

multilayer networks of such units and consider several general issues such as the representational capabilities of ANNs, nature of the hypothesis space search, overfitting problems, and alternatives to the BACKPROPAGATION algorithm. A detailed example is also presented applying BACKPROPAGATION to face recognition, and directions are provided for the reader to obtain the data and code to experiment further with this application.

## 4.4 PERCEPTRONS

One type of ANN system is based on a unit called a *perceptron*, illustrated in Figure 4.2. A perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and  $-1$  otherwise. More precisely, given inputs  $x_1$  through  $x_n$ , the output  $o(x_1, \dots, x_n)$  computed by the perceptron is

$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

where each  $w_i$  is a real-valued constant, or *weight*, that determines the contribution of input  $x_i$  to the perceptron output. Notice the quantity  $(-w_0)$  is a threshold that the weighted combination of inputs  $w_1x_1 + \dots + w_nx_n$  must surpass in order for the perceptron to output a 1.

To simplify notation, we imagine an additional constant input  $x_0 = 1$ , allowing us to write the above inequality as  $\sum_{i=0}^n w_i x_i > 0$ , or in vector form as  $\vec{w} \cdot \vec{x} > 0$ . For brevity, we will sometimes write the perceptron function as

$$o(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$$

where

$$\text{sgn}(y) = \begin{cases} 1 & \text{if } y > 0 \\ -1 & \text{otherwise} \end{cases}$$

Learning a perceptron involves choosing values for the weights  $w_0, \dots, w_n$ . Therefore, the space  $H$  of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors.

$$H = \{\vec{w} \mid \vec{w} \in \mathbb{R}^{(n+1)}\}$$

### 4.4.1 Representational Power of Perceptrons

We can view the perceptron as representing a hyperplane decision surface in the  $n$ -dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a  $-1$  for instances lying on the other side, as illustrated in Figure 4.3. The equation for this decision hyperplane is  $\vec{w} \cdot \vec{x} = 0$ . Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

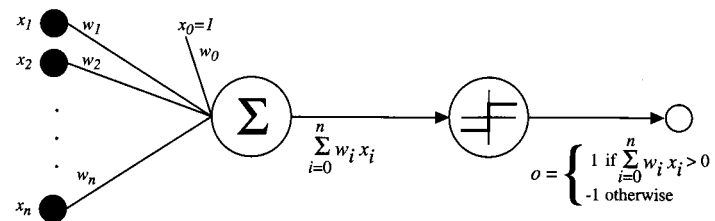


FIGURE 4.2  
A perceptron.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and  $-1$  (false), then one way to use a two-input perceptron to implement the AND function is to set the weights  $w_0 = -.8$ , and  $w_1 = w_2 = .5$ . This perceptron can be made to represent the OR function instead by altering the threshold to  $w_0 = -.3$ . In fact, AND and OR can be viewed as special cases of  $m$ -of- $n$  functions: that is, functions where at least  $m$  of the  $n$  inputs to the perceptron must be true. The OR function corresponds to  $m = 1$  and the AND function to  $m = n$ . Any  $m$ -of- $n$  function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold  $w_0$  accordingly.

Perceptrons can represent all of the primitive boolean functions AND, OR, NAND ( $\neg$  AND), and NOR ( $\neg$  OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if  $x_1 \neq x_2$ . Note the set of linearly nonseparable training examples shown in Figure 4.3(b) corresponds to this XOR function.

The ability of perceptrons to represent AND, OR, NAND, and NOR is important because *every* boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which

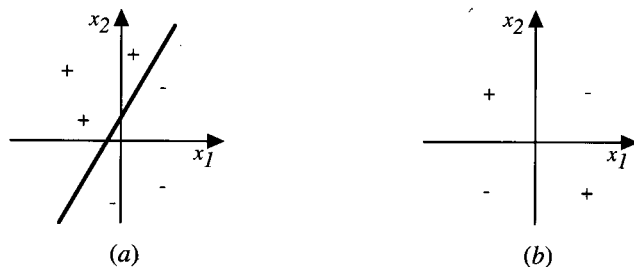


FIGURE 4.3  
The decision surface represented by a two-input perceptron. (a) A set of training examples and decision surface of a perceptron that classifies them correctly. (b) A set of training examples that are not linearly separable (i.e., that cannot be correctly classified by any straight line).  $x_1$  and  $x_2$  are perceptron inputs. Positive examples are indicated by "+", negative by "-".

the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight.

Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

#### 4.4.2 The Perceptron Training Rule

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct  $\pm 1$  output for each of the given training examples.

Several algorithms are known to solve this learning problem. Here we consider two: the perceptron rule and the delta rule (a variant of the LMS rule used in Chapter 1 for learning evaluation functions). These two algorithms are guaranteed to converge to somewhat different acceptable hypotheses, under somewhat different conditions. They are important to ANNs because they provide the basis for learning networks of many units.

One way to learn an acceptable weight vector is to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the *perceptron training rule*, which revises the weight  $w_i$  associated with input  $x_i$  according to the rule

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

Here  $t$  is the target output for the current training example,  $o$  is the output generated by the perceptron, and  $\eta$  is a positive constant called the *learning rate*. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Why should this update rule converge toward successful weight values? To get an intuitive feel, consider some specific cases. Suppose the training example is correctly classified already by the perceptron. In this case,  $(t - o)$  is zero, making  $\Delta w_i$  zero, so that no weights are updated. Suppose the perceptron outputs a  $-1$ , when the target output is  $+1$ . To make the perceptron output a  $+1$  instead of  $-1$  in this case, the weights must be altered to increase the value of  $\vec{w} \cdot \vec{x}$ . For example, if  $x_i > 0$ , then increasing  $w_i$  will bring the perceptron closer to correctly classifying

this example. Notice the training rule will increase  $w_i$  in this case, because  $\eta$ , and  $x_i$  are all positive. For example, if  $x_i = .8$ ,  $\eta = 0.1$ ,  $t = 1$ , and  $o = -1$ , then the weight update will be  $\Delta w_i = \eta(t - o)x_i = 0.1(1 - (-1))0.8 = 0.16$ . On the other hand, if  $t = -1$  and  $o = 1$ , then weights associated with positive inputs will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge in a finite number of applications of the perceptron training rule to a weight vector that correctly classifies all training examples, *provided the training examples are linearly separable* and provided a sufficiently small  $\eta$  is used (see Rosenblatt and Papert 1969). If the data are not linearly separable, convergence is not guaranteed.

#### 4.4.3 Gradient Descent and the Delta Rule

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the *delta rule*, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept.

The key idea behind the delta rule is to use *gradient descent* to search through the hypothesis space of possible weight vectors to find the weights that best fit the training examples. This rule is important because gradient descent provides the basis for the BACKPROPAGATION algorithm, which can learn networks with many interconnected units. It is also important because gradient descent can serve as the basis for learning algorithms that must search through hypothesis spaces containing many different types of continuously parameterized hypotheses.

The delta training rule is best understood by considering the task of learning an *unthresholded* perceptron; that is, a *linear unit* for which the output  $o$  is

$$o(\vec{x}) = \vec{w} \cdot \vec{x}$$

Thus, a linear unit corresponds to the first stage of a perceptron, without a threshold.

In order to derive a weight learning rule for linear units, let us specify a measure for the *training error* of a hypothesis (weight vector) relative to the training examples. Although there are many ways to define this error, a common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

where  $D$  is the set of training examples,  $t_d$  is the target output for training example  $d$ , and  $o_d$  is the output of the linear unit for training example  $d$ . By this definition,  $E(\vec{w})$  is simply half the squared difference between the target output  $t_d$  and the linear unit output  $o_d$ , summed over all training examples. Here we choose  $E$  as a function of  $\vec{w}$  because the linear unit output  $o$  depends on the weight vector. Of course  $E$  also depends on the particular set of training examples.

we assume these are fixed during training, so we do not bother to write  $E$  as an explicit function of these. Chapter 6 provides a Bayesian justification for choosing this particular definition of  $E$ . In particular, there we show that under certain conditions the hypothesis that minimizes  $E$  is also the most probable hypothesis in  $H$  given the training data.

#### 4.4.3.1 VISUALIZING THE HYPOTHESIS SPACE

To understand the gradient descent algorithm, it is helpful to visualize the entire hypothesis space of possible weight vectors and their associated  $E$  values, as illustrated in Figure 4.4. Here the axes  $w_0$  and  $w_1$  represent possible values for the two weights of a simple linear unit. The  $w_0, w_1$  plane therefore represents the entire hypothesis space. The vertical axis indicates the error  $E$  relative to some fixed set of training examples. The error surface shown in the figure thus summarizes the desirability of every weight vector in the hypothesis space (we desire a hypothesis with minimum error). Given the way in which we chose to define  $E$ , for linear units this error surface must always be parabolic with a single global minimum. The specific parabola will depend, of course, on the particular set of training examples.

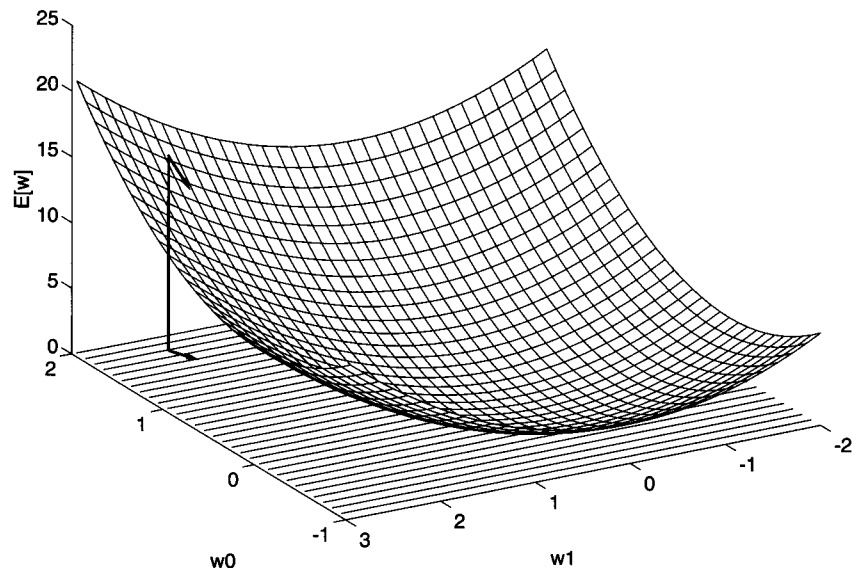


FIGURE 4.4

Error of different hypotheses. For a linear unit with two weights, the hypothesis space  $H$  is the  $w_0, w_1$  plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular point, indicating the direction in the  $w_0, w_1$  plane producing steepest descent along the error surface.

