of producing an accurate approximation of $g(p)$. In the bottom right graph of Figure 11.10 we can see how the 1-3-1 network attempts to approximate $g(p)$ for $i = 8$. The mean square error between the network response and $g(p)$ is minimized, but the network response is only able to match a small part of the function.

In the next example we will approach the problem from a slightly different perspective. This time we will pick one function $g(p)$ and then use larger and larger networks until we are able to accurately represent the function. For $g(p)$ we will use

$$g(p) = 1 + \sin\left(\frac{6\pi}{4}p\right) \text{ for } -2 \le p \le 2. \tag{11.54}$$

To approximate this function we will use two-layer networks, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear (1-S^1-1 networks). As we discussed earlier in

this chapter, the response of this network is a superposition of S^1 sigmoid functions.

Figure 11.11 illustrates the network response as the number of neurons in the first layer (hidden layer) is increased. Unless there are at least five neurons in the hidden layer the network cannot accurately represent $g(p)$.

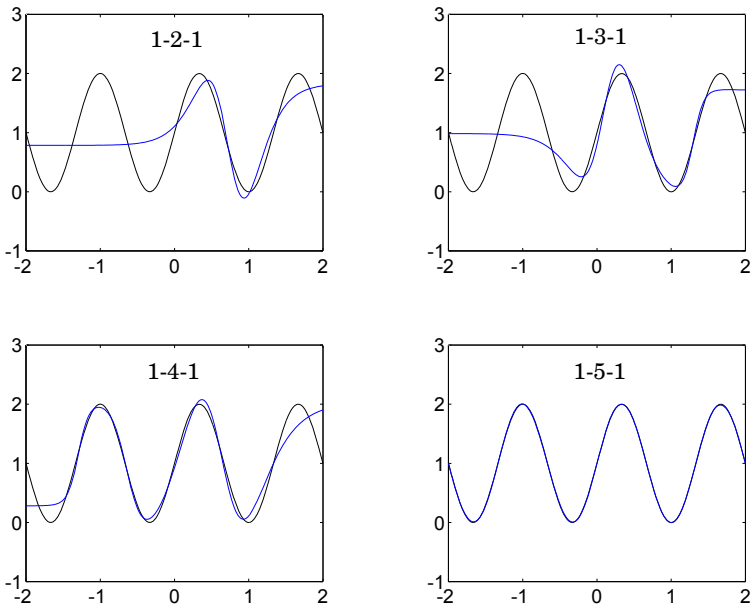
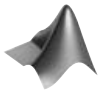


Figure 11.11 Effect of Increasing the Number of Hidden Neurons

To summarize these results, a 1-S^1-1 network, with sigmoid neurons in the hidden layer and linear neurons in the output layer, can produce a response that is a superposition of S^1 sigmoid functions. If we want to approximate a function that has a large number of inflection points, we will need to have a large number of neurons in the hidden layer.



Use the MATLAB® Neural Network Design Demonstration Function Approximation (`nnd11fa`) *to develop more insight into the capability of a two-layer network.*

Convergence

In the previous section we presented some examples in which the network response did not give an accurate approximation to the desired function, even though the backpropagation algorithm produced network parameters that minimized mean square error. This occurred because the capabilities of the network were inherently limited by the number of hidden neurons it contained. In this section we will provide an example in which the network is capable of approximating the function, but the learning algorithm does not produce network parameters that produce an accurate approximation. In the next chapter we will discuss this problem in more detail and explain why it occurs. For now we simply want to illustrate the problem.

The function that we want the network to approximate is

$$g(p) = 1 + \sin(\pi p) \text{ for } -2 \le p \le 2 . \tag{11.55}$$

To approximate this function we will use a 1-3-1 network, where the transfer function for the first layer is log-sigmoid and the transfer function for the second layer is linear.

Figure 11.12 illustrates a case where the learning algorithm converges to a solution that minimizes mean square error. The thin blue lines represent intermediate iterations, and the thick blue line represents the final solution, when the algorithm has converged. (The numbers next to each curve indicate the sequence of iterations, where 0 represents the initial condition and 5 represents the final solution. The numbers do not correspond to the iteration number. There were many iterations for which no curve is represented. The numbers simply indicate an ordering.)

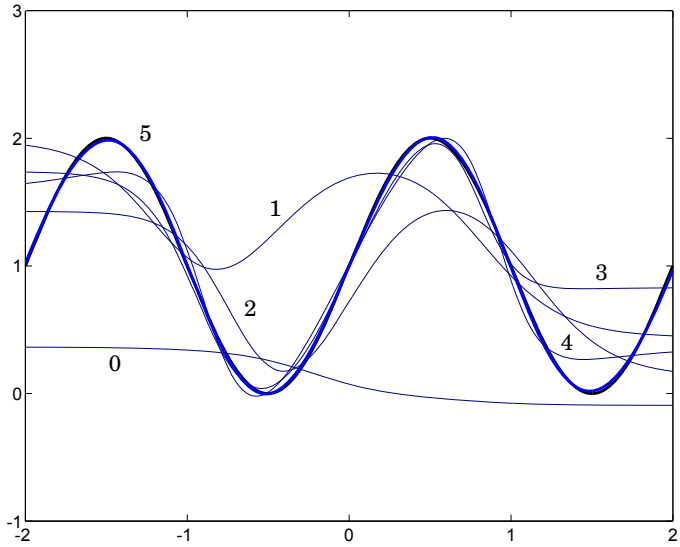


Figure 11.12 Convergence to a Global Minimum

Figure 11.13 illustrates a case where the learning algorithm converges to a solution that does not minimize mean square error. The thick blue line (marked with a 5) represents the network response at the final iteration. The gradient of the mean square error is zero at the final iteration, therefore we have a local minima, but we know that a better solution exists, as evidenced by Figure 11.12. The only difference between this result and the result shown in Figure 11.12 is the initial condition. From one initial condition the algorithm converged to a global minimum point, while from another initial condition the algorithm converged to a local minimum point.

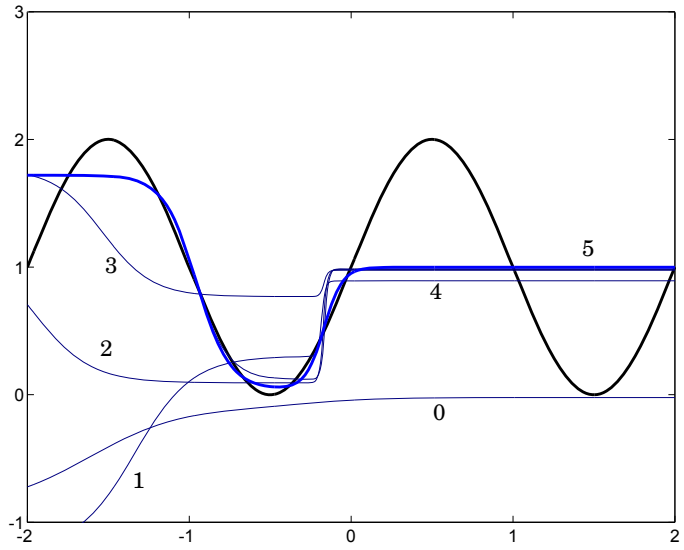


Figure 11.13 Convergence to a Local Minimum

Note that this result could not have occurred with the LMS algorithm. The mean square error performance index for the ADALINE network is a quadratic function with a single minimum point (under most conditions). Therefore the LMS algorithm is guaranteed to converge to the global minimum as long as the learning rate is small enough. The mean square error for the multilayer network is generally much more complex and has many local minima (as we will see in the next chapter). When the backpropagation algorithm converges we cannot be sure that we have an optimum solution. It is best to try several different initial conditions in order to ensure that an optimum solution has been obtained.

Generalization

In most cases the multilayer network is trained with a finite number of examples of proper network behavior:

$$\{\mathbf{p}_1, \mathbf{t}_1\}\ ,\ \{\mathbf{p}_2, \mathbf{t}_2\}\ ,\ \ldots\ ,\ \{\mathbf{p}_Q, \mathbf{t}_Q\}\ . \tag{11.56}$$

This training set is normally representative of a much larger class of possible input/output pairs. It is important that the network successfully *generalize* what it has learned to the total population.

For example, suppose that the training set is obtained by sampling the following function:

$$g(p) = 1 + \sin\left(\frac{\pi}{4}p\right), \tag{11.57}$$

at the points $p = -2, -1.6, -1.2, \ldots, 1.6, 2$. (There are a total of 11 input/target pairs.) In Figure 11.14 we see the response of a 1-2-1 network that has

been trained on this data. The black line represents $g(p)$, the blue line represents the network response, and the '+' symbols indicate the training set.

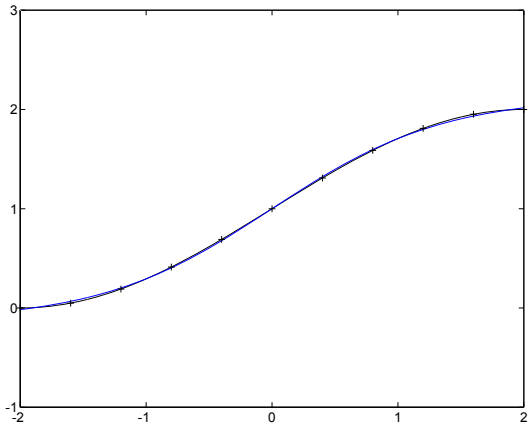


Figure 11.14 1-2-1 Network Approximation of $g(p)$

We can see that the network response is an accurate representation of $g(p)$. If we were to find the response of the network at a value of p that was not contained in the training set (e.g., $p = -0.2$), the network would still produce an output close to $g(p)$. This network generalizes well.

Now consider Figure 11.15, which shows the response of a 1-9-1 network that has been trained on the same data set. Note that the network response accurately models $g(p)$ at all of the training points. However, if we compute the network response at a value of p not contained in the training set (e.g., $p = -0.2$) the network might produce an output far from the true response $g(p)$. This network does not generalize well.

