CHAPTER 4

ARTIFICIAL NEURAL NETWORKS

Artificial neural networks (ANNs) provide a general, practical method for learning real-valued, discrete-valued, and vector-valued functions from examples. Algorithms such as BACKPROPAGATION use gradient descent to tune network parameters to best fit a training set of input-output pairs. ANN learning is robust to errors in the training data and has been successfully applied to problems such as interpreting visual scenes, speech recognition, and learning robot control strategies.

4.1 INTRODUCTION

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the BACKPROPAGATION algorithm described in this chapter has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989), learning to recognize spoken words (Lang et al. 1990), and learning to recognize faces (Cottrell 1990). One survey of practical applications is provided by Rumelhart et al. (1994).

4.1.1 Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neurobiology. The human brain, for example, is estimated to contain a densely interconnected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of 10^{-3} seconds—quite slow compared to computer switching speeds of 10^{-10} seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly. For example, it requires approximately 10^{-1} seconds to visually recognize your mother. Notice the sequence of neuron firings that can take place during this 10^{-1}-second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons. This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications.

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes.

Historically, two groups of researchers have worked with artificial neural networks. One group has been motivated by the goal of using ANNs to study and model biological learning processes. A second group has been motivated by the goal of obtaining highly effective machine learning algorithms, independent of whether these algorithms mirror biological processes. Within this book our interest fits the latter group, and therefore we will not dwell further on biological modeling. For more information on attempts to model biological systems using ANNs, see, for example, Churchland and Sejnowski (1992); Zornetzer et al. (1994); Gabriel and Moore (1990).

4.2 NEURAL NETWORK REPRESENTATIONS

A prototypical example of ANN learning is provided by Pomerleau's (1993) system ALVINN, which uses a learned ANN to steer an autonomous vehicle d[illegible]

at normal speeds on public highways. The input to the neural network is a 30×32 grid of pixel intensities obtained from a forward-pointed camera mounted on the vehicle. The network output is the direction in which the vehicle is steered. The ANN is trained to mimic the observed steering commands of a human driving the vehicle for approximately 5 minutes. ALVINN has used its learned networks to successfully drive at speeds up to 70 miles per hour and for distances of 90 miles on public highways (driving in the left lane of a divided public highway, with other vehicles present).

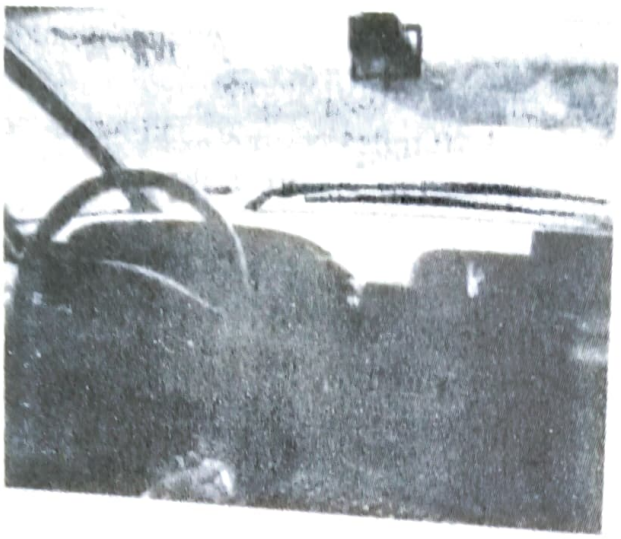
Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the network diagram corresponds to the output of a single network *unit*, and the lines entering the node from below are its inputs. As can be seen, there are four units that receive inputs directly from all of the 30×32 pixels in the image. These are called "hidden" units because their output is available only within the network and is not available as part of the global network output. Each of these four hidden units computes a single real-valued output based on a weighted combination of its 960 inputs. These hidden unit outputs are then used as inputs to a second layer of 30 "output" units. Each output unit corresponds to a particular steering direction, and the output values of these units determine which steering direction is recommended most strongly.