Gradient descent search determines a weight vector that minimizes  $E$  starting with an arbitrary initial weight vector, then repeatedly modify it in small steps. At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface depicted in Figure 4.4. This process continues until the global minimum error is reached.

#### 4.4.3.2 DERIVATION OF THE GRADIENT DESCENT RULE

How can we calculate the direction of steepest descent along the error surface? This direction can be found by computing the derivative of  $E$  with respect to each component of the vector  $\vec{w}$ . This vector derivative is called the *gradient* of  $E$  with respect to  $\vec{w}$ , written  $\nabla E(\vec{w})$ .

$$\nabla E(\vec{w}) \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$$

Notice  $\nabla E(\vec{w})$  is itself a vector, whose components are the partial derivatives of  $E$  with respect to each of the  $w_i$ . When interpreted as a vector in the hypothesis space, the gradient specifies the direction that produces the steepest increase in  $E$ . The negative of this vector therefore gives the direction of steepest descent. For example, the arrow in Figure 4.4 shows the negated gradient  $-\nabla E(\vec{w})$  at a particular point in the  $w_0, w_1$  plane.

Since the gradient specifies the direction of steepest increase of  $E$ , the update rule for gradient descent is

$$\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$$

where

$$\Delta \vec{w} = -\eta \nabla E(\vec{w})$$

Here  $\eta$  is a positive constant called the learning rate, which determines the step size in the gradient descent search. The negative sign is present because we want to move the weight vector in the direction that *decreases*  $E$ . This training rule can also be written in its component form

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

which makes it clear that steepest descent is achieved by altering each component  $w_i$  of  $\vec{w}$  in proportion to  $\frac{\partial E}{\partial w_i}$ .

To construct a practical algorithm for iteratively updating weights according to Equation (4.5), we need an efficient way of calculating the gradient at each step. Fortunately, this is not difficult. The vector of  $\frac{\partial E}{\partial w_i}$  derivatives that f

gradient can be obtained by differentiating  $E$  from Equation (4.2), as

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d) (-x_{id}) \end{aligned} \quad (4.6)$$

where  $x_{id}$  denotes the single input component  $x_i$  for training example  $d$ . We now have an equation that gives  $\frac{\partial E}{\partial w_i}$  in terms of the linear unit inputs  $x_{id}$ , outputs  $O_d$ , and target values  $t_d$  associated with the training examples. Substituting Equation (4.6) into Equation (4.5) yields the weight update rule for gradient descent

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \quad (4.7)$$

To summarize, the gradient descent algorithm for training linear units is as follows: Pick an initial random weight vector. Apply the linear unit to all training examples, then compute  $\Delta w_i$  for each weight according to Equation (4.7). Update each weight  $w_i$  by adding  $\Delta w_i$ , then repeat this process. This algorithm is given in Table 4.1. Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate  $\eta$  is used. If  $\eta$  is too large, the gradient descent search runs the risk of overstepping the minimum in the error surface rather than settling into it. For this reason, one common modification to the algorithm is to gradually reduce the value of  $\eta$  as the number of gradient descent steps grows.

#### 4.4.3.3 STOCHASTIC APPROXIMATION TO GRADIENT DESCENT

Gradient descent is an important general paradigm for learning. It is a strategy for searching through a large or infinite hypothesis space that can be applied whenever (1) the hypothesis space contains continuously parameterized hypotheses (e.g., the weights in a linear unit), and (2) the error can be differentiated with respect to these hypothesis parameters. The key practical difficulties in applying gradient descent are (1) converging to a local minimum can sometimes be quite slow (i.e., it can require many thousands of gradient descent steps), and (2) if there are multiple local minima in the error surface, then there is no guarantee that the procedure will find the global minimum.

#### GRADIENT-DESCENT(*training\_examples*, $\eta$ )

Each training example is a pair of the form  $(\vec{x}, t)$ , where  $\vec{x}$  is the vector of input values and  $t$  is the target output value.  $\eta$  is the learning rate (e.g., .05).

- Initialize each  $w_i$  to some small random value
- Until the termination condition is met, Do
  - Initialize each  $\Delta w_i$  to zero.
  - For each  $(\vec{x}, t)$  in *training\_examples*, Do
    - Input the instance  $\vec{x}$  to the unit and compute the output  $o$
    - For each linear unit weight  $w_i$ , Do
- For each linear unit weight  $w_i$ , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

$$w_i \leftarrow w_i + \Delta w_i$$

TABLE 4.1

GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation, Equation (T4.2) is deleted, and Equation (T4.1) replaced by  $w_i \leftarrow w_i +$

One common variation on gradient descent intended to alleviate the difficulties is called *incremental gradient descent*, or alternatively *stochastic gradient descent*. Whereas the gradient descent training rule presented in Equation (4.7) computes weight updates after summing over *all* the training examples in the set, the idea behind stochastic gradient descent is to approximate this gradient search by updating weights incrementally, following the calculation of the gradient for *each* individual example. The modified training rule is like the training rule given by Equation (4.7) except that as we iterate through each training example, we update the weight according to

$$\Delta w_i = \eta(t - o) x_i$$

where  $t$ ,  $o$ , and  $x_i$  are the target value, unit output, and  $i$ th input for the current training example in question. To modify the gradient descent algorithm of Table 4.1 to implement this stochastic approximation, Equation (T4.2) is simply deleted, and Equation (T4.1) replaced by  $w_i \leftarrow w_i + \eta(t - o) x_i$ . One way to view this stochastic gradient descent is to consider a distinct error function  $E_d(\vec{w})$  defined for each individual training example  $d$  as follows

$$E_d(\vec{w}) = \frac{1}{2} (t_d - o_d)^2$$

where  $t_d$  and  $o_d$  are the target value and the unit output value for training example  $d$ . Stochastic gradient descent iterates over the training examples  $d$  at each iteration altering the weights according to the gradient with respect to  $E_d(\vec{w})$ . The sequence of these weight updates, when iterated over all the training examples, provides a reasonable approximation to descending the gradient of the error function  $E(\vec{w})$  with respect to our original error function  $E(\vec{w})$ . By making the value of  $\eta$  (the g

descent step size) sufficiently small, stochastic gradient descent can be made to approximate true gradient descent arbitrarily closely. The key differences between standard gradient descent and stochastic gradient descent are:

- In standard gradient descent, the error is summed over all examples before updating weights, whereas in stochastic gradient descent weights are updated upon examining each training example.
- Summing over multiple examples in standard gradient descent requires more computation per weight update step. On the other hand, because it uses the true gradient, standard gradient descent is often used with a larger step size per weight update than stochastic gradient descent.
- In cases where there are multiple local minima with respect to  $E(\vec{w})$ , stochastic gradient descent can sometimes avoid falling into these local minima because it uses the various  $\nabla E_d(\vec{w})$  rather than  $\nabla E(\vec{w})$  to guide its search.

Both stochastic and standard gradient descent methods are commonly used in practice.

The training rule in Equation (4.10) is known as the *delta rule*, or sometimes the LMS (least-mean-square) rule, Adaline rule, or Widrow-Hoff rule (after its inventors). In Chapter 1 we referred to it as the LMS weight-update rule when describing its use for learning an evaluation function for game playing. Notice the delta rule in Equation (4.10) is similar to the perceptron training rule in Equation (4.4.2). In fact, the two expressions appear to be identical. However, the rules are different because in the delta rule  $o$  refers to the linear unit output  $o(\vec{x}) = \vec{w} \cdot \vec{x}$ , whereas for the perceptron rule  $o$  refers to the thresholded output  $o(\vec{x}) = \text{sgn}(\vec{w} \cdot \vec{x})$ .

Although we have presented the delta rule as a method for learning weights for unthresholded linear units, it can easily be used to train thresholded perceptron units, as well. Suppose that  $o = \vec{w} \cdot \vec{x}$  is the unthresholded linear unit output as above, and  $o' = \text{sgn}(\vec{w} \cdot \vec{x})$  is the result of thresholding  $o$  as in the perceptron. Now if we wish to train a perceptron to fit training examples with target values of  $\pm 1$  for  $o'$ , we can use these same target values and examples to train  $o$  instead, using the delta rule. Clearly, if the unthresholded output  $o$  can be trained to fit these values perfectly, then the threshold output  $o'$  will fit them as well (because  $\text{sgn}(1) = 1$ , and  $\text{sgn}(-1) = -1$ ). Even when the target values cannot be fit perfectly, the thresholded  $o'$  value will correctly fit the  $\pm 1$  target value whenever the linear unit output  $o$  has the correct sign. Notice, however, that while this procedure will learn weights that minimize the error in the linear unit output  $o$ , these weights will not necessarily minimize the number of training examples misclassified by the thresholded output  $o'$ .

#### 4.4.4 Remarks

We have considered two similar algorithms for iteratively learning perceptron weights. The key difference between these algorithms is that the perceptron train-

ing rule updates weights based on the error in the *thresholded* perceptron output, whereas the delta rule updates weights based on the error in the *unthresholded* linear combination of inputs.

The difference between these two training rules is reflected in different convergence properties. The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, *provided the training examples are linearly separable*. The delta rule converges only asymptotically toward the minimum error hypothesis, possibly requiring unbounded time, but converges *regardless of whether the training data are linearly separable*. A detailed presentation of the convergence proofs can be found in [Duda et al. (1991)].

A third possible algorithm for learning the weight vector is linear programming. Linear programming is a general, efficient method for solving sets of linear inequalities. Notice each training example corresponds to an inequality of the form  $\vec{w} \cdot \vec{x} > 0$  or  $\vec{w} \cdot \vec{x} \leq 0$ , and their solution is the desired weight vector. Unfortunately, this approach yields a solution only when the training examples are linearly separable; however, Duda and Hart (1973, p. 168) suggest a more general formulation that accommodates the nonseparable case. In any case, the power of linear programming does not scale to training multilayer networks, our primary concern. In contrast, the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks, as discussed in the following section.

