
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 driving

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).

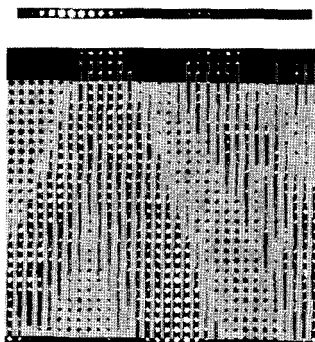
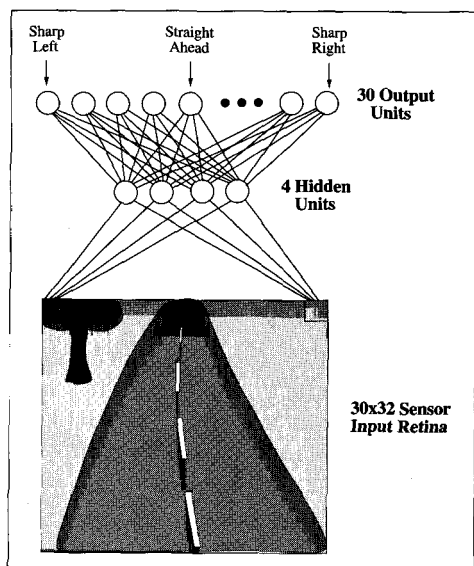
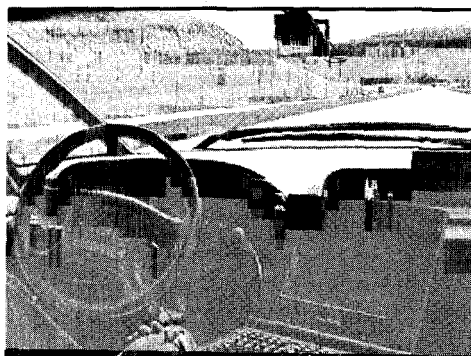
Figure 4.1 illustrates the neural network representation used in one version of the ALVINN system, and illustrates the kind of representation typical of many ANN systems. The network is shown on the left side of the figure, with the input camera image depicted below it. Each node (i.e., circle) in the 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 $o(x_1, \dots, x_n)$ 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 $\sum_{i=0}^n w_ix_i > 0$, 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.

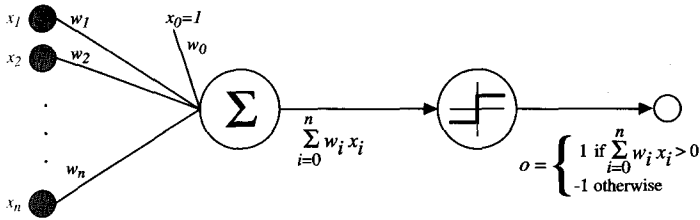


FIGURE 4.2
A perceptron.

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

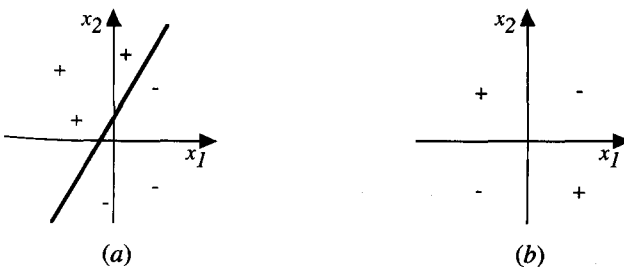


FIGURE 4.3
The decision surface represented by a two-input perceptron. (a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line).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

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

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

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

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

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.