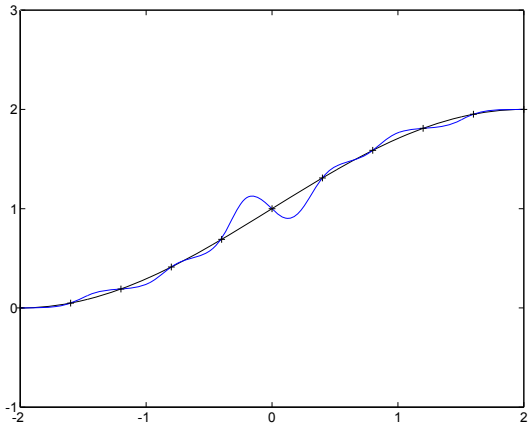


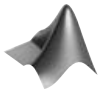
Figure 11.15 1-9-1 Network Approximation of $g(p)$

The 1-9-1 network has too much flexibility for this problem; it has a total of 28 adjustable parameters (18 weights and 10 biases), and yet there are only 11 data points in the training set. The 1-2-1 network has only 7 param-

eters and is therefore much more restricted in the types of functions that it can implement.

For a network to be able to generalize, it should have fewer parameters than there are data points in the training set. In neural networks, as in all modeling problems, we want to use the simplest network that can adequately represent the training set. Don't use a bigger network when a smaller network will work (a concept often referred to as Ockham's Razor).

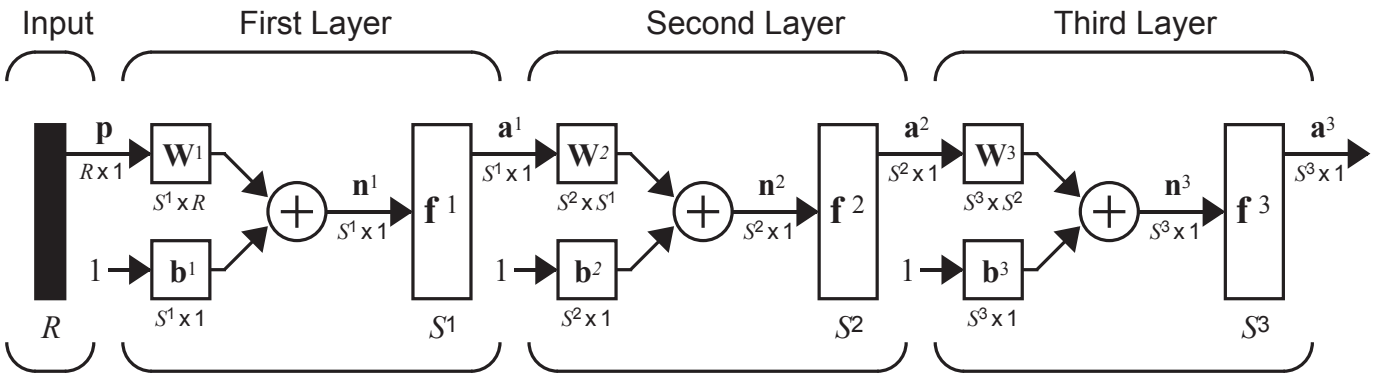
An alternative to using the simplest network is to stop the training before the network overfits. A reference to this procedure and other techniques to improve generalization are given in Chapter 13.



To experiment with generalization in neural networks, use the MATLAB® Neural Network Design Demonstration Generalization (nnd11gn).

Summary of Results

Multilayer Network



$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{a}^2+\mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{W}^3\mathbf{f}^2(\mathbf{W}^2\mathbf{f}^1(\mathbf{W}^1\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Backpropagation Algorithm

Performance Index

$$F(\mathbf{x}) = E[\mathbf{e}^T\mathbf{e}] = E[(\mathbf{t}-\mathbf{a})^T(\mathbf{t}-\mathbf{a})]$$

Approximate Performance Index

$$\hat{F}(\mathbf{x}) = \mathbf{e}^T(k)\mathbf{e}(k) = (\mathbf{t}(k)-\mathbf{a}(k))^T(\mathbf{t}(k)-\mathbf{a}(k))$$

Sensitivity

$$\mathbf{s}^m \equiv \frac{\partial \hat{F}}{\partial \mathbf{n}^m} = \begin{bmatrix} \frac{\partial \hat{F}}{\partial n_1^m} \\ \frac{\partial \hat{F}}{\partial n_2^m} \\ \vdots \\ \frac{\partial \hat{F}}{\partial n_{S^m}^m} \end{bmatrix}$$

Forward Propagation

$$\mathbf{a}^0 = \mathbf{p},$$

$$\mathbf{a}^{m+1} = \mathbf{f}^{m+1}(\mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1}) \text{ for } m = 0, 1, \ldots, M-1,$$

$$\mathbf{a} = \mathbf{a}^M.$$

Backward Propagation

$$\mathbf{s}^M = -2\dot{\mathbf{F}}^M(\mathbf{n}^M)(\mathbf{t} - \mathbf{a}),$$

$$\mathbf{s}^m = \dot{\mathbf{F}}^m(\mathbf{n}^m)(\mathbf{W}^{m+1})^T\mathbf{s}^{m+1}, \text{ for } m = M-1, \ldots, 2, 1,$$

where

$$\dot{\mathbf{F}}^m(\mathbf{n}^m) = \begin{bmatrix} \dot{f}^m(n_1^m) & 0 & \ldots & 0 \\ 0 & \dot{f}^m(n_2^m) & \ldots & 0 \\ \vdots & \vdots & & \vdots \\ 0 & 0 & & \dot{f}^m(n_{S^m}^m) \end{bmatrix},$$

$$\dot{f}^m(n_j^m) = \frac{\partial f^m(n_j^m)}{\partial n_j^m}.$$

Weight Update (Approximate Steepest Descent)

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha\mathbf{s}^m(\mathbf{a}^{m-1})^T,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha\mathbf{s}^m.$$

Solved Problems

P11.1 Consider the two classes of patterns that are shown in Figure P11.1. Class I represents vertical lines and Class II represents horizontal lines.

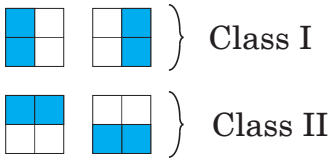


Figure P11.1 Pattern Classes for Problem P11.1

i. Are these categories linearly separable?

ii. Design a multilayer network to distinguish these categories.

i. Let's begin by converting the patterns to vectors by scanning each 2X2 grid one column at a time. Each white square will be represented by a "-1" and each blue square by a "1". The vertical lines (Class I patterns) then become

$$\mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} \text{ and } \mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix},$$

and the horizontal lines (Class II patterns) become

$$\mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \\ 1 \\ -1 \end{bmatrix} \text{ and } \mathbf{p}_4 = \begin{bmatrix} -1 \\ 1 \\ -1 \\ 1 \end{bmatrix}.$$

In order for these categories to be linearly separable we must be able to place a hyperplane between the two categories. This means there must be a weight matrix $\mathbf{W}$ and a bias b such that

$$\mathbf{W}\mathbf{p}_1 + b > 0,\ \mathbf{W}\mathbf{p}_2 + b > 0,\ \mathbf{W}\mathbf{p}_3 + b < 0,\ \mathbf{W}\mathbf{p}_4 + b < 0.$$

These conditions can be converted to

$$\begin{bmatrix} w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix} = \begin{bmatrix} w_{1,1} + w_{1,2} - w_{1,3} - w_{1,4} \end{bmatrix} > 0 ,$$

$$\begin{bmatrix} -w_{1,1} - w_{1,2} + w_{1,3} + w_{1,4} \end{bmatrix} > 0 ,$$

$$\begin{bmatrix} w_{1,1} - w_{1,2} + w_{1,3} - w_{1,4} \end{bmatrix} < 0 ,$$

$$\begin{bmatrix} -w_{1,1} + w_{1,2} - w_{1,3} + w_{1,4} \end{bmatrix} < 0 .$$

The first two conditions reduce to

$$w_{1,1} + w_{1,2} > w_{1,3} + w_{1,4} \text{ and } w_{1,3} + w_{1,4} > w_{1,1} + w_{1,2} ,$$

which are contradictory. The final two conditions reduce to

$$w_{1,1} + w_{1,3} > w_{1,2} + w_{1,4} \text{ and } w_{1,2} + w_{1,4} > w_{1,1} + w_{1,3} ,$$

which are also contradictory. Therefore there is no hyperplane that can separate these two categories.

ii. There are many different multilayer networks that could solve this problem. We will design a network by first noting that for the Class I vectors either the first two elements or the last two elements will be “1”. The Class II vectors have alternating “1” and “-1” patterns. This leads to the network shown in Figure P11.2.

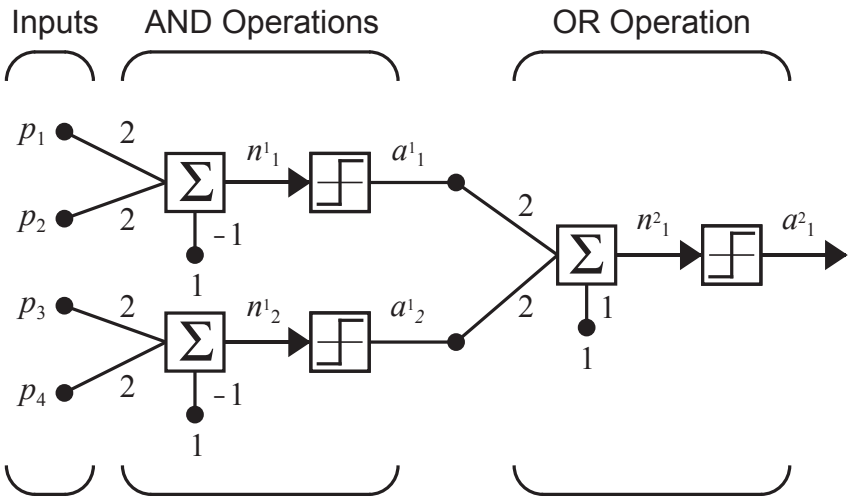


Figure P11.2 Network to Categorize Horizontal and Vertical Lines

The first neuron in the first layer tests the first two elements of the input vector. If they are both "1" it outputs a "1", otherwise it outputs a "-1". The second neuron in the first layer tests the last two elements of the input vector in the same way. Both of the neurons in the first layer perform AND operations. The second layer of the network tests whether either of the outputs of the first layer are "1". It performs an OR operation. In this way, the network will output a "1" if either the first two elements or the last two elements of the input vector are both "1".