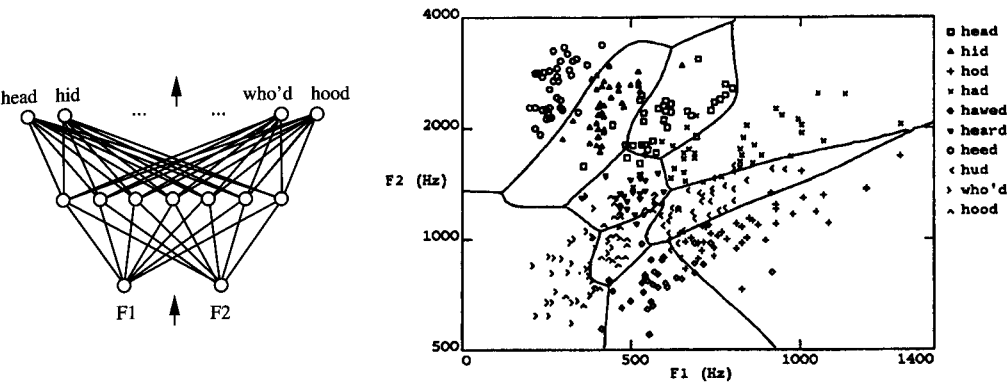
## 4.5 MULTILAYER NETWORKS AND THE BACKPROPAGATION ALGORITHM

As noted in Section 4.4.1, single perceptrons can only express linear decision surfaces. In contrast, the kind of multilayer networks learned by the BACKPROPAGATION algorithm are capable of expressing a rich variety of nonlinear decision surfaces. For example, a typical multilayer network and decision surface are depicted in Figure 4.5. Here the speech recognition task involves distinguishing among 10 possible vowels, all spoken in the context of “h\_d” (i.e., “hid,” “head,” “hood,” etc.). The input speech signal is represented by two parameters obtained from a spectral analysis of the sound, allowing us to visualize the decision surface over the two-dimensional instance space. As shown in the figure, it is possible for the multilayer network to represent highly complex decision surfaces that are much more expressive than the linear decision surfaces of single units shown earlier in Figure 4.3.

This section discusses how to learn such multilayer networks using the gradient descent algorithm similar to that discussed in the previous section.

### 4.5.1 A Differentiable Threshold Unit

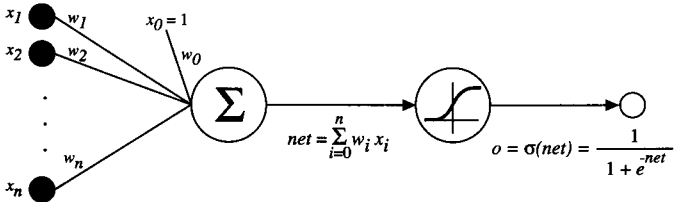
What type of unit shall we use as the basis for constructing multilayer networks? At first we might be tempted to choose the linear units discussed in the



**FIGURE 4.5** Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context “h\_d” (e.g., “had,” “hid”). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network outputs correspond to the 10 possible vowel sounds. The network prediction is the output whose value is highest. The plot on the right illustrates the highly nonlinear decision surface represented by the learned network. Points shown on the plot are test examples distinct from the examples used to train the network. (Reprinted by permission from Haung and Lippmann (1988).)

section, for which we have already derived a gradient descent learning rule. However, multiple layers of cascaded linear units still produce only linear functions, and we prefer networks capable of representing highly nonlinear functions. The perceptron unit is another possible choice, but its discontinuous threshold makes it undifferentiable and hence unsuitable for gradient descent. What we need is a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs. One solution is the *sigmoid unit*—a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

The sigmoid unit is illustrated in Figure 4.6. Like the perceptron, the sigmoid unit first computes a linear combination of its inputs, then applies a threshold to the result. In the case of the sigmoid unit, however, the threshold output is a



**FIGURE 4.6** The sigmoid threshold unit.

continuous function of its input. More precisely, the sigmoid unit computes an output  $o$  as

$$o = \sigma(\vec{w} \cdot \vec{x})$$

where

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

$\sigma$  is often called the sigmoid function or, alternatively, the logistic function. Its output ranges between 0 and 1, increasing monotonically with its input (see the threshold function plot in Figure 4.6.). Because it maps a very large input range to a small range of outputs, it is often referred to as the *squashing function* of the unit. The sigmoid function has the useful property that its derivative can be expressed in terms of its output [in particular,  $\frac{d\sigma(y)}{dy} = \sigma(y) \cdot (1 - \sigma(y))$ ]. As we shall see, the gradient descent learning rule makes use of this derivative. Other differentiable functions with easily calculated derivatives are sometimes used in place of  $\sigma$ . For example, the term  $e^{-k \cdot y}$  in the sigmoid function can sometimes be replaced by  $e^{-k \cdot y}$  where  $k$  is some positive constant that determines the steepness of the threshold. The function *tanh* is also sometimes used instead of the sigmoid function (see Exercise 4.8).

### 4.5.2 The BACKPROPAGATION Algorithm

The BACKPROPAGATION algorithm learns the weights for a multilayer network given a network with a fixed set of units and interconnections. It employs gradient descent to attempt to minimize the squared error between the network output values and the target values for these outputs. This section presents the BACKPROPAGATION algorithm, and the following section gives the derivation for the gradient descent weight update rule used by BACKPROPAGATION.

Because we are considering networks with multiple output units rather than single units as before, we begin by redefining  $E$  to sum the errors over all network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2$$

where *outputs* is the set of output units in the network, and  $t_{kd}$  and  $o_{kd}$  are the target and output values associated with the  $k$ th output unit and training example  $d$ .

The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network. The situation can be visualized in terms of an error surface similar to that shown for linear units in Figure 4.4. The error in that diagram is replaced by the definition of  $E$ , and the other dimensions of the space correspond now to the weights associated with all of the units in the network. As in the case of training a single unit, gradient descent can be used to attempt to find a hypothesis to minimize  $E$ .

BACKPROPAGATION(*training\_examples*,  $\eta$ ,  $n_{in}$ ,  $n_{out}$ ,  $n_{hidden}$ )

Each training example is a pair of the form  $(\vec{x}, \vec{t})$ , where  $\vec{x}$  is the vector of network input values, and  $\vec{t}$  is the vector of target network output values.

$\eta$  is the learning rate (e.g., .05).  $n_{in}$  is the number of network inputs,  $n_{hidden}$  the number of units in the hidden layer, and  $n_{out}$  the number of output units.

The input from unit  $i$  into unit  $j$  is denoted  $x_{ji}$ , and the weight from unit  $i$  to unit  $j$  is denoted  $w_{ji}$ .

- Create a feed-forward network with  $n_{in}$  inputs,  $n_{hidden}$  hidden units, and  $n_{out}$  output units.
- Initialize all network weights to small random numbers (e.g., between  $-.05$  and  $.05$ ).
- Until the termination condition is met, Do
  - For each  $(\vec{x}, \vec{t})$  in *training\_examples*, Do

Propagate the input forward through the network:

1. Input the instance  $\vec{x}$  to the network and compute the output  $o_u$  of every unit  $u$  in the network.

Propagate the errors backward through the network:

2. For each network output unit  $k$ , calculate its error term  $\delta_k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \quad (\text{T4.3})$$

3. For each hidden unit  $h$ , calculate its error term  $\delta_h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh} \delta_k \quad (\text{T4.4})$$

4. Update each network weight  $w_{ji}$

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji} \quad (\text{T4.5})$$

TABLE 4.2

The stochastic gradient descent version of the BACKPROPAGATION algorithm for feedforward networks containing two layers of sigmoid units.

One major difference in the case of multilayer networks is that the error surface can have multiple local minima, in contrast to the single-minimum parabolic error surface shown in Figure 4.4. Unfortunately, this means that gradient descent is guaranteed only to converge toward some local minimum, and not necessarily the global minimum error. Despite this obstacle, in practice BACKPROPAGATION has been found to produce excellent results in many real-world applications.

The BACKPROPAGATION algorithm is presented in Table 4.2. The algorithm as described here applies to layered feedforward networks containing two layers of sigmoid units, with units at each layer connected to all units from the preceding layer. This is the incremental, or stochastic, gradient descent version of BACKPROPAGATION. The notation used here is the same as that used in earlier sections, with the following extensions:

- An index (e.g., an integer) is assigned to each node in the network. A “node” is either an input to the network or the output of some network.
- $x_{ji}$  denotes the input from node  $i$  to unit  $j$ , and  $w_{ji}$  denotes the corresponding weight.
- $\delta_n$  denotes the error term associated with unit  $n$ . It plays a role similar to the quantity  $(t - o)$  in our earlier discussion of the delta training rule. We shall see later,  $\delta_n = -\frac{\partial E}{\partial net_n}$ .

Notice the algorithm in Table 4.2 begins by constructing a network with the desired number of hidden and output units and initializing all network weights to small random values. Given this fixed network structure, the main loop of the algorithm then repeatedly iterates over the training examples. For each training example, it applies the network to the example, calculates the error of the network output for this example, computes the gradient with respect to the error for this example, then updates all weights in the network. This gradient descent process is iterated (often thousands of times, using the same training examples) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is very similar to the delta training rule (Equation [4.10]). Like the delta rule, each weight is updated in proportion to the learning rate  $\eta$ , the input value  $x_{ji}$ , and the error term  $\delta_j$ . The weight is applied, and the error in the output of the unit. The difference is that the error  $(t - o)$  in the delta rule is replaced by a more general error term,  $\delta_j$ . The exact form of  $\delta_j$  follows from the derivation of the delta training rule given in Section 4.5.3. To understand it intuitively, first notice how  $\delta_k$  is computed for each network output unit  $k$  (Equation [T4.3] in Table 4.2).  $\delta_k$  is simply the familiar  $(t_k - o_k)$  from the delta rule, multiplied by the factor  $o_k(1 - o_k)$ , which is the derivative of the sigmoid squashing function. The  $\delta_h$  value for each hidden unit  $h$  has a similar form (Equation [T4.4] in Table 4.2). However, since training examples provide target values  $t_k$  for the network outputs, no target values are directly available to indicate the error in the hidden units' values. Instead, the error term for hidden unit  $h$  is calculated by summing the error terms  $\delta_k$  for each output unit influenced by  $h$ , weighted by the  $\delta_k$ 's by  $w_{kh}$ , the weight from hidden unit  $h$  to output unit  $k$ . This error term characterizes the degree to which hidden unit  $h$  is “responsible for” the error in the output unit  $k$ .

The algorithm in Table 4.2 updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic version of gradient descent. To obtain the true gradient of  $E$  one would average the  $\delta_j x_{ji}$  values over all training examples before altering weight values.

The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application. A variety of termination conditions can be used to halt the procedure. One may choose to halt after a fixed number of iterations through the loop, or once the error on the training examples falls below a certain threshold, or once the error on a separate validation set of examples m

criterion. The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data. This issue is discussed in greater detail in Section 4.6.5.