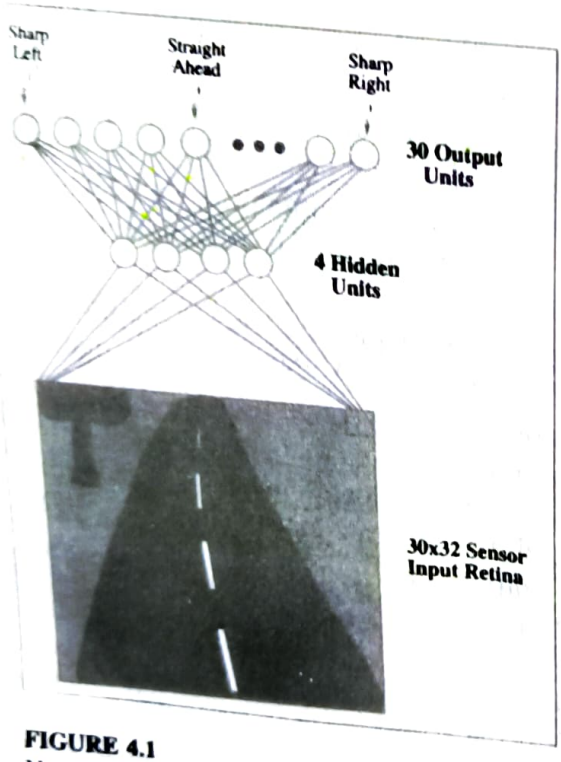
The diagrams on the right side of the figure depict the learned weight values associated with one of the four hidden units in this ANN. The large matrix of black and white boxes on the lower right depicts the weights from the 30×32 pixel inputs into the hidden unit. Here, a white box indicates a positive weight, a black box a negative weight, and the size of the box indicates the weight magnitude. The smaller rectangular diagram directly above the large matrix shows the weights from this hidden unit to each of the 30 output units.

The network structure of ALVINN is typical of many ANNs. Here the individual units are interconnected in layers that form a directed acyclic graph. In general, ANNs can be graphs with many types of structures—acyclic or cyclic, directed or undirected. This chapter will focus on the most common and practical ANN approaches, which are based on the BACKPROPAGATION algorithm. The BACKPROPAGATION algorithm assumes the network is a fixed structure that corresponds to a directed graph, possibly containing cycles. Learning corresponds to choosing a weight value for each edge in the graph. Although certain types of cycles are allowed, the vast majority of practical applications involve acyclic feed-forward networks, similar to the network structure used by ALVINN.

4.3 APPROPRIATE PROBLEMS FOR NEURAL NETWORK LEARNING

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones.





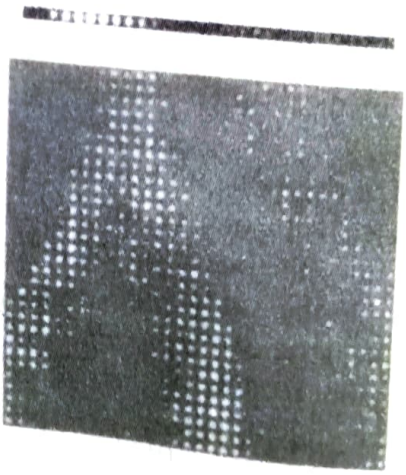


FIGURE 4.1
Neural network learning to steer an autonomous vehicle. The ALVINN system uses BACKPROPAGATION to learn to steer an autonomous vehicle (photo at top) driving at speeds up to 70 miles per hour. The diagram on the left shows how the image of a forward-mounted camera is mapped to 960 neural network inputs, which are fed forward to 4 hidden units, connected to 30 output units. Network outputs encode the commanded steering direction. The figure on the right shows weight values for one of the hidden units in this network. The 30 × 32 weights into the hidden unit are displayed in the large matrix, with white blocks indicating positive and black indicating negative weights. The weights from this hidden unit to the 30 output units are depicted by the smaller rectangular block directly above the large block. As can be seen from these output weights, activation of this particular hidden unit encourages a turn toward the left.

It is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks discussed in Chapter 3. In these cases ANN and decision tree learning often produce results of comparable accuracy. See Shavlik et al. (1991) and Weiss and Kapouleas (1989) for experimental comparisons of decision tree and ANN learning. The BACKPROPAGATION algorithm is the most commonly used ANN learning technique. It is appropriate for problems with the following characteristics:

- *Instances are represented by many attribute-value pairs.* The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values in the ALVINN example. These input attributes may be highly correlated or independent of one another. Input values can be any real values.
- *The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.* For example, in the ALVINN system the output is a vector of 30 attributes, each corresponding to a recommendation regarding the steering direction. The value of each output is some real number between 0 and 1, which in this case corresponds to the confidence in predicting the corresponding steering direction. We can also train a single network to output both the steering command and suggested acceleration, simply by concatenating the vectors that encode these two output predictions.
- *The training examples may contain errors.* ANN learning methods are quite robust to noise in the training data.
- *Long training times are acceptable.* Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- *Fast evaluation of the learned target function may be required.* Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast. For example, ALVINN applies its neural network several times per second to continually update its steering command as the vehicle drives forward.
- *The ability of humans to understand the learned target function is not important.* The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

The rest of this chapter is organized as follows: We first consider several alternative designs for the primitive units that make up artificial neural networks (perceptrons, linear units, and sigmoid units), along with learning algorithms for training single units. We then present the BACKPROPAGATION algorithm for training

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 $o(x_1, \ldots, x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

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 $w_1 x_1 + \cdots + w_n x_n$ 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

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

where

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

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

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

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.

