this example. Notice the training rule will increase w_i in this case, because $(t-o)$, η, 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 x_i will be decreased rather than increased.

In fact, the above learning procedure can be proven to converge within 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 η is used (see Minsky and Papert 1969). If the data are not linearly separable, convergence is not assured.

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 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 training an *unthresholded* perceptron; that is, a *linear unit* for which the output o is given by

$$o(\vec{x}) = \vec{w} \cdot \vec{x} \tag{4.1}$$

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

In order to derive a weight learning rule for linear units, let us begin by specifying 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, one common measure that will turn out to be especially convenient is

$$E(\vec{w}) \equiv \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{4.2}$$

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 characterize E as a function of $\vec{w}$ because the linear unit output o depends on this weight vector. Of course E also depends on the particular set of training examples, but

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.

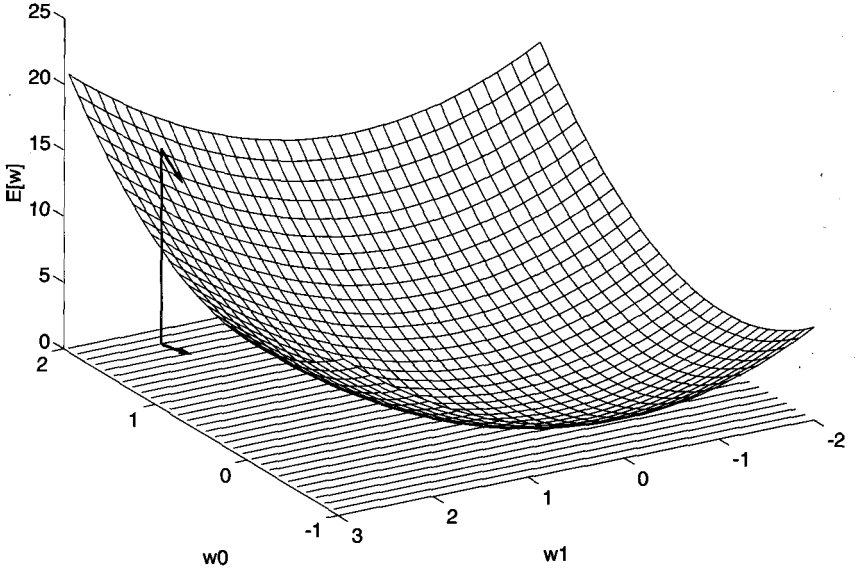

FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular 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 by starting with an arbitrary initial weight vector, then repeatedly modifying 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 $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_n}\right] \tag{4.3}$$

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 weight space, the gradient specifies the direction that produces the steepest increase in* E. The negative of this vector therefore gives the direction of steepest decrease. For example, the arrow in Figure 4.4 shows the negated gradient $-\nabla E(\vec{w})$ for a particular point in the w_0, w_1 plane.

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

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

where

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \tag{4.4}$$

Here η 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

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{4.5}$$

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 form the

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

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

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

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

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 Δw_i for each weight according to Equation (4.7). Update each weight w_i by adding Δw_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 η is used. If η 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 η 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*, η)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$, *where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *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 Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ 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

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i \tag{T4.1}$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i \tag{T4.2}$$

TABLE 4.1
GRADIENT DESCENT algorithm for training a linear unit. To implement the stochastic approximation to gradient descent, Equation (T4.2) is deleted, and Equation (T4.1) replaced by $w_i \leftarrow w_i + \eta(t - o)x_i$.

One common variation on gradient descent intended to alleviate these 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 D, the idea behind stochastic gradient descent is to approximate this gradient descent search by updating weights incrementally, following the calculation of the error 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

$$\Delta w_i = \eta(t - o)\ x_i \tag{4.10}$$

where t, o, and x_i are the target value, unit output, and ith input for the 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

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{4.11}$$

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 in 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 training examples, provides a reasonable approximation to descending the gradient with respect to our original error function $E(\vec{w})$. By making the value of η (the gradient

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

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 $o' = sgn(\vec{w} \cdot \vec{x})$ 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 ± 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 $sgn(1) = 1$, and $sgn(-1) = -1$). Even when the target values cannot be fit perfectly, the thresholded o' value will correctly fit the ± 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 Hertz 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 subtle formulation that accommodates the nonseparable case. In any case, the approach of linear programming does not scale to training multilayer networks, which is our primary concern. In contrast, the gradient descent approach, on which the delta rule is based, can be easily extended to multilayer networks, as shown 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 is 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," "had," "head," "hood," etc.). The input speech signal is represented by two numerical parameters obtained from a spectral analysis of the sound, allowing us to easily 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 nonlinear 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 a 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 previous

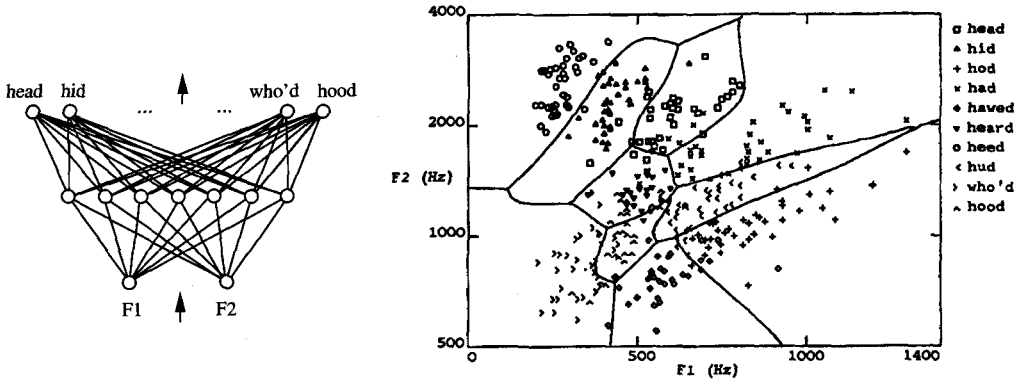

FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network 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