#### 4.5.2.1 ADDING MOMENTUM

Because BACKPROPAGATION is such a widely used algorithm, many variations have been developed. Perhaps the most common is to alter the weight-update rule in Equation (T4.5) in the algorithm by making the weight update on the  $n$ th iteration depend partially on the update that occurred during the  $(n-1)$ th iteration, as follows:

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1) \quad (4.18)$$

Here  $\Delta w_{ji}(n)$  is the weight update performed during the  $n$ th iteration through the main loop of the algorithm, and  $0 \leq \alpha < 1$  is a constant called the *momentum*. Notice the first term on the right of this equation is just the weight-update rule of Equation (T4.5) in the BACKPROPAGATION algorithm. The second term on the right is new and is called the momentum term. To see the effect of this momentum term, consider that the gradient descent search trajectory is analogous to that of a (momentumless) ball rolling down the error surface. The effect of  $\alpha$  is to add momentum that tends to keep the ball rolling in the same direction from one iteration to the next. This can sometimes have the effect of keeping the ball rolling through small local minima in the error surface, or along flat regions in the surface where the ball would stop if there were no momentum. It also has the effect of gradually increasing the step size of the search in regions where the gradient is unchanging, thereby speeding convergence.

#### 4.5.2.2 LEARNING IN ARBITRARY ACYCLIC NETWORKS

The definition of BACKPROPAGATION presented in Table 4.2 applies only to two-layer networks. However, the algorithm given there easily generalizes to feedforward networks of arbitrary depth. The weight update rule seen in Equation (T4.5) is retained, and the only change is to the procedure for computing  $\delta$  values. In general, the  $\delta_r$  value for a unit  $r$  in layer  $m$  is computed from the  $\delta$  values at the next deeper layer  $m+1$  according to

$$\delta_r = o_r (1 - o_r) \sum_{s \in \text{layer } m+1} w_{sr} \delta_s \quad (4.19)$$

Notice this is identical to Step 3 in the algorithm of Table 4.2, so all we are really saying here is that this step may be repeated for any number of hidden layers in the network.

It is equally straightforward to generalize the algorithm to any directed acyclic graph, regardless of whether the network units are arranged in uniform layers as we have assumed up to now. In the case that they are not, the rule for calculating  $\delta$  for any internal unit (i.e., any unit that is not an output) is

$$\delta_r = o_r (1 - o_r) \sum_{s \in \text{Downstream}(r)} w_{sr} \delta_s \quad (4.20)$$

where  $\text{Downstream}(r)$  is the set of units immediately downstream from the network: that is, all units whose inputs include the output of unit  $r$ . The general form of the weight-update rule that we derive in Section 4.5.3.

#### 4.5.3 Derivation of the BACKPROPAGATION Rule

This section presents the derivation of the BACKPROPAGATION weight-update rule. It may be skipped on a first reading, without loss of continuity.

The specific problem we address here is deriving the stochastic gradient descent rule implemented by the algorithm in Table 4.2. Recall from Equation (T4.2) that stochastic gradient descent involves iterating through the training examples one at a time, for each training example  $d$  descending the gradient of  $E_d$  with respect to this single example. In other words, for each training example  $d$  every weight  $w_{ji}$  is updated by adding to it  $\Delta w_{ji}$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

where  $E_d$  is the error on training example  $d$ , summed over all output units in the network

$$E_d(\vec{w}) \equiv \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$$

Here  $\text{outputs}$  is the set of output units in the network,  $t_k$  is the target value for training example  $d$ , and  $o_k$  is the output of unit  $k$  given training example  $d$ .

The derivation of the stochastic gradient descent rule is conceptually straightforward, but requires keeping track of a number of subscripts and variables. We will follow the notation shown in Figure 4.6, adding a subscript  $j$  to  $w_{ji}$  to denote the  $j$ th unit of the network as follows:

- $x_{ji}$  = the  $i$ th input to unit  $j$
- $w_{ji}$  = the weight associated with the  $i$ th input to unit  $j$
- $\text{net}_j = \sum_i w_{ji} x_{ji}$  (the weighted sum of inputs for unit  $j$ )
- $o_j$  = the output computed by unit  $j$
- $t_j$  = the target output for unit  $j$
- $\sigma$  = the sigmoid function
- $\text{outputs}$  = the set of units in the final layer of the network
- $\text{Downstream}(j)$  = the set of units whose immediate inputs include the output of unit  $j$

We now derive an expression for  $\frac{\partial E_d}{\partial w_{ji}}$  in order to implement the stochastic gradient descent rule seen in Equation (4.21). To begin, notice that  $w_{ji}$  can influence the rest of the network only through  $\text{net}_j$ . Therefore, we can



chain rule to write

$$\begin{aligned}\frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji}\end{aligned}\quad (4.22)$$

Given Equation (4.22), our remaining task is to derive a convenient expression for  $\frac{\partial E_d}{\partial net_j}$ . We consider two cases in turn: the case where unit  $j$  is an output unit for the network, and the case where  $j$  is an internal unit.

**Case 1: Training Rule for Output Unit Weights.** Just as  $w_{ji}$  can influence the rest of the network only through  $net_j$ ,  $net_j$  can influence the network only through  $o_j$ . Therefore, we can invoke the chain rule again to write

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j}\quad (4.23)$$

To begin, consider just the first term in Equation (4.23)

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in \text{outputs}} (t_k - o_k)^2$$

The derivatives  $\frac{\partial}{\partial o_j} (t_k - o_k)^2$  will be zero for all output units  $k$  except when  $k = j$ . We therefore drop the summation over output units and simply set  $k = j$ .

$$\begin{aligned}\frac{\partial E_d}{\partial o_j} &= \frac{\partial}{\partial o_j} \frac{1}{2} (t_j - o_j)^2 \\ &= \frac{1}{2} 2(t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j} \\ &= -(t_j - o_j)\end{aligned}\quad (4.24)$$

Next consider the second term in Equation (4.23). Since  $o_j = \sigma(net_j)$ , the derivative  $\frac{\partial o_j}{\partial net_j}$  is just the derivative of the sigmoid function, which we have already noted is equal to  $\sigma(net_j)(1 - \sigma(net_j))$ . Therefore,

$$\begin{aligned}\frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j)\end{aligned}\quad (4.25)$$

Substituting expressions (4.24) and (4.25) into (4.23), we obtain

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j) o_j(1 - o_j)\quad (4.26)$$

and combining this with Equations (4.21) and (4.22), we have the standard gradient descent rule for output units

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta (t_j - o_j) o_j(1 - o_j) x_{ji}$$

Note this training rule is exactly the weight update rule implemented by Equations (T4.3) and (T4.5) in the algorithm of Table 4.2. Furthermore, we now know that  $\delta_k$  in Equation (T4.3) is equal to the quantity  $-\frac{\partial E_d}{\partial net_k}$ . In the remainder of this section we will use  $\delta_i$  to denote the quantity  $-\frac{\partial E_d}{\partial net_i}$  for an arbitrary unit  $i$ .

**Case 2: Training Rule for Hidden Unit Weights.** In the case where  $j$  is an internal, or hidden unit in the network, the derivation of the training rule must take into account the indirect ways in which  $w_{ji}$  can influence the network outputs and hence  $E_d$ . For this reason, we will find it useful to refer to the set of all units immediately downstream of unit  $j$  in the network (i.e., all units whose direct inputs include the output of unit  $j$ ). We denote this set of units as  $Downstream(j)$ . Notice that  $net_j$  can influence the network outputs (and thus  $E_d$ ) only through the units in  $Downstream(j)$ . Therefore, we can write

$$\begin{aligned}\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} \\ &= \sum_{k \in Downstream(j)} -\delta_k w_{kj} o_j(1 - o_j)\end{aligned}$$

Rearranging terms and using  $\delta_j$  to denote  $-\frac{\partial E_d}{\partial net_j}$ , we have

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

and

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

which is precisely the general rule from Equation (4.20) for updating unit weights in arbitrary acyclic directed graphs. Notice Equation (T4.4) in Table 4.2 is just a special case of this rule, in which  $Downstream(j) = \text{outputs}$ .

## 4.6 REMARKS ON THE BACKPROPAGATION ALGORITHM

### 4.6.1 Convergence and Local Minima

As shown above, the BACKPROPAGATION algorithm implements a gradient descent search through the space of possible network weights, iteratively reducing the error  $E$  between the training example target values and the network outputs. Because the error surface for multilayer networks may contain many different local minima, gradient descent can become trapped in any of these. As a result, BACKPROPAGATION over multilayer networks is only guaranteed to converge toward some local minimum in  $E$  and not necessarily to the global minimum error.

Despite the lack of assured convergence to the global minimum error, BACKPROPAGATION is a highly effective function approximation method in practice. In many practical applications the problem of local minima has not been found to be as severe as one might fear. To develop some intuition here, consider that networks with large numbers of weights correspond to error surfaces in very high dimensional spaces (one dimension per weight). When gradient descent falls into a local minimum with respect to one of these weights, it will not necessarily be in a local minimum with respect to the other weights. In fact, the more weights in the network, the more dimensions that might provide “escape routes” for gradient descent to fall away from the local minimum with respect to this single weight.

A second perspective on local minima can be gained by considering the manner in which network weights evolve as the number of training iterations increases. Notice that if network weights are initialized to values near zero, then during early gradient descent steps the network will represent a very smooth function that is approximately linear in its inputs. This is because the sigmoid threshold function itself is approximately linear when the weights are close to zero (see the plot of the sigmoid function in Figure 4.6). Only after the weights have had time to grow will they reach a point where they can represent highly nonlinear network functions. One might expect more local minima to exist in the region of the weight space that represents these more complex functions. One hopes that by the time the weights reach this point they have already moved close enough to the global minimum that even local minima in this region are acceptable.

Despite the above comments, gradient descent over the complex error surfaces represented by ANNs is still poorly understood, and no methods are known to predict with certainty when local minima will cause difficulties. Common heuristics to attempt to alleviate the problem of local minima include:

- Add a momentum term to the weight-update rule as described in Equation (4.18). Momentum can sometimes carry the gradient descent procedure through narrow local minima (though in principle it can also carry it through narrow global minima into other local minima!).
- Use stochastic gradient descent rather than true gradient descent. As discussed in Section 4.4.3.3, the stochastic approximation to gradient descent effectively descends a different error surface for each training example, re-

lying on the average of these to approximate the gradient with respect to the full training set. These different error surfaces typically will have local minima, making it less likely that the process will get stuck in any one of them.

- Train multiple networks using the same data, but initializing each with different random weights. If the different training efforts lead to different local minima, then the network with the best performance on a separate validation data set can be selected. Alternatively, all networks can be retained and treated as a “committee” of networks whose output is the (possibly weighted) average of the individual network outputs.