P11.2 Figure P11.3 illustrates a classification problem, where Class I vectors are represented by light circles, and Class II vectors are represented by dark circles. These categories are not linearly separable. Design a multilayer network to correctly classify these categories.

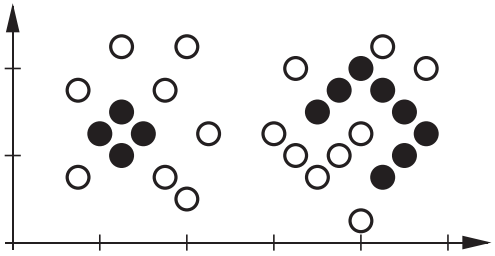


Figure P11.3 Classification Problem

We will solve this problem with a procedure that can be used for arbitrary classification problems. It requires a three-layer network, with hard-limiting neurons in each layer. In the first layer we create a set of linear decision boundaries that separate every Class I vector from every Class II vector. For this problem we used 11 such boundaries. They are shown in Figure P11.4.

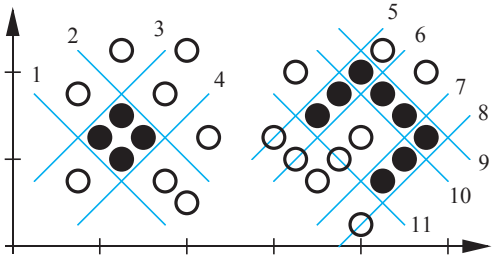


Figure P11.4 First Layer Decision Boundaries

Each row of the weight matrix in the first layer corresponds to one decision boundary. The weight matrix and bias vector for the first layer are

$$(\mathbf{W}^1)^T = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 & 1 & 1 \end{bmatrix},$$

$$(\mathbf{b}^1)^T = \begin{bmatrix} -2 & 3 & 0.5 & 0.5 & -1.75 & 2.25 & -3.25 & 3.75 & 6.25 & -5.75 & -4.75 \end{bmatrix}.$$

(Review Chapters 3, 4 and 10 for procedures for calculating the appropriate weight matrix and bias for a given decision boundary.) Now we can combine the outputs of the 11 first layer neurons into groups with a second layer of AND neurons, such as those we used in the first layer of the network in Problem P11.1. The second layer weight matrix and bias are

$$\mathbf{W}^2 = \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix}, \quad \mathbf{b}^T = \begin{bmatrix} -3 \\ -3 \\ -3 \\ -3 \end{bmatrix}.$$

The four decision boundaries for the second layer are shown in Figure P11.5. For example, the neuron 2 decision boundary is obtained by combining the boundaries 5, 6, 9 and 11 from layer 1. This can be seen by looking at row 2 of $\mathbf{W}^2$.

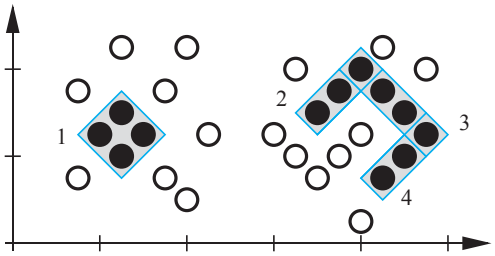


Figure P11.5 Second Layer Decision Regions

In the third layer of the network we will combine together the four decision regions of the second layer into one decision region using an OR operation, just as in the last layer of the network in Problem P11.1. The weight matrix and bias for the third layer are

$$\mathbf{W}^3 = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{b}^3 = \begin{bmatrix} 3 \end{bmatrix}.$$

The complete network is shown in Figure P11.6.

The procedure that we used to develop this network can be used to solve classification problems with arbitrary decision boundaries as long as we have enough neurons in the hidden layers. The idea is to use the first layer to create a number of linear boundaries, which can be combined by using AND neurons in the second layer and OR neurons in the third layer. The decision regions of the second layer are convex, but the final decision boundaries created by the third layer can have arbitrary shapes.

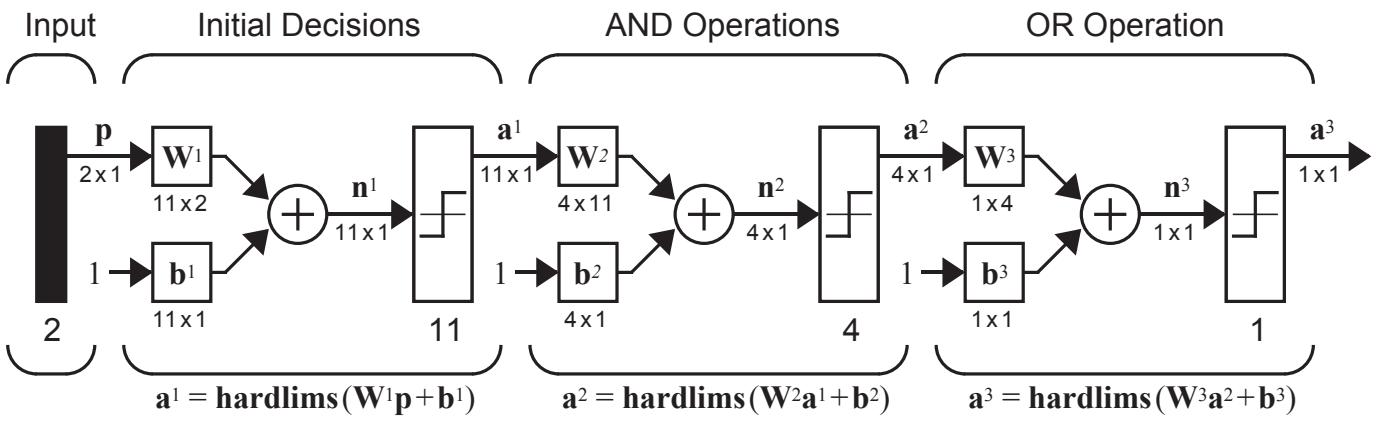


Figure P11.6 Network for Problem P11.2

The final network decision regions are given in Figure P11.7. Any vector in the shaded areas will produce a network output of 1, which corresponds to Class II. All other vectors will produce a network output of -1, which corresponds to Class I.

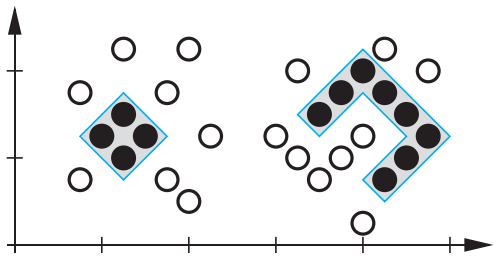


Figure P11.7 Final Decision Regions

P11.3 Show that a multilayer network with linear transfer functions is equivalent to a single-layer linear network.

For a multilayer linear network the forward equations would be

$$\mathbf{a}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1$$

$$\mathbf{a}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{b}^2 = \mathbf{W}^2\mathbf{W}^1\mathbf{p} + [\mathbf{W}^2\mathbf{b}^1 + \mathbf{b}^2],$$

$$\mathbf{a}^3 = \mathbf{W}^3\mathbf{a}^2 + \mathbf{b}^3 = \mathbf{W}^3\mathbf{W}^2\mathbf{W}^1\mathbf{p} + [\mathbf{W}^3\mathbf{W}^2\mathbf{b}^1 + \mathbf{W}^3\mathbf{b}^2 + \mathbf{b}^3].$$

If we continue this process we can see that for an M-layer linear network, the equivalent single-layer linear network would have the following weight matrix and bias vector

$$\mathbf{W} = \mathbf{W}^M\mathbf{W}^{M-1}\ldots\mathbf{W}^2\mathbf{W}^1,$$

$$\mathbf{b} = [\mathbf{W}^M\mathbf{W}^{M-1}\ldots\ \mathbf{W}^2]\mathbf{b}^1 + [\mathbf{W}^M\mathbf{W}^{M-1}\ldots\ \mathbf{W}^3]\mathbf{b}^2 + \cdots + \mathbf{b}^M .$$

P11.4 The purpose of this problem is to illustrate the use of the chain rule. Consider the following dynamic system:

$$y(k+1) = f(y(k)) .$$

We want to choose the initial condition $y(0)$ so that at some final time $k = K$ the system output $y(K)$ will be as close as possible to some target output t. We will minimize the performance index

$$F(y(0)) = (t - y(K))^2$$

using steepest descent, so we need the gradient

$$\frac{\partial}{\partial y(0)} F(y(0)) .$$

Find a procedure for computing this using the chain rule.

The gradient is

$$\frac{\partial}{\partial y(0)} F(y(0)) = \frac{\partial (t - y(K))^2}{\partial y(0)} = 2(t - y(K))\left[-\frac{\partial}{\partial y(0)} y(K)\right] .$$

The key term is

$$\left[\frac{\partial}{\partial y(0)} y(K)\right],$$

which cannot be computed directly, since $y(K)$ is not an explicit function of $y(0)$. Let's define an intermediate term

$$r(k) \equiv \frac{\partial}{\partial y(0)} y(k) .$$

Then we can use the chain rule:

$$r(k+1) = \frac{\partial}{\partial y(0)} y(k+1) = \frac{\partial y(k+1)}{\partial y(k)} \times \frac{\partial y(k)}{\partial y(0)} = \frac{\partial y(k+1)}{\partial y(k)} \times r(k) .$$

From the system dynamics we know

$$\frac{\partial y(k+1)}{\partial y(k)} = \frac{\partial f(y(k))}{\partial y(k)} = \dot{f}(y(k)) .$$

Therefore the recursive equation for the computation of $r(k)$ is

$$r(k+1) = \dot{f}(y(k))r(k).$$

This is initialized at $k = 0$:

$$r(0) = \frac{\partial y(0)}{\partial y(0)} = 1.$$

The total procedure for computing the gradient is then

$$r(0) = 1,$$

$$r(k+1) = \dot{f}(y(k))r(k), \text{ for } k = 0, 1, \ldots, K-1,$$

$$\frac{\partial}{\partial y(0)}F(y(0)) = 2(t - y(K))[-r(K)].$$

P11.5 Consider the two-layer network shown in Figure P11.8. The initial weights and biases are set to

$$w^1 = 1,\ b^1 = 1,\ w^2 = -2,\ b^2 = 1.$$

An input/target pair is given to be

$$((p = 1),(t = 1)).$$

i. Find the squared error $(e)^2$ as an explicit function of all weights and biases.

ii. Using part (i) find $\partial(e)^2/\partial w^1$ at the initial weights and biases.

iii. Repeat part (ii) using backpropagation and compare results.