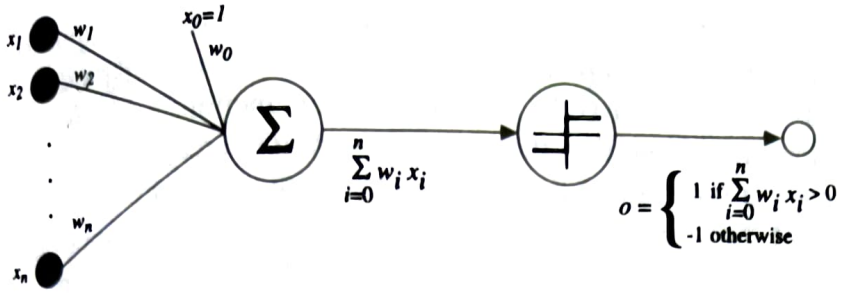


FIGURE 4.2
A perceptron.

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 (¬ AND), and NOR (¬ 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

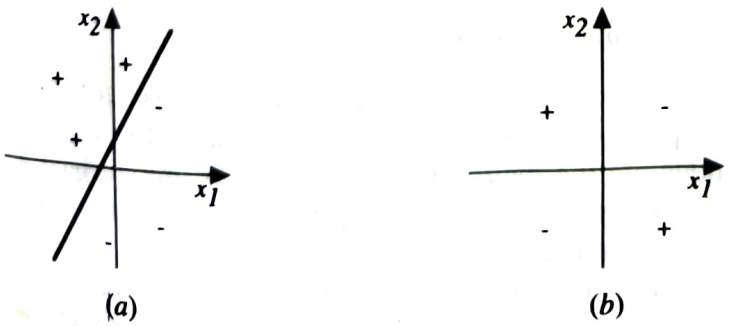


FIGURE 4.3
The decision surface represented by a two-input perceptron. (*a*) A set of training examples and the decision surface of a perceptron that classifies them correctly. (*b*) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x_1 and x_2 are the 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 ± 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 η 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 Δw_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 clas

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.