### 4.6.2 Representational Power of Feedforward Networks

What set of functions can be represented by feedforward networks? The answer depends on the width and depth of the networks. Although still unknown about which function classes can be described by which networks, three quite general results are known:

- *Boolean functions.* Every boolean function can be represented exactly by some network with two layers of units, although the number of hidden units required grows exponentially in the worst case with the number of input units. To see how this can be done, consider the following general method for representing an arbitrary boolean function: For each possible input pattern, create a distinct hidden unit and set its weights so that it activates if and only if this specific vector is input to the network. This produces a hidden layer that will always have exactly one unit active. Now implement the output layer as an OR gate that activates just for the desired input patterns.
- *Continuous functions.* Every bounded continuous function can be approximated with arbitrarily small error (under a finite norm) by a network with two layers of units (Cybenko 1989; Hornik et al. 1989). The theorem in this case applies to networks that use sigmoid units at the hidden layer and (unthresholded) linear units at the output layer. The number of hidden units required depends on the function to be approximated.
- *Arbitrary functions.* Any function can be approximated to arbitrary accuracy by a network with three layers of units (Cybenko 1988). Again, the first layer uses linear units, the two hidden layers use sigmoid units, and the number of units required at each layer is not known in general. The proof of this involves showing that any function can be approximated by a linear combination of many localized functions that have value 0 everywhere except for some small region, and then showing that two layers of sigmoid units are sufficient to produce good local approximations.

These results show that limited depth feedforward networks provide a highly expressive hypothesis space for BACKPROPAGATION. However, it is impo-

keep in mind that the network weight vectors reachable by gradient descent from the initial weight values may not include all possible weight vectors. Hertz et al. (1991) provide a more detailed discussion of the above results.

### 4.6.3 Hypothesis Space Search and Inductive Bias

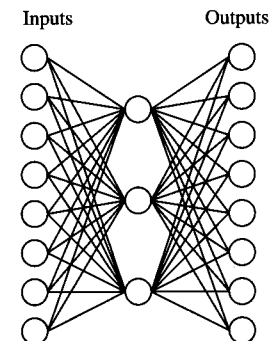
It is interesting to compare the hypothesis space search of BACKPROPAGATION to the search performed by other learning algorithms. For BACKPROPAGATION, every possible assignment of network weights represents a syntactically distinct hypothesis that in principle can be considered by the learner. In other words, the hypothesis space is the  $n$ -dimensional Euclidean space of the  $n$  network weights. Notice this hypothesis space is *continuous*, in contrast to the hypothesis spaces of decision tree learning and other methods based on discrete representations. The fact that it is continuous, together with the fact that  $E$  is differentiable with respect to the continuous parameters of the hypothesis, results in a well-defined error gradient that provides a very useful structure for organizing the search for the best hypothesis. This structure is quite different from the general-to-specific ordering used to organize the search for symbolic concept learning algorithms, or the simple-to-complex ordering over decision trees used by the ID3 and C4.5 algorithms.

What is the inductive bias by which BACKPROPAGATION generalizes beyond the observed data? It is difficult to characterize precisely the inductive bias of BACKPROPAGATION learning, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as *smooth interpolation between data points*. Given two positive training examples with no negative examples between them, BACKPROPAGATION will tend to label points in between as positive examples as well. This can be seen, for example, in the decision surface illustrated in Figure 4.5, in which the specific sample of training examples gives rise to smoothly varying decision regions.

### 4.6.4 Hidden Layer Representations

One intriguing property of BACKPROPAGATION is its ability to discover useful intermediate representations at the hidden unit layers inside the network. Because training examples constrain only the network inputs and outputs, the weight-tuning procedure is free to set weights that define whatever hidden unit representation is most effective at minimizing the squared error  $E$ . This can lead BACKPROPAGATION to define new hidden layer features that are not explicit in the input representation, but which capture properties of the input instances that are most relevant to learning the target function.

Consider, for example, the network shown in Figure 4.7. Here, the eight network inputs are connected to three hidden units, which are in turn connected to the eight output units. Because of this structure, the three hidden units will be forced to re-represent the eight input values in some way that captures their



Input	Hidden Values	Output
10000000	→ .89 .04 .08	→ 10
01000000	→ .15 .99 .99	→ 01
00100000	→ .01 .97 .27	→ 00
00010000	→ .99 .97 .71	→ 00
00001000	→ .03 .05 .02	→ 00
00000100	→ .01 .11 .88	→ 00
00000010	→ .80 .01 .98	→ 00
00000001	→ .60 .94 .01	→ 00

FIGURE 4.7

Learned Hidden Layer Representation. This  $8 \times 3 \times 8$  network was trained to learn the target function, using the eight training examples shown. After 5000 training epochs, the three hidden unit values encode the eight distinct inputs using the encoding shown on the right. Notice if the hidden unit values are rounded to zero or one, the result is the standard binary encoding for eight distinct inputs.

relevant features, so that this hidden layer representation can be used by the output units to compute the correct target values.

Consider training the network shown in Figure 4.7 to learn the target function  $f(\vec{x}) = \vec{x}$ , where  $\vec{x}$  is a vector containing seven 0's and a single 1. The network must learn to reproduce the eight inputs at the corresponding eight output units. Although this is a simple function, the network in this case is constrained to use only three hidden units. Therefore, the essential information from the eight input units must be captured by the three learned hidden units.

When BACKPROPAGATION is applied to this task, using each of the eight possible vectors as training examples, it successfully learns the target function. The hidden layer representation is created by the gradient descent BACKPROPAGATION algorithm? By examining the hidden unit values generated by the learned network for each of the eight possible input vectors, it is easy to see that the learned hidden layer encoding is similar to the familiar standard binary encoding of eight values using three bits (e.g., 000, 001, 010, ..., 111). The exact values of the hidden unit values in one typical run of BACKPROPAGATION are shown in Figure 4.7.

This ability of multilayer networks to automatically discover useful intermediate representations at the hidden layers is a key feature of ANN learning. In contrast to other learning methods that are constrained to use only predefined features provided by the human designer, this provides an important degree of flexibility that allows the learner to invent features not explicitly introduced by the human designer. Of course these invented features must still be computable as sigmoid unit outputs of the provided network inputs. Note when more layers of units are used in the network, more complex features can be invented. Another example of hidden layer features is provided in the face recognition application discussed in Section 4.7.

In order to develop a better intuition for the operation of BACKPROPAGATION, in this example, let us examine the operation of the gradient descent procedure

greater detail<sup>†</sup>. The network in Figure 4.7 was trained using the algorithm shown in Table 4.2, with initial weights set to random values in the interval  $(-0.1, 0.1)$ , learning rate  $\eta = 0.3$ , and no weight momentum (i.e.,  $\alpha = 0$ ). Similar results were obtained by using other learning rates and by including nonzero momentum. The hidden unit encoding shown in Figure 4.7 was obtained after 5000 training iterations through the outer loop of the algorithm (i.e., 5000 iterations through each of the eight training examples). Most of the interesting weight changes occurred, however, during the first 2500 iterations.

We can directly observe the effect of BACKPROPAGATION's gradient descent search by plotting the squared output error as a function of the number of gradient descent search steps. This is shown in the top plot of Figure 4.8. Each line in this plot shows the squared output error summed over all training examples, for one of the eight network outputs. The horizontal axis indicates the number of iterations through the outermost loop of the BACKPROPAGATION algorithm. As this plot indicates, the sum of squared errors for each output decreases as the gradient descent procedure proceeds, more quickly for some output units and less quickly for others.

The evolution of the hidden layer representation can be seen in the second plot of Figure 4.8. This plot shows the three hidden unit values computed by the learned network for one of the possible inputs (in particular, 01000000). Again, the horizontal axis indicates the number of training iterations. As this plot indicates, the network passes through a number of different encodings before converging to the final encoding given in Figure 4.7.

Finally, the evolution of individual weights within the network is illustrated in the third plot of Figure 4.8. This plot displays the evolution of weights connecting the eight input units (and the constant 1 bias input) to one of the three hidden units. Notice that significant changes in the weight values for this hidden unit coincide with significant changes in the hidden layer encoding and output squared errors. The weight that converges to a value near zero in this case is the bias weight  $w_0$ .

#### 4.6.5 Generalization, Overfitting, and Stopping Criterion

In the description of the BACKPROPAGATION algorithm in Table 4.2, the termination condition for the algorithm has been left unspecified. What is an appropriate condition for terminating the weight update loop? One obvious choice is to continue training until the error  $E$  on the training examples falls below some predetermined threshold. In fact, this is a poor strategy because BACKPROPAGATION is susceptible to overfitting the training examples at the cost of decreasing generalization accuracy over other unseen examples.

To see the dangers of minimizing the error over the training data, consider how the error  $E$  varies with the number of weight iterations. Figure 4.9 shows

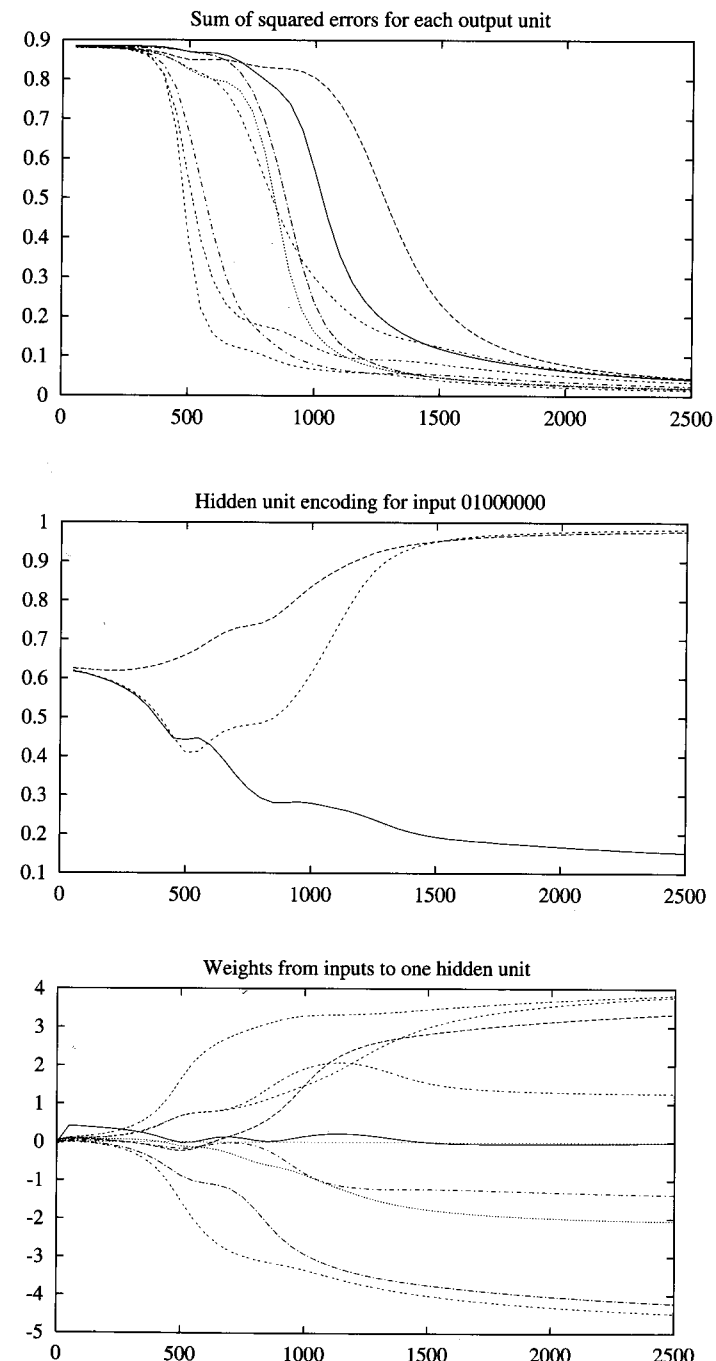


FIGURE 4.8

Learning the  $8 \times 3 \times 8$  Network. The top plot shows the evolving sum of squared errors for the eight output units, as the number of training iterations (epochs) increases. The middle plot shows the evolving hidden layer representation for the input string "01000000." The bottom plot shows the evolving weights for one of the three hidden units.