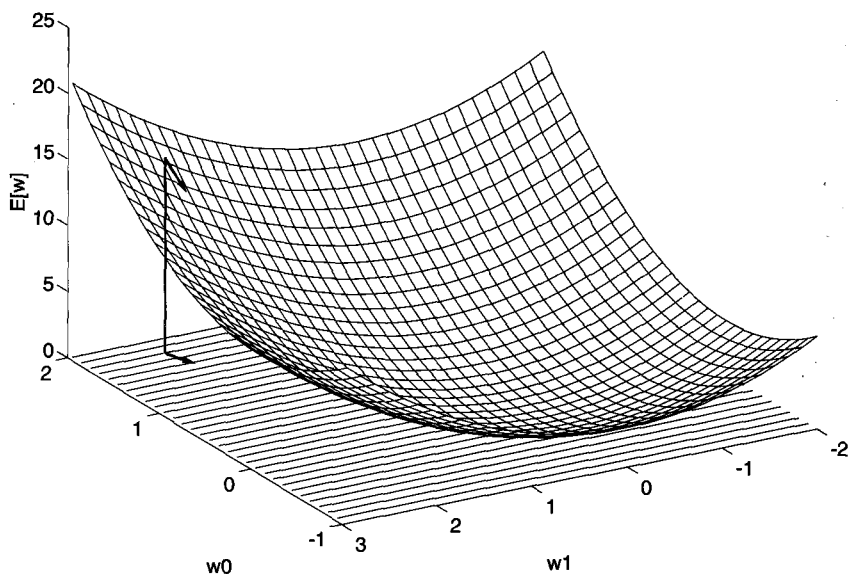


FIGURE 4.4

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 $\nabla E(\vec{w})$.

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

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

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

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

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \quad (4.5)$$

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

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

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

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

- For each linear unit weight w_i , Do

$$w_i \leftarrow w_i + \Delta w_i \quad (\text{T4.2})$$

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

$$\Delta w_i = \eta(t - o) x_i \quad (4.10)$$

where t , o , and x_i are the target value, unit output, and i th 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

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

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

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 ± 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 ± 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

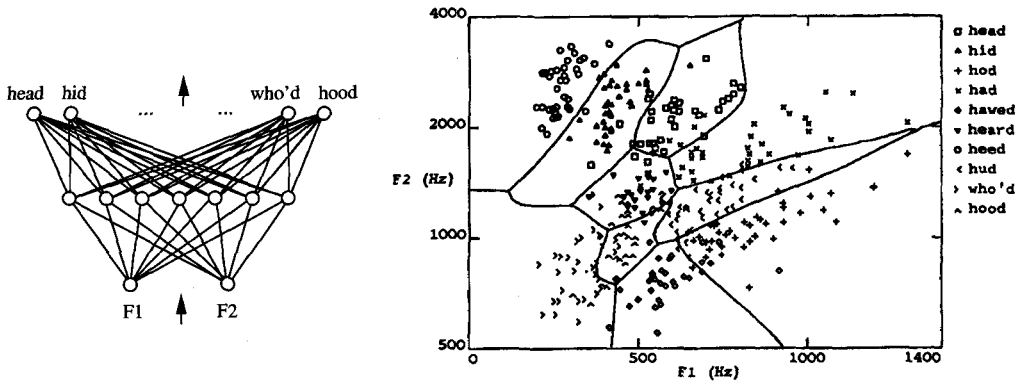


FIGURE 4.5
 Decision regions of a multilayer feedforward network. The network shown here was trained to recognize 1 of 10 vowel sounds occurring in the context “h_d” (e.g., “had,” “hid”). The network input consists of two parameters, F1 and F2, obtained from a spectral analysis of the sound. The 10 network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

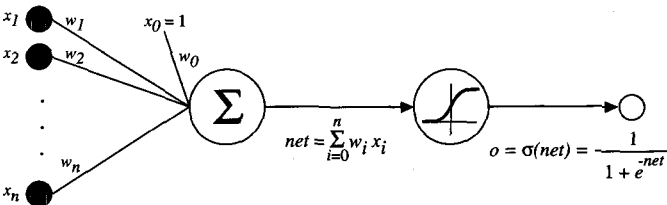


FIGURE 4.6
 The sigmoid threshold unit.

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

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

where

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

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

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

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 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 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

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

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

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

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, 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 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 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

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

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

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

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}

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} \quad (4.21)$$

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

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

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 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
- $net_j = \sum_i w_{ji}x_{ji}$ (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

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

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 stochastic gradient descent rule for output units

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

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

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

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 $Downstream(j) = outputs$.

~~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 E that is differentiable with respect to the parameterized hypothesis space. While the basic BACKPROPAGATION algorithm defines E in terms of the sum of squared errors of the network, other definitions have been suggested in order to incorporate other constraints into the weight-tuning rule. For each new definition of E a new 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 weight vector. This causes the gradient descent search to seek weight vectors with small 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 rule, except that each weight is multiplied by the constant $(1 - 2\gamma\eta)$ upon 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 desired 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 add a term measuring the discrepancy between these training derivatives and the actual 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 value

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 μ 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 finding 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 a weight-update method as involving two decisions: choosing a direction in which to alter 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 η .