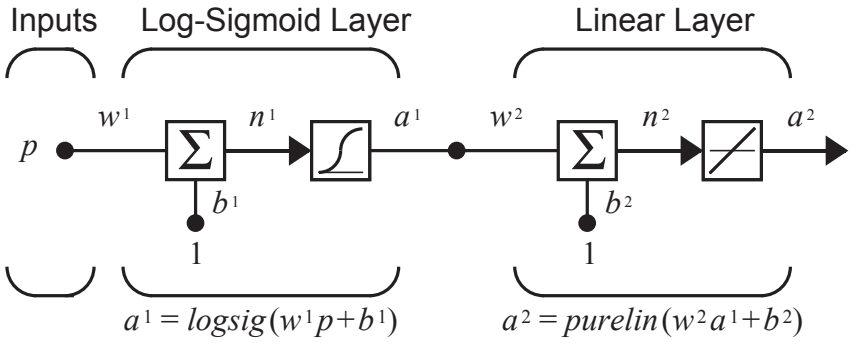


Figure P11.8 Two-Layer Network for Problem P11.5

i. The squared error is given by

$$(e)^2 = (t-a^2)^2 = \left(t-\left\{w^2\frac{1}{(1+\exp(-(w^1p+b^1)))}+b^2\right\}\right)^2.$$

ii. The derivative is

$$\frac{\partial(e)^2}{\partial w^1} = 2e\frac{\partial e}{\partial w^1} = 2e\left\{w^2\frac{1}{(1+\exp(-(w^1p+b^1)))^2}\exp(-(w^1p+b^1))(-p)\right\}$$

To evaluate this at the initial weights and biases we find

$$a^1 = \frac{1}{(1+\exp(-(w^1p+b^1)))} = \frac{1}{(1+\exp(-(1(1)+1)))} = 0.8808$$

$$a^2 = w^2a^1+b^2 = (-2)0.8808+1 = -0.7616$$

$$e = (t-a^2) = (1-(-0.7616)) = 1.7616$$

$$\frac{\partial(e)^2}{\partial w^1} = 2e\left\{w^2\frac{1}{(1+\exp(-(w^1p+b^1)))^2}\exp(-(w^1p+b^1))(-p)\right\}$$

$$= 2(1.7616)\left\{(-2)\frac{1}{(1+\exp(-(1(1)+1)))^2}\exp(-(1(1)+1))(-1)\right\}$$

$$= 3.5232\left(0.2707\frac{1}{(1.289)^2}\right) = 0.7398\ .$$

iii. To backpropagate the sensitivities we use Eq. (11.44) and Eq. (11.45):

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t}-\mathbf{a}) = -2(1)(1-(-0.7616)) = -3.5232\,,$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T\mathbf{s}^2 = [a^1(1-a^1)](-2)\mathbf{s}^2$$

$$= [0.8808(1-0.8808)](-2)(-3.5232) = 0.7398\ .$$

From Eq. (11.23) we can compute $\partial(e)^2/\partial w^1$:

$$\frac{\partial(e)^2}{\partial w^1} = s^1a^0 = s^1p = (0.7398)(1) = 0.7398\,.$$

This agrees with our result from part (ii).

P11.6 Earlier in this chapter we showed that if the neuron transfer function is log-sigmoid,

$$a = f(n) = \frac{1}{1+e^{-n}},$$

then the derivative can be conveniently computed by

$$\dot{f}(n) = a(1-a).$$

Find a convenient way to compute the derivative for the hyperbolic tangent sigmoid:

$$a = f(n) = tansig(n) = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}.$$

Computing the derivative directly we find

$$\dot{f}(n) = \frac{df(n)}{dn} = \frac{d}{dn}\left(\frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}\right) = -\frac{e^{n}-e^{-n}}{(e^{n}+e^{-n})^{2}}(e^{n}-e^{-n}) + \frac{e^{n}+e^{-n}}{e^{n}+e^{-n}}$$

$$= 1 - \frac{(e^{n}-e^{-n})^{2}}{(e^{n}+e^{-n})^{2}} = 1-(a)^{2}.$$

P11.7 For the network shown in Figure P11.9 the initial weights and biases are chosen to be

$$w^{1}(0) = -1,\ b^{1}(0) = 1,\ w^{2}(0) = -2,\ b^{2}(0) = 1.$$

An input/target pair is given to be

$$((p=-1),(t=1)).$$

Perform one iteration of backpropagation with $\alpha = 1$.

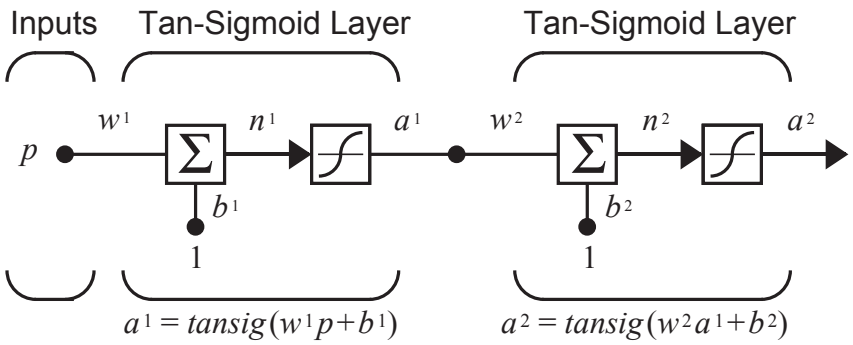


Figure P11.9 Two-Layer Tan-Sigmoid Network

The first step is to propagate the input through the network.

$$n^1 = w^1 p + b^1 = (-1)(-1) + 1 = 2$$

$$a^1 = tansig(n^1) = \frac{\exp(n^1) - \exp(-n^1)}{\exp(n^1) + \exp(-n^1)} = \frac{\exp(2) - \exp(-2)}{\exp(2) + \exp(-2)} = 0.964$$

$$n^2 = w^2 a^1 + b^2 = (-2)(0.964) + 1 = -0.928$$

$$a^2 = tansig(n^2) = \frac{\exp(n^2) - \exp(-n^2)}{\exp(n^2) + \exp(-n^2)} = \frac{\exp(-0.928) - \exp(0.928)}{\exp(-0.928) + \exp(0.928)}$$

$$= -0.7297$$

$$e = (t - a^2) = (1 - (-0.7297)) = 1.7297$$

Now we backpropagate the sensitivities using Eq. (11.44) and Eq. (11.45).

$$\mathbf{s}^2 = -2\dot{\mathbf{F}}^2(\mathbf{n}^2)(\mathbf{t} - \mathbf{a}) = -2[1 - (a^2)^2](e) = -2[1 - (-0.7297)^2]1.7297$$

$$= -1.6175$$

$$\mathbf{s}^1 = \dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{W}^2)^T \mathbf{s}^2 = [1 - (a^1)^2]w^2 \mathbf{s}^2 = [1 - (0.964)^2](-2)(-1.6175)$$

$$= 0.2285$$

Finally, the weights and biases are updated using Eq. (11.46) and Eq. (11.47):

$$w^2(1) = w^2(0) - \alpha s^2 (a^1)^T = (-2) - 1(-1.6175)(0.964) = -0.4407,$$

$$w^1(1) = w^1(0) - \alpha s^1 (a^0)^T = (-1) - 1(0.2285)(-1) = -0.7715,$$

$$b^2(1) = b^2(0) - \alpha s^2 = 1 - 1(-1.6175) = 2.6175,$$

$$b^1(1) = b^1(0) - \alpha s^1 = 1 - 1(0.2285) = 0.7715.$$

P11.8 In Figure P11.10 we have a network that is a slight modification to the standard two-layer feedforward network. It has a connection from the input directly to the second layer. Derive the backpropagation algorithm for this network.

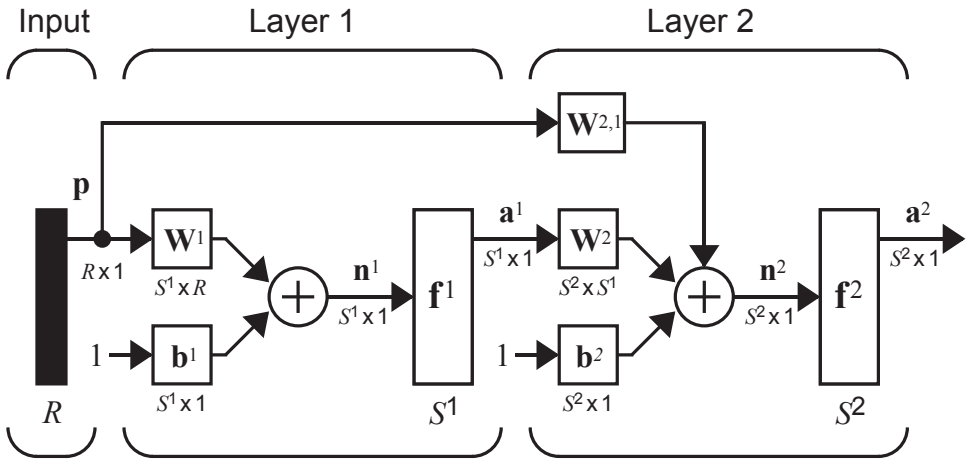


Figure P11.10 Network with Bypass Connection

We begin with the forward equations:

$$\mathbf{n}^1 = \mathbf{W}^1\mathbf{p} + \mathbf{b}^1,$$

$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{n}^1) = \mathbf{f}^1(\mathbf{W}^1\mathbf{p} + \mathbf{b}^1),$$

$$\mathbf{n}^2 = \mathbf{W}^2\mathbf{a}^1 + \mathbf{W}^{2,1}\mathbf{p} + \mathbf{b}^2,$$

$$\mathbf{a}^2 = \mathbf{f}^2(\mathbf{n}^2) = \mathbf{f}^2(\mathbf{W}^2\mathbf{a}^1 + \mathbf{W}^{2,1}\mathbf{p} + \mathbf{b}^2).$$

The backpropagation equations for the sensitivities will not change from those for a standard two-layer network. The sensitivities are the derivatives of the squared error with respect to the net inputs; these derivatives don't change, since we are simply adding a term to the net input.