<sup>†</sup>The source code to reproduce this example is available at <http://www.cs.cmu.edu/~tom/mlbook.html>.

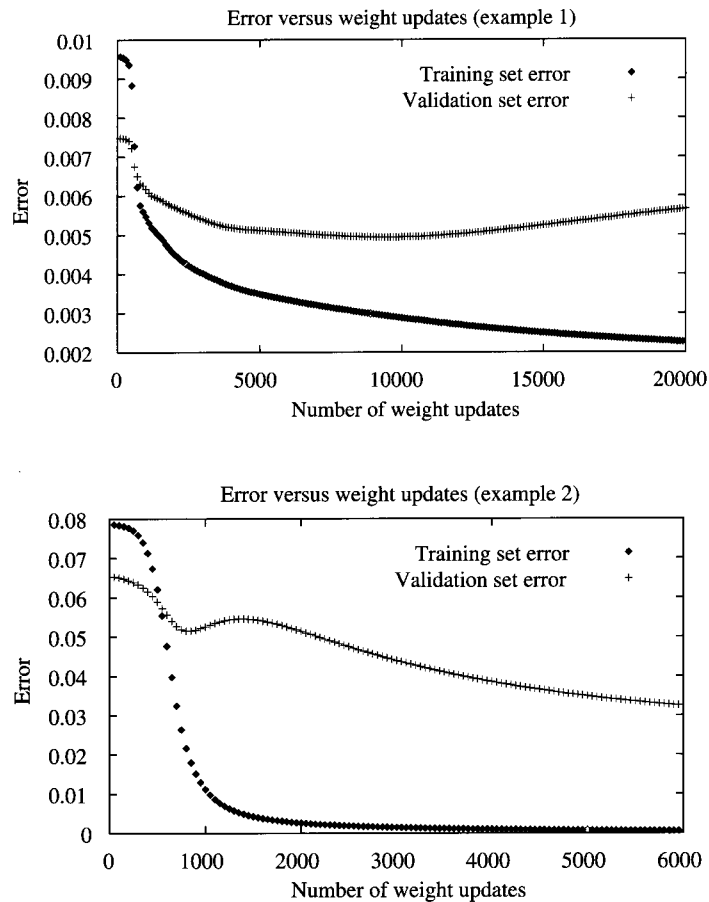


FIGURE 4.9

Plots of error  $E$  as a function of the number of weight updates, for two different robot perception tasks. In both learning cases, error  $E$  over the training examples decreases monotonically, as gradient descent minimizes this measure of error. Error over the separate “validation” set of examples typically decreases at first, then may later increase due to overfitting the training examples. The network most likely to generalize correctly to unseen data is the network with the lowest error over the validation set. Notice in the second plot, one must be careful to not stop training too soon when the validation set error begins to increase.

this variation for two fairly typical applications of BACKPROPAGATION. Consider first the top plot in this figure. The lower of the two lines shows the monotonically decreasing error  $E$  over the training set, as the number of gradient descent iterations grows. The upper line shows the error  $E$  measured over a different *validation set* of examples, distinct from the training examples. This line measures the *generalization accuracy* of the network—the accuracy with which it fits examples beyond the training data.

Notice the generalization accuracy measured over the validation set first decreases, then increases, even as the error over the training examples continues to decrease. How can this occur? This occurs because the weights are tuned to fit idiosyncrasies of the training examples that are not representative of the general distribution of examples. The large number of weight parameters in ANNs provides many degrees of freedom for fitting such idiosyncrasies.

Why does overfitting tend to occur during later iterations, but not during earlier iterations? Consider that network weights are initialized to small random values. With weights of nearly identical value, only very smooth decision surfaces are describable. As training proceeds, some weights begin to grow in order to reduce the error over the training data, and the complexity of the learned decision surface increases. Thus, the effective complexity of the hypotheses that can be represented by BACKPROPAGATION increases with the number of weight-tuning iterations. After enough weight-tuning iterations, BACKPROPAGATION will often be able to learn overly complex decision surfaces that fit noise in the training data or unique characteristics of the particular training sample. This overfitting problem is analogous to the overfitting problem in decision tree learning (see Chapter 3).

Several techniques are available to address the overfitting problem in BACKPROPAGATION learning. One approach, known as *weight decay*, is to decrease the weight by some small factor during each iteration. This is equivalent to modifying the definition of  $E$  to include a penalty term corresponding to the total magnitude of the network weights. The motivation for this approach is to keep weights small, to bias learning against complex decision surfaces.

One of the most successful methods for overcoming the overfitting problem is to simply provide a set of validation data to the algorithm in addition to the training data. The algorithm monitors the error with respect to this validation set while using the training set to drive the gradient descent search. In essence, this allows the algorithm itself to plot the two curves shown in Figure 4.9. How many weight-tuning iterations should the algorithm perform? Clearly, it should perform the number of iterations that produces the lowest error *over the validation set*; this is the best indicator of network performance over unseen examples. In most implementations of this approach, two copies of the network weights are maintained: one copy for training and a separate copy of the best-performing weights. The training is terminated when the validation set error is significantly higher than the error of the stored weights. Once the training is terminated and the stored weights are returned as the final hypothesis, this procedure is applied in the case of the top plot of Figure 4.9, it would return the network weights obtained after 9100 iterations. The second plot in Figure 4.9 shows that it is not always obvious when the lowest error on the validation set has been reached. In this plot, the validation set error decreases, then increases, then decreases again. Care must be taken to avoid the mistaken conclusion that the network has reached its lowest validation set error at iteration 850.

In general, the issue of overfitting and how to overcome it is a subtle one. The above cross-validation approach works best when extra data are available to provide a validation set. Unfortunately, however, the problem of overfitting

severe for small training sets. In these cases, a  $k$ -fold cross-validation approach is sometimes used, in which cross validation is performed  $k$  different times, each time using a different partitioning of the data into training and validation sets, and the results are then averaged. In one version of this approach, the  $m$  available examples are partitioned into  $k$  disjoint subsets, each of size  $m/k$ . The cross-validation procedure is then run  $k$  times, each time using a different one of these subsets as the validation set and combining the other subsets for the training set. Thus, each example is used in the validation set for one of the experiments and in the training set for the other  $k - 1$  experiments. On each experiment the above cross-validation approach is used to determine the number of iterations  $i$  that yield the best performance on the validation set. The mean  $\bar{i}$  of these estimates for  $i$  is then calculated, and a final run of BACKPROPAGATION is performed *training on all  $n$  examples* for  $\bar{i}$  iterations, with no validation set. This procedure is closely related to the procedure for comparing two learning methods based on limited data, described in Chapter 5.

#### 4.7 AN ILLUSTRATIVE EXAMPLE: FACE RECOGNITION

To illustrate some of the practical design choices involved in applying BACKPROPAGATION, this section discusses applying it to a learning task involving face recognition. All image data and code used to produce the examples described in this section are available at World Wide Web site <http://www.cs.cmu.edu/~tom/mlbook.html>, along with complete documentation on how to use the code. Why not try it yourself?

##### 4.7.1 The Task

The learning task here involves classifying camera images of faces of various people in various poses. Images of 20 different people were collected, including approximately 32 images per person, varying the person's expression (happy, sad, angry, neutral), the direction in which they were looking (left, right, straight ahead, up), and whether or not they were wearing sunglasses. As can be seen from the example images in Figure 4.10, there is also variation in the background behind the person, the clothing worn by the person, and the position of the person's face within the image. In total, 624 greyscale images were collected, each with a resolution of  $120 \times 128$ , with each image pixel described by a greyscale intensity value between 0 (black) and 255 (white).

A variety of target functions can be learned from this image data. For example, given an image as input we could train an ANN to output the identity of the person, the direction in which the person is facing, the gender of the person, whether or not they are wearing sunglasses, etc. All of these target functions can be learned to high accuracy from this image data, and the reader is encouraged to try out these experiments. In the remainder of this section we consider one particular task: learning the direction in which the person is facing (to their left, right, straight ahead, or upward).