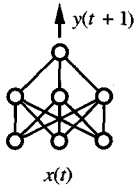
One optimization method, known as *line search*, involves a different approach to choosing the distance for the weight update. In particular, once a line is chosen that specifies the direction of the update, the update distance is chosen by finding the minimum of the error function along this line. Notice this can result in a very large or very small weight update, depending on the position of the point 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 sequence of line searches is performed to search for a minimum in the error surface. On the first step in this sequence, the direction chosen is the negative of the 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 improved efficiency in training the network, methods such as conjugate gradient tend to 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 discussion of several parameter optimization methods for training networks.

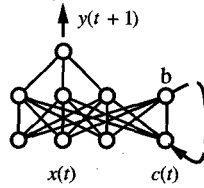
4.8.3 Recurrent Networks

Up to this point we have considered only network topologies that correspond to 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 the input to other units at time $t + 1$. In this way, they support a form of directed cycles in the network. To illustrate, consider the time series prediction task 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 the input values $x(t)$. Such a network is shown in Figure 4.11(a).

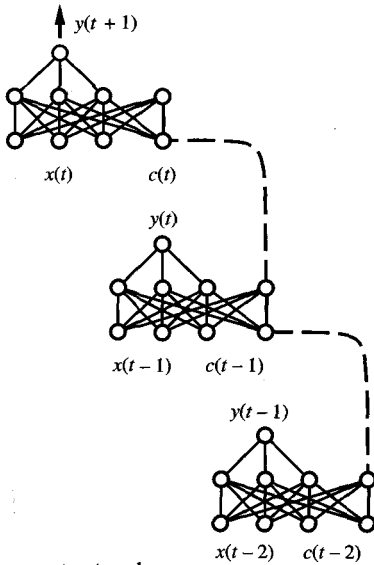
One limitation of such a network is that the prediction of $y(t + 1)$ depends only on $x(t)$ and cannot capture possible dependencies of $y(t + 1)$ on earlier values of x . This might be necessary, for example, if tomorrow's stock market average $y(t + 1)$ depends on the difference between today's economic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



(a) Feedforward network



(b) Recurrent network



(c) Recurrent network unfolded in time

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 layers of units between the input and unit b , and we could have added several context units in parallel where we added the single units b and c .

How can such recurrent networks be trained? There are several variants of recurrent networks, and several training methods have been proposed (see, 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, consider Figure 4.11(c), which shows the data flow of the recurrent network “unfolded” in time. Here we have made several copies of the recurrent network, replacing the feedback loop by connections between the various copies. Notice that this large unfolded network contains no cycles. Therefore, the weights in the unfolded network can be trained directly using BACKPROPAGATION. Of course in practice we wish to keep only one copy of the recurrent network and one set of 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} weights 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 no feedback loops and do not generalize as reliably. However, they remain important 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 problem 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 interconnections in an attempt to improve generalization accuracy and training efficiency.

One idea is to begin with a network containing no hidden units, then grow the network as needed by adding hidden units until the training error is reduced to 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 task, for example, it would construct a network containing only the four output units completely connected to the 30×32 input nodes. After this network is trained for some time, we may well find that there remains a significant residual error 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 installed into the network, with its weight values held fixed, and a new connection from this new unit is added to each output unit. The process is now repeated. The original weights are retrained (holding the hidden unit weights fixed), the residual error is checked, and a second hidden unit added if the residual error is still above threshold. Whenever a new hidden unit is added, its inputs include all of the original network inputs plus the outputs of any existing hidden units. The network is

grown in this fashion, accumulating hidden units until the network residual error is reduced to some acceptable level. Fahlman and Lebiere (1990) report cases in which CASCADE-CORRELATION significantly reduces training times, due to the fact that only a single layer of units is trained at each step. One practical difficulty is that because the algorithm can add units indefinitely, it is quite easy for it to overfit the training data, and precautions to avoid overfitting must be taken.

A second idea for dynamically altering network structure is to take the opposite approach. Instead of beginning with the simplest possible network and adding complexity, we begin with a complex network and prune it as we find that certain connections are inessential. One way to decide whether a particular weight is inessential is to see whether its value is close to zero. A second way, which appears to be more successful in practice, is to consider the effect that a small variation in the weight has on the error E . The effect on E of varying w (i.e., $\frac{\partial E}{\partial w}$) can be taken as a measure of the salience of the connection. LeCun et al. (1990) describe a process in which a network is trained, the least salient connections removed, and this process iterated until some termination condition is met. They refer to this as the “optimal brain damage” approach, because at each step the algorithm attempts to remove the least useful connections. They report that in a character recognition application this approach reduced the number of weights in a large network by a factor of 4, with a slight improvement in generalization accuracy and a significant improvement in subsequent training efficiency.