Next we need the elements of the gradient for the weight update equations. For the standard weights and biases we have

$$\frac{\partial \hat{F}}{\partial w_{i,j}^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial w_{i,j}^m} = s_i^m a_j^{m-1},$$

$$\frac{\partial \hat{F}}{\partial b_i^m} = \frac{\partial \hat{F}}{\partial n_i^m} \times \frac{\partial n_i^m}{\partial b_i^m} = s_i^m .$$

Therefore the update equations for $\mathbf{W}^1$, $\mathbf{b}^1$, $\mathbf{W}^2$ and $\mathbf{b}^2$ do not change. We do need an additional equation for $\mathbf{W}^{2,1}$:

$$\frac{\partial \hat{F}}{\partial w_{i,j}^{2,1}} = \frac{\partial \hat{F}}{\partial n_i^2} \times \frac{\partial n_i^2}{\partial w_{i,j}^{2,1}} = s_i^2 \times \frac{\partial n_i^2}{\partial w_{i,j}^{2,1}}.$$

To find the derivative on the right-hand side of this equation note that

$$n_i^2 = \sum_{j=1}^{S^1} w_{i,j}^2 a_j^1 + \sum_{j=1}^{R} w_{i,j}^{2,1} p_j + b_i^2 .$$

Therefore

$$\frac{\partial n_i^2}{\partial w_{i,j}^{2,1}} = p_j \text{ and } \frac{\partial \hat{F}}{\partial w_{i,j}^{2,1}} = s_i^2 p_j .$$

The update equations can thus be written in matrix form as:

$$\mathbf{W}^m(k+1) = \mathbf{W}^m(k) - \alpha \mathbf{s}^m (\mathbf{a}^{m-1})^T, \; m = 1, 2,$$

$$\mathbf{b}^m(k+1) = \mathbf{b}^m(k) - \alpha \mathbf{s}^m, \; m = 1, 2 .$$

$$\mathbf{W}^{2,1}(k+1) = \mathbf{W}^{2,1}(k) - \alpha \mathbf{s}^2 (\mathbf{a}^0)^T = \mathbf{W}^{2,1}(k) - \alpha \mathbf{s}^2 (\mathbf{p})^T .$$

The main point of this problem is that the backpropagation concept can be used on networks more general than the standard multilayer feedforward network.

P11.9 Find an algorithm, based on the backpropagation concept, that can be used to update the weights w_1 and w_2 in the recurrent network shown in Figure P11.11.

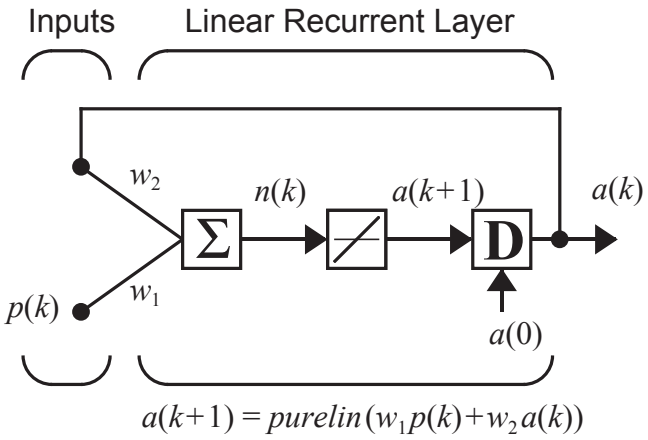


Figure P11.11 Linear Recurrent Network

The first step is to define our performance index. As with the multilayer networks, we will use squared error:

$$\hat{F}(\mathbf{x}) = (t(k) - a(k))^2 = (e(k))^2 .$$

For our weight updates we will use the steepest descent algorithm:

$$\Delta w_i = -\alpha \frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) .$$

These derivatives can be computed as follows:

$$\frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) = \frac{\partial}{\partial w_i} (t(k) - a(k))^2 = 2(t(k) - a(k)) \left\{ -\frac{\partial a(k)}{\partial w_i} \right\} .$$

Therefore, the key terms we need to compute are

$$\frac{\partial a(k)}{\partial w_i} .$$

To compute these terms we first need to write out the network equation:

$$a(k+1) = purelin(w_1 p(k) + w_2 a(k)) = w_1 p(k) + w_2 a(k) .$$

Next we take the derivative of both sides of this equation with respect to the network weights:

$$\frac{\partial a(k+1)}{\partial w_1} = p(k) + w_2 \frac{\partial a(k)}{\partial w_1} ,$$

$$\frac{\partial a(k+1)}{\partial w_2} = a(k) + w_2 \frac{\partial a(k)}{\partial w_2} .$$

(Note that we had to take account of the fact that $a(k)$ is itself a function of w_1 and w_2.) These two recursive equations are then used to compute the derivatives needed for the steepest descent weight update. The equations are initialized with

$$\frac{\partial a(0)}{\partial w_1} = 0, \frac{\partial a(0)}{\partial w_2} = 0,$$

since the initial condition is not a function of the weight.

To illustrate the process, let's say that $a(0) = 0$. The first network update would be

$$a(1) = w_1 p(0) + w_2 a(0) = w_1 p(0).$$

The first derivatives would be computed:

$$\frac{\partial a(1)}{\partial w_1} = p(0) + w_2 \frac{\partial a(0)}{\partial w_1} = p(0), \frac{\partial a(1)}{\partial w_2} = a(0) + w_2 \frac{\partial a(0)}{\partial w_2} = 0.$$

The first weight updates would be

$$\Delta w_i = -\alpha \frac{\partial}{\partial w_i} \hat{F}(\mathbf{x}) = -\alpha \left[2(t(1) - a(1)) \left\{ -\frac{\partial a(1)}{\partial w_i} \right\} \right]$$

$$\Delta w_1 = -2\alpha(t(1) - a(1))\{-p(0)\}$$

$$\Delta w_2 = -2\alpha(t(1) - a(1))\{0\} = 0.$$

This algorithm is a type of *dynamic backpropagation*, in which the gradient is computed by means of a difference equation.

P11.10 Show that backpropagation reduces to the LMS algorithm for a single-layer linear network (ADALINE).

The sensitivity calculation for a single-layer linear network would be:

$$\mathbf{s}^1 = -2\dot{\mathbf{F}}^1(\mathbf{n}^1)(\mathbf{t} - \mathbf{a}) = -2\mathbf{I}(\mathbf{t} - \mathbf{a}) = -2\mathbf{e},$$

The weight update (Eq. (11.46) and Eq. (11.47)) would be

$$\mathbf{W}^1(k+1) = \mathbf{W}^1(k) - \alpha \mathbf{s}^1 (\mathbf{a}^0)^T = \mathbf{W}^1(k) - \alpha(-2\mathbf{e})\mathbf{p}^T = \mathbf{W}^1(k) + 2\alpha\mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^1(k+1) = \mathbf{b}^1(k) - \alpha \mathbf{s}^1 = \mathbf{b}^1(k) - \alpha(-2\mathbf{e}) = \mathbf{b}^1(k) + 2\alpha\mathbf{e}.$$

This is identical to the LMS algorithm of Chapter 10.

Epilogue

In this chapter we have presented the multilayer perceptron network and the backpropagation learning rule. The multilayer network is a powerful extension of the single-layer perceptron network. Whereas the single-layer network is only able to classify linearly separable patterns, the multilayer network can be used for arbitrary classification problems. In addition, multilayer networks can be used as universal function approximators. It has been shown that a two-layer network, with sigmoid-type transfer functions in the hidden layer, can approximate any practical function, given enough neurons in the hidden layer.

The backpropagation algorithm is an extension of the LMS algorithm that can be used to train multilayer networks. Both LMS and backpropagation are approximate steepest descent algorithms that minimize squared error. The only difference between them is in the way in which the gradient is calculated. The backpropagation algorithm uses the chain rule in order to compute the derivatives of the squared error with respect to the weights and biases in the hidden layers. It is called backpropagation because the derivatives are computed first at the last layer of the network, and then propagated backward through the network, using the chain rule, to compute the derivatives in the hidden layers.

One of the major problems with backpropagation has been the long training times. It is not feasible to use the basic backpropagation algorithm on practical problems, because it can take weeks to train a network, even on a large computer. Since backpropagation was first popularized, there has been considerable work on methods to accelerate the convergence of the algorithm. In Chapter 12 we will discuss the reasons for the slow convergence of backpropagation and will present several techniques for improving the performance of the algorithm.

Another key problem in training multilayer networks is overfitting. The network may memorize the data in the training set, but fail to generalize to new situations. In Chapter 13 we will describe in detail training procedures that can be used to produce networks with excellent generalization.

This chapter has focused mainly on the theoretical development of the backpropagation learning rule for training multilayer networks. Practical aspects of training networks with this method are discussed in Chapter 22. Real-world case studies that demonstrate how to train and validate multilayer networks are provided in Chapter 23 (function approximation), Chapter 24 (probability estimation) and Chapter 25 (pattern recognition).

Further Reading

[HoSt89] K. M. Hornik, M. Stinchcombe and H. White, "Multilayer feedforward networks are universal approximators," *Neural Networks*, vol. 2, no. 5, pp. 359–366, 1989.

This paper proves that multilayer feedforward networks with arbitrary squashing functions can approximate any Borel integrable function from one finite dimensional space to another finite dimensional space.

[LeCu85] Y. Le Cun, "Une procedure d'apprentissage pour reseau a seuil assymetrique," *Cognitiva*, vol. 85, pp. 599–604, 1985.

Yann Le Cun discovered the backpropagation algorithm at about the same time as Parker and Rumelhart, Hinton and Williams. This paper describes his algorithm.

[Park85] D. B. Parker, "Learning-logic: Casting the cortex of the human brain in silicon," Technical Report TR-47, Center for Computational Research in Economics and Management Science, MIT, Cambridge, MA, 1985.

David Parker independently derived the backpropagation algorithm at about the same time as Le Cun and Rumelhart, Hinton and Williams. This report describes his algorithm.

[RuHi86] D. E. Rumelhart, G. E. Hinton and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

This paper contains the most widely publicized description of the backpropagation algorithm.

[RuMc86] D. E. Rumelhart and J. L. McClelland, eds., *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, vol. 1, Cambridge, MA: MIT Press, 1986.