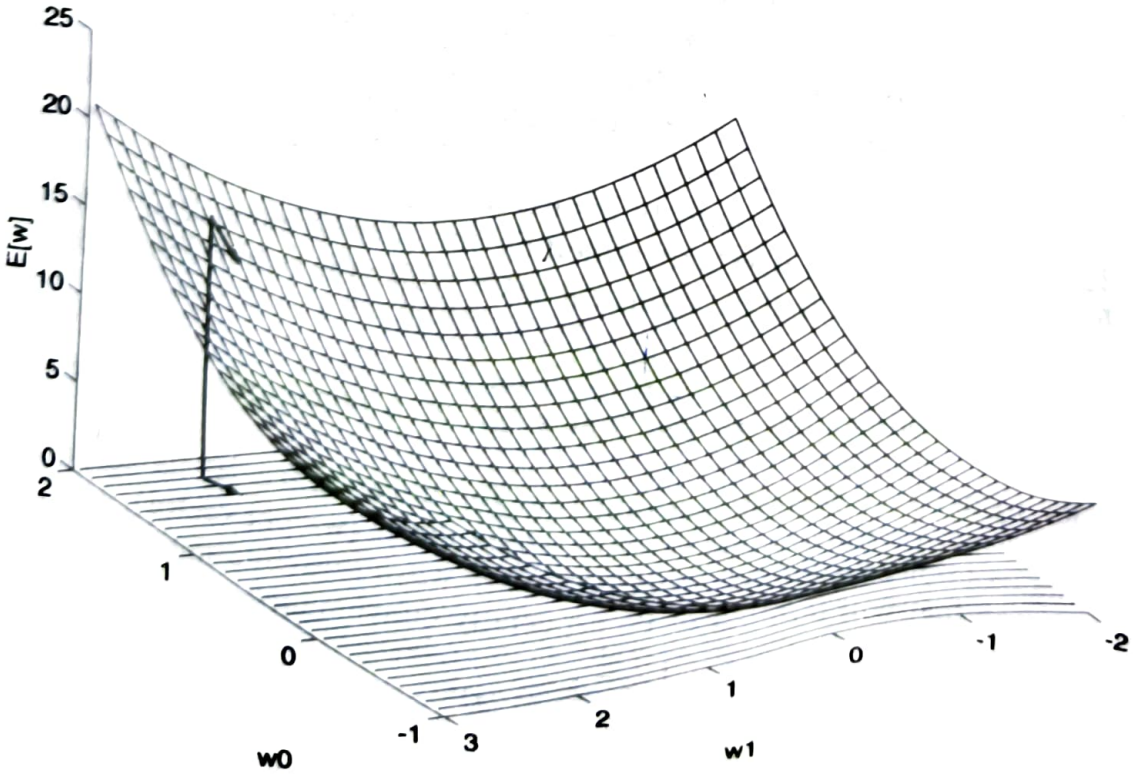


FIGURE 4.4
Error of different hypotheses. For a linear unit with two weights, the hypothesis space H is the w_0, w_1 plane. The vertical axis indicates the error of the corresponding weight vector hypothesis, relative to a fixed set of training examples. The arrow shows the negated gradient at one particular 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

$$\frac{\partial E}{\partial w_i} = \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2$$

$$= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d)$$

$$= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id}) \tag{4.6}$$

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 t_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

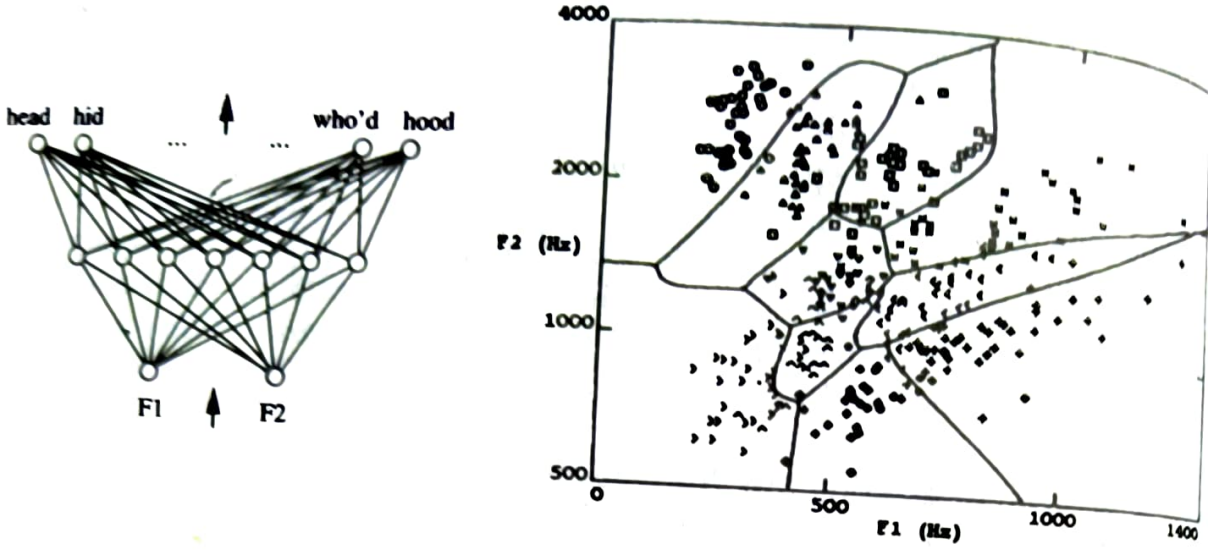


FIGURE 4.5
Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context "h_d" (e.g., "had," "hid"). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network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

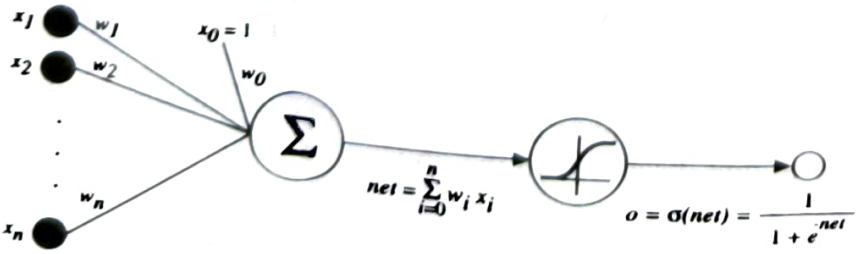


FIGURE 4.6
The sigmoid threshold unit.

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

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.*

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 $(\vec{x}, \vec{t})$ 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 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