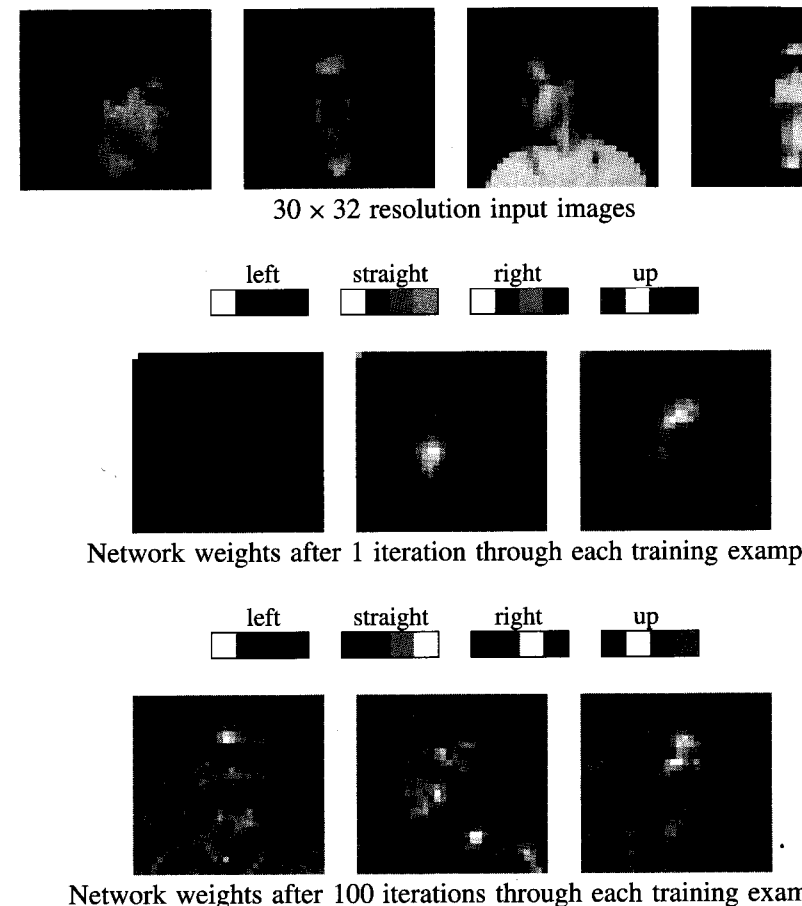


FIGURE 4.10

Learning an artificial neural network to recognize face pose. Here a  $960 \times 3 \times 4$  network on grey-level images of faces (see top), to predict whether a person is looking to the left, right, straight ahead, or up. After training on 260 such images, the network achieves an accuracy of 95% on a separate test set. The learned network weights are shown after one weight-tuning iteration through the training examples and after 100 iterations. Each output unit (left, straight, right, up) has weights, shown by dark (negative) and light (positive) blocks. The leftmost block corresponds to the weight  $w_0$ , which determines the unit threshold, and the three blocks to the right correspond to the weights on inputs from the three hidden units. The weights from the image pixels to each output unit are also shown, with each weight plotted in the position of the corresponding image pixel.

##### 4.7.2 Design Choices

In applying BACKPROPAGATION to any given task, a number of design choices must be made. We summarize these choices below for our task of learning the direction in which a person is facing. Although no attempt was made to determine the precise optimal design choices for this task, the design described here

the target function quite well. After training on a set of 260 images, classification accuracy over a separate test set is 90%. In contrast, the default accuracy achieved by randomly guessing one of the four possible face directions is 25%.

**Input encoding.** Given that the ANN input is to be some representation of the image, one key design choice is how to encode this image. For example, we could preprocess the image to extract edges, regions of uniform intensity, or other local image features, then input these features to the network. One difficulty with this design option is that it would lead to a variable number of features (e.g., edges) per image, whereas the ANN has a fixed number of input units. The design option chosen in this case was instead to encode the image as a fixed set of  $30 \times 32$  pixel intensity values, with one network input per pixel. The pixel intensity values ranging from 0 to 255 were linearly scaled to range from 0 to 1 so that network inputs would have values in the same interval as the hidden unit and output unit activations. The  $30 \times 32$  pixel image is, in fact, a coarse resolution summary of the original  $120 \times 128$  captured image, with each coarse pixel intensity calculated as the mean of the corresponding high-resolution pixel intensities. Using this coarse-resolution image reduces the number of inputs and network weights to a much more manageable size, thereby reducing computational demands, while maintaining sufficient resolution to correctly classify the images. Recall from Figure 4.1 that the ALVINN system uses a similar coarse-resolution image as input to the network. One interesting difference is that in ALVINN, each coarse resolution pixel intensity is obtained by selecting the intensity of a single pixel at random from the appropriate region within the high-resolution image, rather than taking the mean of all pixel intensities within this region. The motivation for this in ALVINN is that it significantly reduces the computation required to produce the coarse-resolution image from the available high-resolution image. This efficiency is especially important when the network must be used to process many images per second while autonomously driving the vehicle.

**Output encoding.** The ANN must output one of four values indicating the direction in which the person is looking (left, right, up, or straight). Note we could encode this four-way classification using a single output unit, assigning outputs of, say, 0.2, 0.4, 0.6, and 0.8 to encode these four possible values. Instead, we use four distinct output units, each representing one of the four possible face directions, with the highest-valued output taken as the network prediction. This is often called a *1-of- $n$*  output encoding. There are two motivations for choosing the *1-of- $n$*  output encoding over the single unit option. First, it provides more degrees of freedom to the network for representing the target function (i.e., there are  $n$  times as many weights available in the output layer of units). Second, in the *1-of- $n$*  encoding the difference between the highest-valued output and the second-highest can be used as a measure of the confidence in the network prediction (ambiguous classifications may result in near or exact ties). A further design choice here is “what should be the target values for these four output units?” One obvious choice would be to use the four target values (1, 0, 0, 0) to encode a face looking to the

left, (0, 1, 0, 0) to encode a face looking straight, etc. Instead of 0 and we use values of 0.1 and 0.9, so that (0.9, 0.1, 0.1, 0.1) is the target output for a face looking to the left. The reason for avoiding target values of 0 and 1 is that sigmoid units cannot produce these output values given finite weights. If we attempt to train the network to fit target values of exactly 0 and 1, gradient descent will force the weights to grow without bound. On the other hand, target values of 0.1 and 0.9 are achievable using a sigmoid unit with finite weights.

**Network graph structure.** As described earlier, BACKPROPAGATION can be applied to any acyclic directed graph of sigmoid units. Therefore, another design choice we face is how many units to include in the network and how to connect them. The most common network structure is a layered network with feedforward connections from every unit in one layer to every unit in the next. In the current design we chose this standard structure, using two layers of sigmoid units (one hidden layer and one output layer). It is common to use two layers of sigmoid units and, occasionally, three layers. It is not common to use more layers than this because training times become very long and networks with three layers of sigmoid units can already express a rich variety of target functions (see Section 4.6.2). Given our choice of a layered feedforward network with one hidden layer, how many hidden units should we include? The results reported in Figure 4.10, only three hidden units were used, yielding a test set accuracy of 90%. In other experiments 30 hidden units were used, yielding a test set accuracy one to two percent higher. Although the generalization accuracy varied only a small amount between these two experiments, the second experiment required significantly more training time. Using 260 training images, the training time was approximately 1 hour on a Sun Sparc5 workstation for the 30 hidden unit network, compared to approximately 5 minutes for the 3 hidden unit network. In many applications it has been found that some minimum number of hidden units is required in order to learn the target function accurately and that extra hidden units above this number do not dramatically affect generalization accuracy. Provided cross-validation methods are used to determine how many gradient descent iterations should be performed. If such methods are not used, then increasing the number of hidden units often increases the tendency to overfit the training set, thereby reducing generalization accuracy.

**Other learning algorithm parameters.** In these learning experiments the learning rate  $\eta$  was set to 0.3, and the momentum  $\alpha$  was set to 0.3. Lower values of these parameters produced roughly equivalent generalization accuracy, but longer training times. If these values are set too high, training fails to converge to a minimum with acceptable error over the training set. Full gradient descent was used in these experiments (in contrast to the stochastic approximation to gradient descent in the algorithm of Table 4.2). Network weights in the output units were initialized to small random values. However, input unit weights were initialized to zero because this yields much more intelligible visualizations of the learned weights (see Figure 4.10), without any noticeable impact on generalization accuracy.

number of training iterations was selected by partitioning the available data into a training set and a separate validation set. Gradient descent was used to minimize the error over the training set, and after every 50 gradient descent steps the performance of the network was evaluated over the validation set. The final selected network was the one with the highest accuracy over the validation set. See Section 4.6.5 for an explanation and justification of this procedure. The final reported accuracy (e.g., 90% for the network in Figure 4.10) was measured over yet a third set of test examples that were not used in any way to influence training.

### 4.7.3 Learned Hidden Representations

It is interesting to examine the learned weight values for the 2899 weights in the network. Figure 4.10 depicts the values of each of these weights after one iteration through the weight update for all training examples, and again after 100 iterations.

To understand this diagram, consider first the four rectangular blocks just below the face images in the figure. Each of these rectangles depicts the weights for one of the four output units in the network (encoding left, straight, right, and up). The four squares within each rectangle indicate the four weights associated with this output unit—the weight  $w_0$ , which determines the unit threshold (on the left), followed by the three weights connecting the three hidden units to this output. The brightness of the square indicates the weight value, with bright white indicating a large positive weight, dark black indicating a large negative weight, and intermediate shades of grey indicating intermediate weight values. For example, the output unit labeled “up” has a near zero  $w_0$  threshold weight, a large positive weight from the first hidden unit, and a large negative weight from the second hidden unit.

The weights of the hidden units are shown directly below those for the output units. Recall that each hidden unit receives an input from each of the  $30 \times 32$  image pixels. The  $30 \times 32$  weights associated with these inputs are displayed so that each weight is in the position of the corresponding image pixel (with the  $w_0$  threshold weight superimposed in the top left of the array). Interestingly, one can see that the weights have taken on values that are especially sensitive to features in the region of the image in which the face and body typically appear.

The values of the network weights after 100 gradient descent iterations through each training example are shown at the bottom of the figure. Notice the leftmost hidden unit has very different weights than it had after the first iteration, and the other two hidden units have changed as well. It is possible to understand to some degree the encoding in this final set of weights. For example, consider the output unit that indicates a person is looking to his right. This unit has a strong positive weight from the second hidden unit and a strong negative weight from the third hidden unit. Examining the weights of these two hidden units, it is easy to see that if the person’s face is turned to his right (i.e., our left), then his bright skin will roughly align with strong positive weights in this hidden unit, and his dark hair will roughly align with negative weights, resulting in this unit outputting a large value. The same image will cause the third hidden unit to output a value

close to zero, as the bright face will tend to align with the large negative weights in this case.

## 4.8 ADVANCED TOPICS IN ARTIFICIAL NEURAL NETWORKS

### 4.8.1 Alternative Error Functions

As noted earlier, gradient descent can be performed for any function differentiable with respect to the parameterized hypothesis space. While the BACKPROPAGATION algorithm defines  $E$  in terms of the sum of squares of the network, other definitions have been suggested in order to incorporate other constraints into the weight-tuning rule. For each new definition of the weight-tuning rule for gradient descent must be derived. Examples of alternative definitions of  $E$  include

- Adding a penalty term for weight magnitude. As discussed above, we can add a term to  $E$  that increases with the magnitude of the weights. This causes the gradient descent search to seek weight vectors with smaller magnitudes, thereby reducing the risk of overfitting. One way to do this is to redefine  $E$  as

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

which yields a weight update rule identical to the BACKPROPAGATION algorithm except that each weight is multiplied by the constant  $(1 - 2\gamma\eta)$  on each iteration. Thus, choosing this definition of  $E$  is equivalent to using a weight decay strategy (see Exercise 4.10.)