This book was one of the two key influences in the resurgence of interest in the neural networks field during the 1980s. Among other topics, it presents the backpropagation algorithm for training multilayer neural networks.

[Werbo74] P. J. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. Thesis, Harvard University, Cambridge, MA, 1974.

This Ph.D. thesis contains what appears to be the first description of the backpropagation algorithm (although that

name is not used). The algorithm is described here in the context of general networks, with neural networks as a special case. Backpropagation did not become widely known until it was rediscovered in the mid 1980s by Rumelhart, Hinton and Williams [RuHi86], David Parker [Park85] and Yann Le Cun [LeCu85].

Exercises

E11.1 Design multilayer networks to perform the classifications illustrated in Figure E11.1. The network should output a 1 whenever the input vector is in the shaded region (or on the boundary) and a -1 otherwise. Draw the network diagram in abbreviated notation and show the weight matrices and bias vectors.

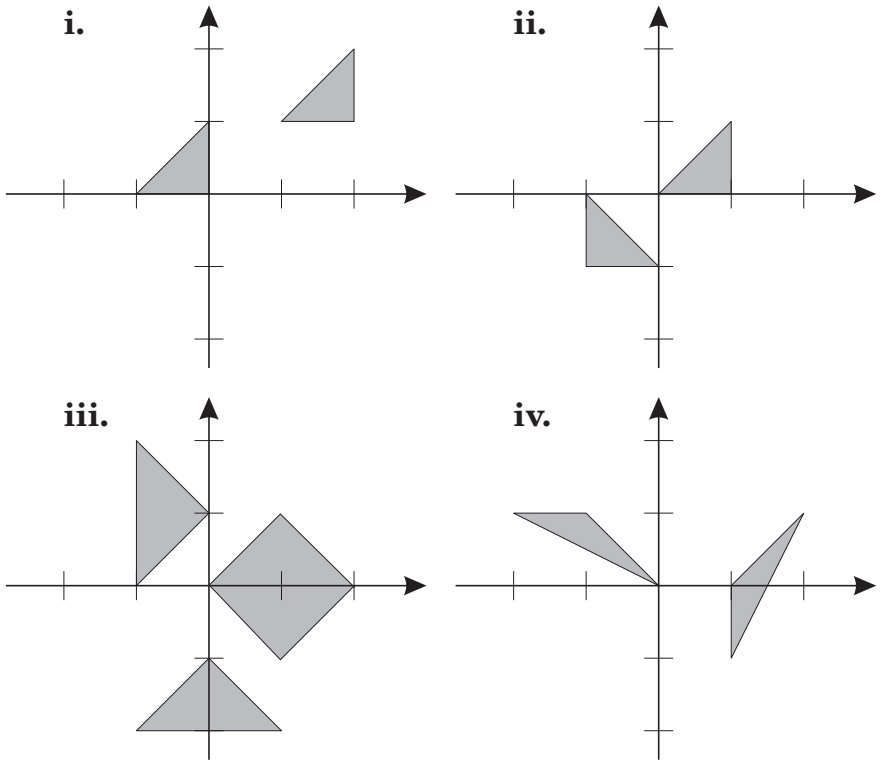


Figure E11.1 Pattern Classification Tasks

E11.2 Choose the weights and biases for the 1-2-1 network shown in Figure 11.4 so that the network response passes through the points indicated by the blue circles in Figure E11.2.

Use the MATLAB® Neural Network Design Demonstration Two-Layer Network Function (`nnd11nf`) *to check your result.*

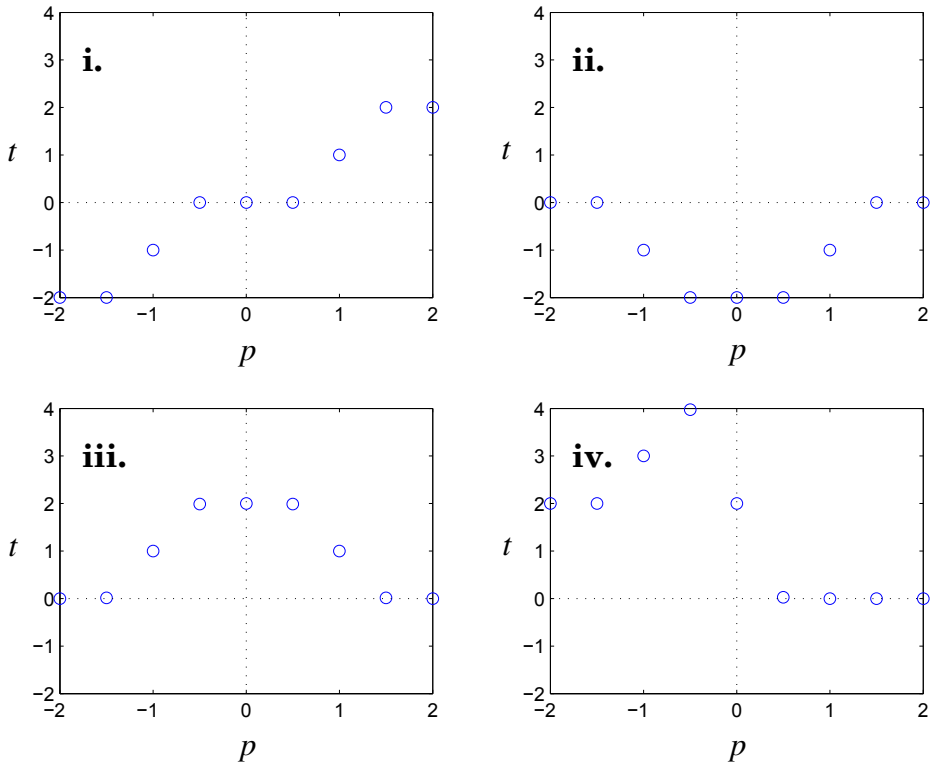


Figure E11.2 Function Approximation Tasks

E11.3 Find a single-layer network that has the same input/output characteristic as the network in Figure E11.3.

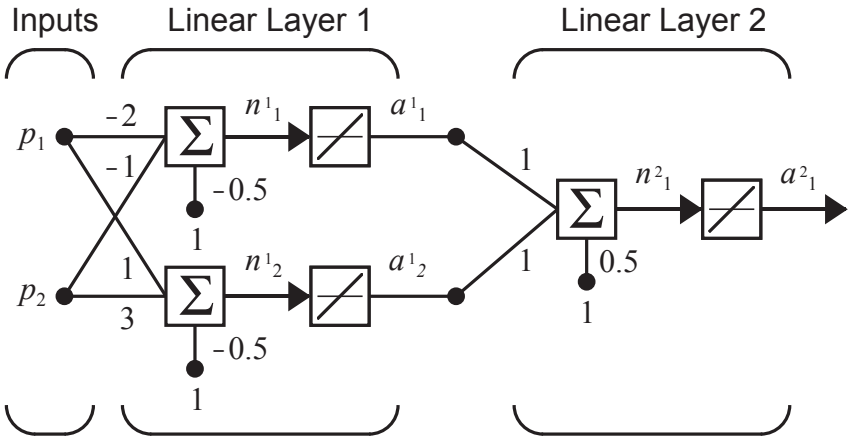


Figure E11.3 Two-Layer Linear Network

E11.4 Use the chain rule to find the derivative $\partial f / \partial w$ in the following cases:

i. $f(n) = \sin(n)$, $n(w) = w^2$.

ii. $f(n) = \tanh(n)$, $n(w) = 5w$.

iii. $f(n) = \exp(n)$, $n(w) = \cos(w)$.

iv. $f(n) = \text{logsig}(n)$, $n(w) = \exp(w)$.

E11.5 Consider again the backpropagation example that begins on page 11-14.

i. Find the squared error $(e)^2$ as an explicit function of all weights and biases.

ii. Using part (i), find $\partial (e)^2 / \partial w^1_{1,1}$ at the initial weights and biases.

iii. Compare the results of part (ii) with the backpropagation results described in the text.

E11.6 For the network shown in Figure E11.4 the initial weights and biases are chosen to be

$$w^1(0) = 1,\ b^1(0) = -2,\ w^2(0) = 1,\ b^2(0) = 1.$$

The network transfer functions are

$$f^1(n) = (n)^2,\ f^2(n) = \frac{1}{n},$$

and an input/target pair is given to be

$$\{p = 1, t = 1\}.$$

Perform one iteration of backpropagation with $\alpha = 1$.

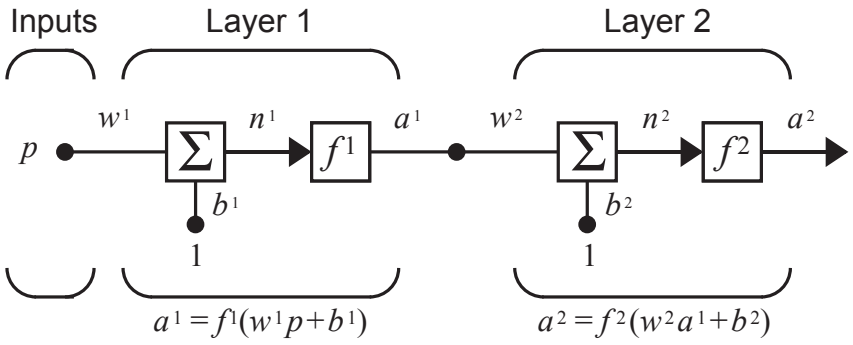


Figure E11.4 Two-Layer Network for Exercise E11.6

E11.7 Consider the two-layer network in Figure E11.5.