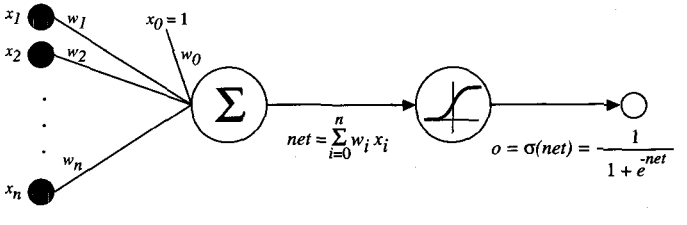

FIGURE 4.6
The sigmoid threshold unit.

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

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

where

$$\sigma(y) = \frac{1}{1 + e^{-y}} \tag{4.12}$$

σ is often called the sigmoid function or, alternatively, the logistic function. Note 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 domain 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 is easily 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 σ. For example, the term e^{-y} in the sigmoid function definition is sometimes 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 in place 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 of the network output units

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 \tag{4.13}$$

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 kth 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 our new definition of E, and the other dimensions of the space correspond now to all of 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*, η, n_{in}, n_{out}, n_{hidden})

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

η *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 $\langle \vec{x}, \vec{t} \rangle$ 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 δ_k

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

3. For each hidden unit h, calculate its error term δ_h

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

4. Update each network weight w_{ji}

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

where

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji} \tag{T4.5}$$

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, where a "node" is either an input to the network or the output of some unit in the network.
- x_{ji} denotes the input from node i to unit j, and w_{ji} denotes the corresponding weight.
- δ_n denotes the error term associated with unit n. It plays a role analogous to the quantity $(t - o)$ in our earlier discussion of the delta training rule. As 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 on this example, then updates all weights in the network. This gradient descent step is iterated (often thousands of times, using the same training examples multiple times) until the network performs acceptably well.

The gradient descent weight-update rule (Equation [T4.5] in Table 4.2) is similar to the delta training rule (Equation [4.10]). Like the delta rule, it updates each weight in proportion to the learning rate η, the input value x_{ji} to which the weight is applied, and the error in the output of the unit. The only difference is that the error $(t - o)$ in the delta rule is replaced by a more complex error term, δ_j. The exact form of δ_j follows from the derivation of the weight-tuning rule given in Section 4.5.3. To understand it intuitively, first consider how δ_k is computed for each network *output* unit k (Equation [T4.3] in the algorithm). δ_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 δ_h value for each *hidden* unit h has a similar form (Equation [T4.4] in the algorithm). However, since training examples provide target values t_k *only* for network outputs, no target values are directly available to indicate the error of hidden units' values. Instead, the error term for hidden unit h is calculated by summing the error terms δ_k for each output unit influenced by h, weighting each of the δ_k's by w_{kh}, the weight from hidden unit h to output unit k. This weight characterizes the degree to which hidden unit h is "responsible for" the error in output unit k.

The algorithm in Table 4.2 updates weights incrementally, following the presentation of each training example. This corresponds to a stochastic approximation to gradient descent. To obtain the true gradient of E one would sum 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 some threshold, or once the error on a separate validation set of examples meets some

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 nth iteration depend partially on the update that occurred during the $(n-1)$th iteration, as follows:

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

Here $\Delta w_{ji}(n)$ is the weight update performed during the nth 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 α 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 δ values. In general, the δ_r value for a unit r in layer m is computed from the δ values at the next deeper layer $m+1$ according to

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

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 δ for any internal unit (i.e., any unit that is not an output) is

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

where $Downstream(r)$ is the set of units immediately downstream from unit r in the network: that is, all units whose inputs include the output of unit r. It is this 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-tuning 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 (4.11) that stochastic gradient descent involves iterating through the training examples one at a time, for each training example d descending the gradient of the error 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 Δw_{ji}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \tag{4.21}$$

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

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

Here *outputs* is the set of output units in the network, t_k is the target value of unit k 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 denote to the jth unit of the network as follows:

- x_{ji} = the ith input to unit j
- w_{ji} = the weight associated with the ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ (the weighted sum of inputs for unit j)
- o_j = the output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = 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 weight w_{ji} can influence the rest of the network only through net_j. Therefore, we can use the

chain rule to write

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}$$
$$= \frac{\partial E_d}{\partial net_j} x_{ji} \tag{4.22}$$

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

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j} \tag{4.23}$$

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

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

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 $k = j$.

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2}(t_j - o_j)^2$$
$$= \frac{1}{2} 2(t_j - o_j) \frac{\partial (t_j - o_j)}{\partial o_j}$$
$$= -(t_j - o_j) \tag{4.24}$$

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,

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$
$$= o_j(1 - o_j) \tag{4.25}$$

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

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

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

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\ (t_j - o_j)\ o_j(1 - o_j) x_{ji} \tag{4.27}$$

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 can see now that δ_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 δ_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 for w_{ji} 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 by $Downstream(j)$. Notice that net_j can influence the network outputs (and therefore E_d) only through the units in $Downstream(j)$. Therefore, we can write

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k\ w_{kj}\ o_j(1 - o_j)
\end{aligned} \tag{4.28}$$

Rearranging terms and using δ_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 internal unit weights in arbitrary acyclic directed graphs. Notice Equation (T4.4) from Table 4.2 is just a special case of this rule, in which $Downstream(j) = outputs$.