- Adding a term for errors in the *slope*, or derivative of the target function. In some cases, training information may be available regarding the derivatives of the target function, as well as desired values. For example, Simard et al. (1992) describe an application to character recognition in which certain training derivatives are used to constrain the network to learn character recognition functions that are invariant of translation within the image. Mitchell and Thrun (1993) describe methods for calculating training derivatives based on the learner’s prior knowledge. In both of these systems (described in Chapter 12), the error function is modified to account for measuring the discrepancy between these training derivatives and the derivatives of the learned network. One example of such an error function is

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in \text{outputs}} \left[ (t_{kd} - o_{kd})^2 + \mu \sum_{j \in \text{inputs}} \left( \frac{\partial t_{kd}}{\partial x_d^j} - \frac{\partial o_{kd}}{\partial x_d^j} \right)^2 \right]$$

Here  $x_d^j$  denotes the value of the  $j$ th input unit for training example  $d$ . Thus,  $\frac{\partial t_{kd}}{\partial x_d^j}$  is the training derivative describing how the target output



$t_{kd}$  should vary with a change in the input  $x_d^j$ . Similarly,  $\frac{\partial o_{kd}}{\partial x_d^j}$  denotes the corresponding derivative of the actual learned network. The constant  $\mu$  determines the relative weight placed on fitting the training values versus the training derivatives.

- Minimizing the *cross entropy* of the network with respect to the target values. Consider learning a probabilistic function, such as predicting whether a loan applicant will pay back a loan based on attributes such as the applicant's age and bank balance. Although the training examples exhibit only boolean target values (either a 1 or 0, depending on whether this applicant paid back the loan), the underlying target function might be best modeled by outputting the *probability* that the given applicant will repay the loan, rather than attempting to output the actual 1 and 0 value for each input instance. Given such situations in which we wish for the network to output probability estimates, it can be shown that the best (i.e., maximum likelihood) probability estimates are given by the network that minimizes the cross entropy, defined as

$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log(1 - o_d)$$

Here  $o_d$  is the probability estimate output by the network for training example  $d$ , and  $t_d$  is the 1 or 0 target value for training example  $d$ . Chapter 6 discusses when and why the most probable network hypothesis is the one that minimizes this cross entropy and derives the corresponding gradient descent weight-tuning rule for sigmoid units. That chapter also describes other conditions under which the most probable hypothesis is the one that minimizes the sum of squared errors.

- Altering the effective error function can also be accomplished by weight sharing, or "tying together" weights associated with different units or inputs. The idea here is that different network weights are forced to take on identical values, usually to enforce some constraint known in advance to the human designer. For example, Waibel et al. (1989) and Lang et al. (1990) describe an application of neural networks to speech recognition, in which the network inputs are the speech frequency components at different times within a 144 millisecond time window. One assumption that can be made in this application is that the frequency components that identify a specific sound (e.g., "eee") should be independent of the exact time that the sound occurs within the 144 millisecond window. To enforce this constraint, the various units that receive input from different portions of the time window are forced to share weights. The net effect is to constrain the space of potential hypotheses, thereby reducing the risk of overfitting and improving the chances for accurately generalizing to unseen situations. Such weight sharing is typically implemented by first updating each of the shared weights separately within each unit that uses the weight, then replacing each instance of the shared weight by the mean of their values. The result of this procedure is that shared weights effectively adapt to a different error function than do the unshared weights.

## 4.8.2 Alternative Error Minimization Procedures

While gradient descent is one of the most general search methods for a hypothesis to minimize the error function, it is not always the most efficient. It is not uncommon for BACKPROPAGATION to require tens of thousands of iterations through the weight update loop when training complex networks. For this reason, a number of alternative weight optimization algorithms have been proposed and explored. To see some of the other possibilities, it is helpful to think of gradient descent as an update method as involving two decisions: choosing a direction in which to update the current weight vector and choosing a distance to move. In BACKPROPAGATION, the direction is chosen by taking the negative of the gradient, and the distance is determined by the learning rate constant  $\eta$ .

One optimization method, known as *line search*, involves a different approach to choosing the distance for the weight update. In particular, once a direction is chosen that specifies the direction of the update, the update distance is chosen by finding the minimum of the error function along this line. Notice this is not true in a very large or very small weight update, depending on the position along the line that minimizes error. A second method, that builds on the idea of line search, is called the *conjugate gradient* method. Here, a search along a line is performed to search for a minimum in the error surface. In the first step in this sequence, the direction chosen is the negative of the error gradient. On each subsequent step, a new direction is chosen so that the component of the error gradient that has just been made zero, remains zero.

While alternative error-minimization methods sometimes lead to improvements in efficiency in training the network, methods such as conjugate gradient search have no significant impact on the generalization error of the final network. The only likely impact on the final error is that different error-minimization procedures may fall into different local minima. Bishop (1996) contains a general overview of several parameter optimization methods for training networks.

## 4.8.3 Recurrent Networks

Up to this point we have considered only network topologies that can be represented by acyclic directed graphs. Recurrent networks are artificial neural networks that apply to time series data and that use outputs of network units at time  $t$  as input to other units at time  $t + 1$ . In this way, they support a form of feedback over cycles in the network. To illustrate, consider the time series prediction problem of predicting the next day's stock market average  $y(t + 1)$  based on the current day's economic indicators  $x(t)$ . Given a time series of such data, one obvious approach is to train a feedforward network to predict  $y(t + 1)$  as its output, based on input values  $x(t)$ . Such a network is shown in Figure 4.11(a).

One limitation of such a network is that the prediction of  $y(t + 1)$  is based only on  $x(t)$  and cannot capture possible dependencies of  $y(t + 1)$  on earlier values of  $x$ . This might be necessary, for example, if tomorrow's stock market average  $y(t + 1)$  depends on the difference between today's economic indicators  $x(t)$  and yesterday's values  $x(t - 1)$ . Of course we could remedy this

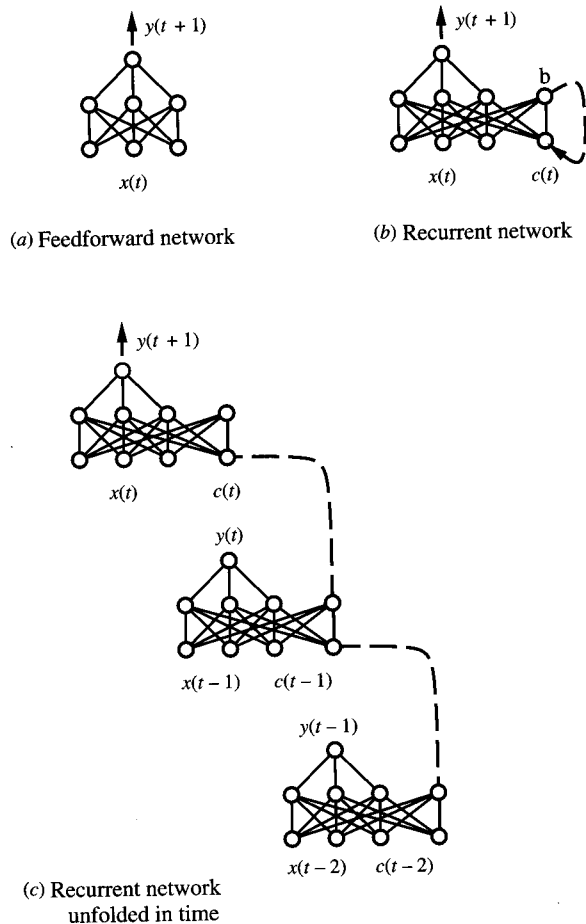


FIGURE 4.11  
Recurrent networks.

by making both  $x(t)$  and  $x(t-1)$  inputs to the feedforward network. However, if we wish the network to consider an arbitrary window of time in the past when predicting  $y(t+1)$ , then a different solution is required. The recurrent network shown in Figure 4.11(b) provides one such solution. Here, we have added a new unit  $b$  to the hidden layer, and new input unit  $c(t)$ . The value of  $c(t)$  is defined as the value of unit  $b$  at time  $t-1$ ; that is, the input value  $c(t)$  to the network at one time step is simply copied from the value of unit  $b$  on the previous time step. Notice this implements a recurrence relation, in which  $b$  represents information about the history of network inputs. Because  $b$  depends on both  $x(t)$  and on  $c(t)$ , it is possible for  $b$  to summarize information from earlier values of  $x$  that are arbitrarily distant in time. Many other network topologies also can be used to

represent recurrence relations. For example, we could have inserted several units between the input and unit  $b$ , and we could have added several units in parallel where we added the single units  $b$  and  $c$ .

How can such recurrent networks be trained? There are several recurrent networks, and several training methods have been proposed. For example, Jordan 1986; Elman 1990; Mozer 1995; Williams and Zipser 1995. Interestingly, recurrent networks such as the one shown in Figure 4.11(b) can be trained using a simple variant of BACKPROPAGATION. To understand how this works, consider the large unfolded network shown in Figure 4.11(c), which shows the data flow of the recurrent network over time. Here we have made several copies of the recurrent network, and connected them by the feedback loop by connections between the various copies. Notice that the large unfolded network contains no cycles. Therefore, the weights in the network can be trained directly using BACKPROPAGATION. Of course, if we wish to keep only one copy of the recurrent network and one set of weights, we can use the unfolded network to train the weights. Therefore, after training the unfolded network, the final weight  $w_{ji}$  in the recurrent network can be taken to be the mean value of the corresponding  $w_{ji}$  in the various copies. Mozer (1995) describes this training process in greater detail. In practice, recurrent networks are more difficult to train than networks with feedback loops and do not generalize as reliably. However, they remain popular due to their increased representational power.

#### 4.8.4 Dynamically Modifying Network Structure

Up to this point we have considered neural network learning as a process of adjusting weights within a fixed graph structure. A variety of methods have been proposed to dynamically grow or shrink the number of network units and connections in an attempt to improve generalization accuracy and training efficiency.

One idea is to begin with a network containing no hidden units, and to grow the network as needed by adding hidden units until the training error reaches some acceptable level. The CASCADE-CORRELATION algorithm (Fahlman and Lebiere 1990) is one such algorithm. CASCADE-CORRELATION begins by constructing a network with no hidden units. In the case of our face-direction learning problem, it would construct a network containing only the four output units completely connected to the  $30 \times 32$  input nodes. After this network is trained for some time, we may well find that there remains a significant residual error. This is due to the fact that the target function cannot be perfectly represented by a network with this single-layer structure. In this case, the algorithm adds a hidden unit, choosing its weight values to maximize the correlation between the hidden unit value and the residual error of the overall network. The new unit is now added into the network, with its weight values held fixed, and a new connection is added from this new unit to each output unit. The process is now repeated: the original weights are retrained (holding the hidden unit weights fixed), the error is checked, and a second hidden unit added if the residual error is above a threshold. Whenever a new hidden unit is added, its inputs include all original network inputs plus the outputs of any existing hidden units. The