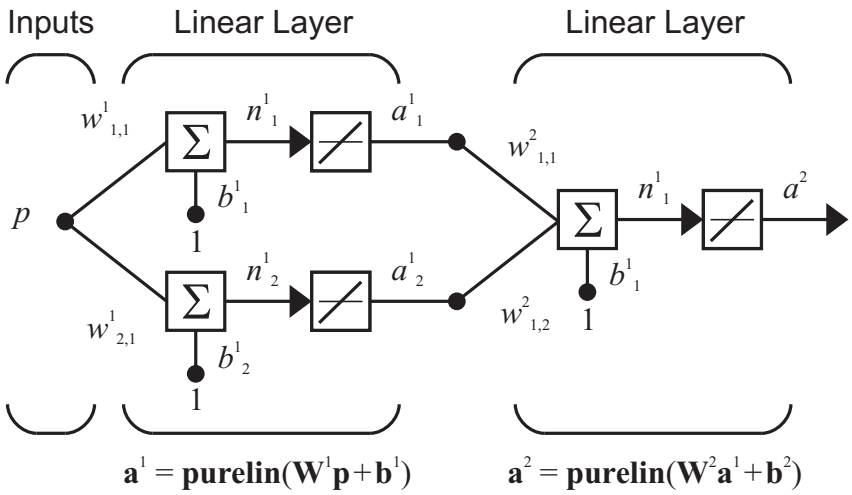


Figure E11.5 Two-Layer Network for Exercise E11.7

with the following input and target: $\{p_1 = 1, t_1 = 2\}$. The initial weights and biases are given by

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} -1 & 1 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 3 \end{bmatrix}$$

i. Apply the input to the network and make one pass forward through the network to compute the output and the error.

ii. Compute the sensitivities by backpropagating through the network.

iii. Compute the derivative $\partial(e)^2/\partial w^1_{1,1}$ using the results of part ii. (Very little calculation is required here.)

E11.8 For the network shown in Figure E11.6 the neuron transfer function is

$$f^1(n) = (n)^2,$$

and an input/target pair is given to be

$$\left\{\mathbf{p} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{t} = \begin{bmatrix} 8 \\ 2 \end{bmatrix}\right\}.$$

Perform one iteration of backpropagation with $\alpha = 1$.

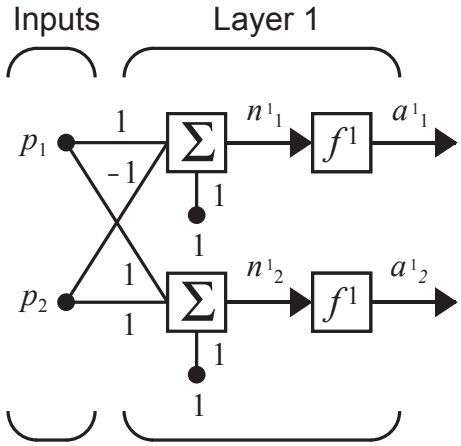


Figure E11.6 Single-Layer Network for Exercise E11.8

E11.9 We want to train the network in Figure E11.7 using the standard back-propagation algorithm (approximate steepest descent).

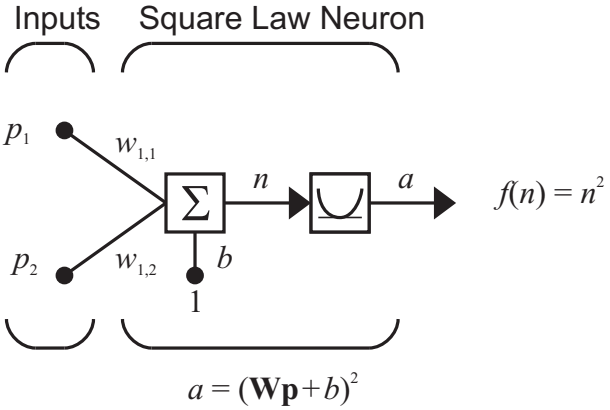


Figure E11.7 Square Law Neuron

The following input and target are given:

$$\left\{\mathbf{p} = \begin{bmatrix}1\\1\end{bmatrix}, t = \begin{bmatrix}0\end{bmatrix}\right\}$$

The initial weights and bias are

$$\mathbf{W}(0) = \begin{bmatrix}1 & -1\end{bmatrix}, b(0) = 1.$$

i. Propagate the input forward through the network.

ii. Compute the error.

iii. Propagate the sensitivities backward through the network.

iv. Compute the gradient of the squared error with respect to the weights and bias.

v. Update the weights and bias (assume a learning rate of $\alpha = 0.1$).

E11.10 Consider the following multilayer perceptron network. (The transfer function of the hidden layer is $f(n) = n^2$.)

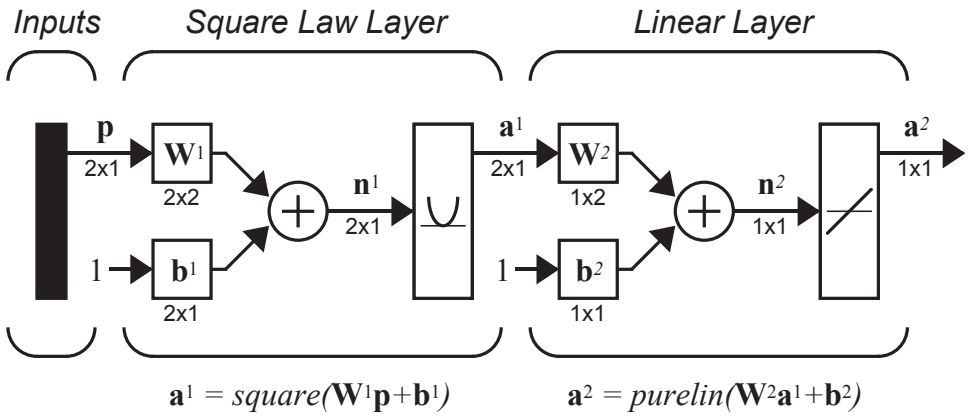


Figure E11.8 Two-Layer Square Law Network

The initial weights and biases are:

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 2 & 1 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} -1 \end{bmatrix}.$$

Perform one iteration of the standard steepest descent backpropagation (use matrix operations) with learning rate $\alpha = 0.5$ for the following input/target pair:

$$\left\{ \mathbf{p} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t = \begin{bmatrix} 2 \end{bmatrix} \right\}$$

E11.11 Consider the network shown in Figure E11.9.

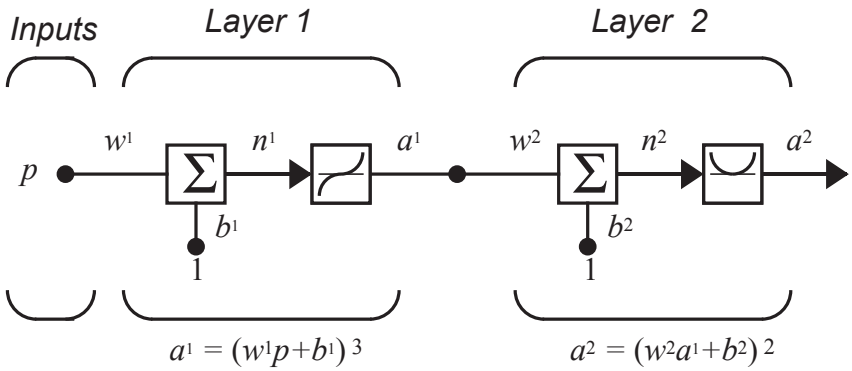


Figure E11.9 Two-Layer Network for Exercise E11.11

The initial weights and biases are chosen to be

$$w^1(0) = -2,\ b^1(0) = 1,\ w^2(0) = 1,\ b^2(0) = -2\ .$$

An input/target pair is given to be

$$\{p_1 = 1, t_1 = 0\}\ ,$$

Perform one iteration of backpropagation (steepest descent) with $\alpha = 1$.

E11.12 Consider the multilayer perceptron network in Figure E11.10. (The transfer function of the hidden layer is $f(n) = n^3$.)

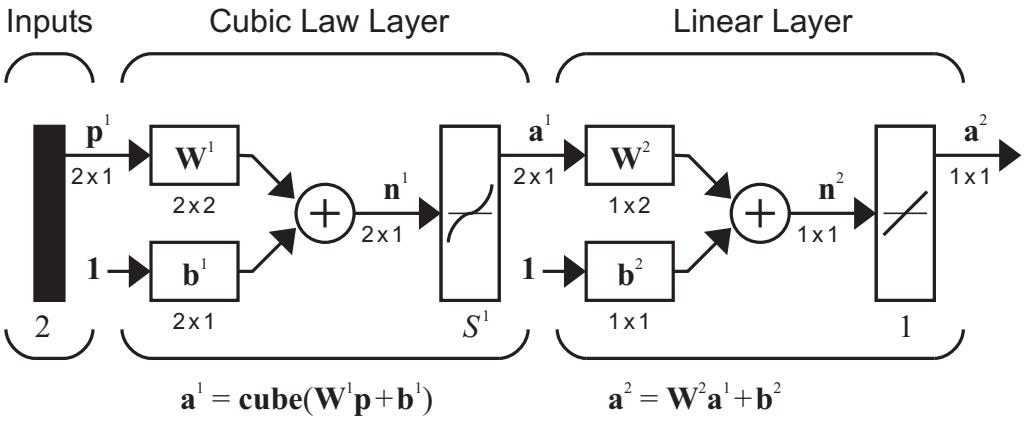


Figure E11.10 Cubic Law Neural Network

The initial weights and biases are:

$$\mathbf{W}^1(0) = \begin{bmatrix} 1 & -1 \\ 1 & 0 \end{bmatrix}, \mathbf{b}^1(0) = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \mathbf{W}^2(0) = \begin{bmatrix} 1 & 1 \end{bmatrix}, \mathbf{b}^2(0) = \begin{bmatrix} 1 \end{bmatrix}.$$

Perform one iteration of the standard steepest descent backpropagation (use matrix operations) with learning rate $\alpha = 0.5$ for the following input/target pair:

$$\left\{ \mathbf{p} = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t = \begin{bmatrix} -1 \end{bmatrix} \right\}.$$

E11.13 Someone has proposed that the standard multilayer network should be modified to include a scalar gain at each layer. This means that the net input at layer m would be computed as

$$\mathbf{n}^m = \beta^m [\mathbf{W}^m \mathbf{a}^{m-1} + \mathbf{b}^m],$$

where β^m is the scalar gain at layer m. This gain would be trained like the weights and biases of the network. Modify the backpropagation algorithm (Eq. (11.41) to Eq. (11.47)) for this new network. (There will need to be a new equation added to update β^m, but some of the other equations may have to be modified as well.)

E11.14 Consider the two-layer network shown in Figure E11.11.

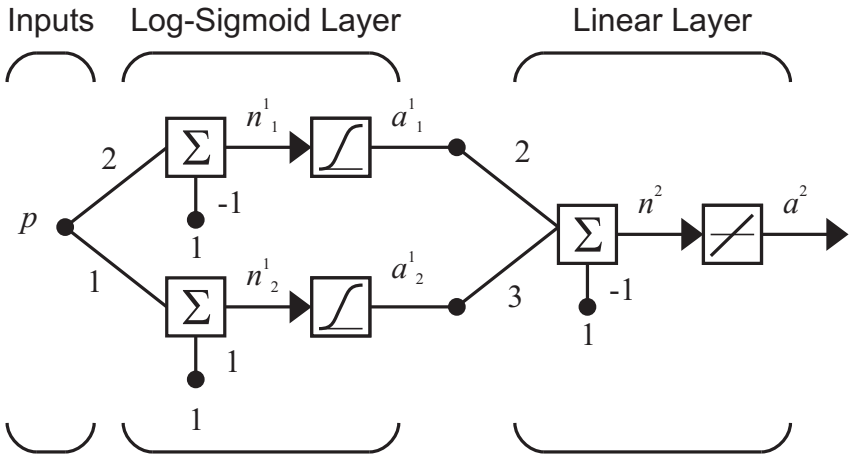


Figure E11.11 Two-Layer Network for Exercise E11.14

i. If $p = 1$, use a (slightly) modified form of backpropagation (as developed in Eq. (11.41) through Eq. (11.47)) to find

$$\frac{\partial a^2}{\partial n^2}, \frac{\partial a^2}{\partial n_1^1}, \frac{\partial a^2}{\partial n_2^1}.$$

ii. Use the results of i. and the chain rule to find $\frac{\partial a^2}{\partial p}$.

Your answers to both parts should be numerical.

E11.15 Consider the network shown in Figure E11.12, where the inputs to the neuron involve both the original inputs and their product. This is a type of higher-order network.

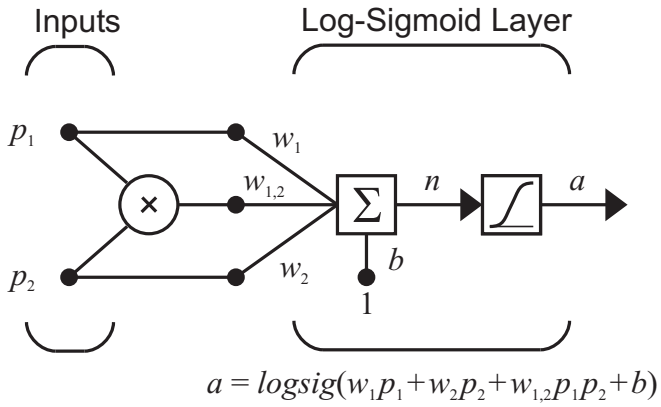


Figure E11.12 Higher-Order Network

i. Find a learning rule for the network parameters, using the approximate steepest descent algorithm (as was done for backpropagation).

ii. For the following initial parameter values, inputs and target, perform one iteration of your learning rule with learning rate $\alpha = 1$:

$$w_1 = 1, w_2 = -1, w_{1,2} = 0.5, b_1 = 1, p_1 = 0, p_2 = 1, t = 0.75$$

E11.16 In Figure E11.13 we have a two-layer network that has an additional connection from the input directly to the second layer. Derive the backpropagation algorithm for this network.

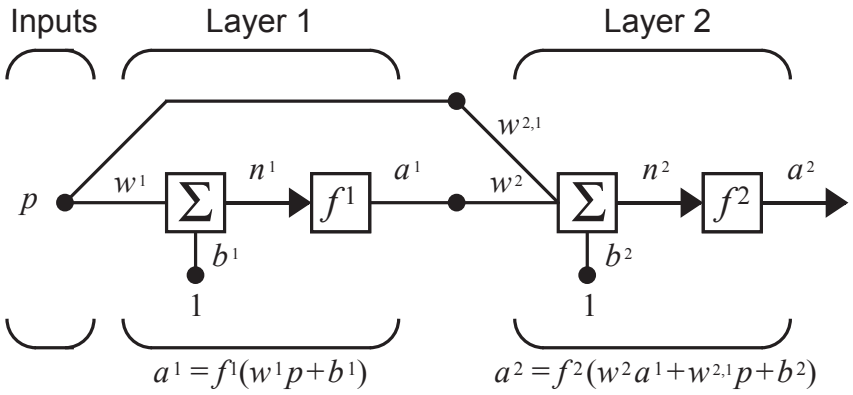


Figure E11.13 Two-Layer Network with Bypass Connection

E11.17 In the multilayer network, the net input is computed as follows

$$\mathbf{n}^{m+1} = \mathbf{W}^{m+1}\mathbf{a}^m + \mathbf{b}^{m+1} \text{ or } n_i^{m+1} = \sum_{j=1}^{S^m} w_{i,j}^{m+1} a_j^m + b_i^{m+1}.$$

If the net input calculation is changed to the following equation (squared distance calculation), how will the sensitivity backpropagation (Eq. (11.35)) change?

$$n_i^{m+1} = \sum_{j=1}^{S^m} (w_{i,j}^{m+1} - a_j^m)^2$$

E11.18 Consider again the net input calculation, as described in Exercise E11.17. If the net input calculation is changed to the following equation (multiply by the bias, instead of add), how will the sensitivity backpropagation (Eq. (11.35)) change?

$$n_i^{m+1} = \left(\sum_{j=1}^{S^m} w_{i,j}^{m+1} a_j^m \right) \times b_i^{m+1}.$$

E11.19 Consider the system shown in Figure E11.14. There are a series of stages, with different transfer functions in each stage. (There are no weights or biases.) We want to take the derivative of the output of this system (a^M) with respect to the input of the system (p). Derive a recursive algorithm that you can use to compute this derivative. Use the concepts that we used to derive the backpropagation algorithm, and use the following intermediate variable in your algorithm:

$$q^i = \frac{\partial a^M}{\partial a^i}.$$

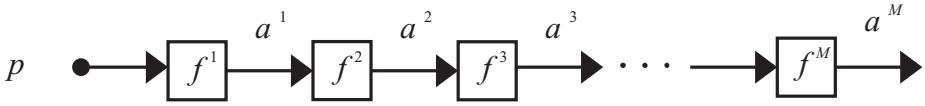


Figure E11.14 Cascade System

E11.20 The backpropagation algorithm is used to compute the gradient of the squared error with respect to the weights and biases of a multilayer network. How would the algorithm be changed if you wanted to compute the gradient with respect to the inputs of the network (i.e., with respect to the elements of the input vector **p**)? Carefully explain all of your steps, and write out the final algorithm.

E11.21 With the standard backpropagation algorithm, we want to compute the derivative

$$\frac{\partial F}{\partial w}.$$

To calculate this derivative, we use the chain rule in the form

$$\frac{\partial F}{\partial w} = \frac{\partial F}{\partial n} \cdot \frac{\partial n}{\partial w}.$$

Suppose that we want to use Newton's method. We would need to find the second derivative

$$\frac{\partial^2 F}{\partial w^2}.$$

What form will the chain rule take in this case?

E11.22 The standard steepest descent backpropagation algorithm, which is summarized in Eq. (11.41) through Eq. (11.47), was designed to minimize the performance function that was the sum of squares of the network errors, as given in Eq. (11.12). Suppose that we want to change the performance function to the sum of the fourth powers of the errors (e^4) plus the sum of the squares of the weights and biases in the network. Show how Eq. (11.41)

through Eq. (11.47) will change for this new performance function. (You don't need to rederive any steps which are already given in this chapter and do not change.)

E11.23 Repeat Problem P11.4 using the "backward" method described below.

In Problem P11.4. we had the dynamic system

$$y(k+1) = f(y(k)) .$$

We had to choose the initial condition $y(0)$ so that at some final time $k = K$ the system output $y(K)$ would be as close as possible to some target output t. We minimized the performance index

$$F(y(0)) = (t - y(K))^2 = e^2(K)$$

using steepest descent, so we needed the gradient

$$\frac{\partial}{\partial y(0)} F(y(0)) .$$

We developed a procedure for computing this gradient using the chain rule. The procedure involved a recursive equation for the term

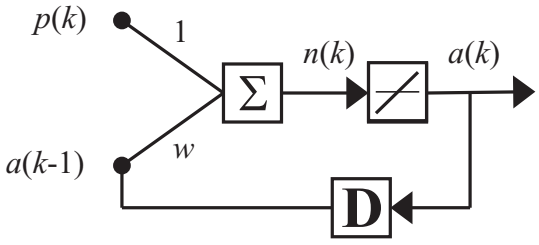
$$r(k) \equiv \frac{\partial}{\partial y(0)} y(k) ,$$

which evolved forward in time. The gradient can also be computed in a different way by evolving the term

$$q(k) \equiv \frac{\partial}{\partial y(k)} e^2(K)$$

backward through time.

E11.24 Consider the recurrent neural network in Figure E11.15.



$$a(k) = p(k) + w\, a(k\text{-}1)$$

Figure E11.15 Recurrent Network

We want to find the weight value w so that at some final time $k = K$ the system output $a(K)$ will be as close as possible to some target output t. We will minimize the performance index $F(w) = (t - a(K))^2$ using steepest descent, so we need the gradient $\partial F(w)/\partial w$.

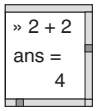
i. Find a general procedure to compute this gradient using the chain rule. Develop an equation to evolve the following term forward through time:

$$s(k) \equiv \frac{\partial}{\partial w} a(k) .$$

Show each step of your entire procedure carefully. This will involve updating $s(k)$ and also computing the gradient $\partial F(w)/\partial w$.

ii. Assume that $K = 3$. Write out the complete expression for $a(3)$ as a function of $p(1)$, $p(2)$, $p(3)$ and w (assuming $a(0) = 0$). Take the derivative of this expression with respect to w, and show that it equals $s(3)$.

E11.25 Write a MATLAB program to implement the backpropagation algorithm for a $1 - S^1 - 1$ network. Write the program using matrix operations, as in Eq. (11.41) to Eq. (11.47). Choose the initial weights and biases to be random numbers uniformly distributed between -0.5 and 0.5 (using the MATLAB function `rand`), and train the network to approximate the function



$$g(p) = 1 + \sin\left(\frac{\pi}{2} p\right) \text{ for } -2 \le p \le 2 .$$

Use $S^1 = 2$ and $S^1 = 10$. Experiment with several different values for the learning rate α, and use several different initial conditions. Discuss the convergence properties of the algorithm as the learning rate changes.