In general, techniques for dynamically modifying network structure have met with mixed success. It remains to be seen whether they can reliably improve on the generalization accuracy of BACKPROPAGATION. However, they have been shown in some cases to provide significant improvements in training times.

4.9 SUMMARY AND FURTHER READING

Main points of this chapter include:

- Artificial neural network learning provides a practical method for learning real-valued and vector-valued functions over continuous and discrete-valued attributes, in a way that is robust to noise in the training data. The BACKPROPAGATION algorithm is the most common network learning method and has been successfully applied to a variety of learning tasks, such as handwriting recognition and robot control.
- The hypothesis space considered by the BACKPROPAGATION algorithm is the space of all functions that can be represented by assigning weights to the given, fixed network of interconnected units. Feedforward networks containing three layers of units are able to approximate *any* function to arbitrary accuracy, given a sufficient (potentially very large) number of units in each layer. Even networks of practical size are capable of representing a rich space of highly nonlinear functions, making feedforward networks a good choice for learning discrete and continuous functions whose general form is unknown in advance.

- **BACKPROPAGATION** searches the space of possible hypotheses using gradient descent to iteratively reduce the error in the network fit to the training examples. Gradient descent converges to a local minimum in the training error with respect to the network weights. More generally, gradient descent is a potentially useful method for searching many continuously parameterized hypothesis spaces where the training error is a differentiable function of hypothesis parameters.
- One of the most intriguing properties of **BACKPROPAGATION** is its ability to invent new features that are not explicit in the input to the network. In particular, the internal (hidden) layers of multilayer networks learn to represent intermediate features that are useful for learning the target function and that are only implicit in the network inputs. This capability is illustrated, for example, by the ability of the $8 \times 3 \times 8$ network in Section 4.6.4 to invent the boolean encoding of digits from 1 to 8 and by the image features represented by the hidden layer in the face-recognition application of Section 4.7.
- Overfitting the training data is an important issue in ANN learning. Overfitting results in networks that generalize poorly to new data despite excellent performance over the training data. Cross-validation methods can be used to estimate an appropriate stopping point for gradient descent search and thus to minimize the risk of overfitting.
- Although **BACKPROPAGATION** is the most common ANN learning algorithm, many others have been proposed, including algorithms for more specialized tasks. For example, recurrent neural network methods train networks containing directed cycles, and algorithms such as **CASCADE CORRELATION** alter the network structure as well as the network weights.

Additional information on ANN learning can be found in several other chapters in this book. A Bayesian justification for choosing to minimize the sum of squared errors is given in Chapter 6, along with a justification for minimizing the cross-entropy instead of the sum of squared errors in other cases. Theoretical results characterizing the number of training examples needed to reliably learn boolean functions and the Vapnik-Chervonenkis dimension of certain types of networks can be found in Chapter 7. A discussion of overfitting and how to avoid it can be found in Chapter 5. Methods for using prior knowledge to improve the generalization accuracy of ANN learning are discussed in Chapter 12.

Work on artificial neural networks dates back to the very early days of computer science. McCulloch and Pitts (1943) proposed a model of a neuron that corresponds to the perceptron, and a good deal of work through the 1960s explored variations of this model. During the early 1960s Widrow and Hoff (1960) explored perceptron networks (which they called “adelines”) and the delta rule, and Rosenblatt (1962) proved the convergence of the perceptron training rule. However, by the late 1960s it became clear that single-layer perceptron networks had limited representational capabilities, and no effective algorithms were known for training multilayer networks. Minsky and Papert (1969) showed that even

simple functions such as XOR could not be represented or learned with single-layer perceptron networks, and work on ANNs receded during the 1970s.

During the mid-1980s work on ANNs experienced a resurgence, caused in large part by the invention of BACKPROPAGATION and related algorithms for training multilayer networks (Rumelhart and McClelland 1986; Parker 1985). These ideas can be traced to related earlier work (e.g., Werbos 1975). Since the 1980s, BACKPROPAGATION has become a widely used learning method, and many other ANN approaches have been actively explored. The advent of inexpensive computers during this same period has allowed experimenting with computationally intensive algorithms that could not be thoroughly explored during the 1960s.

A number of textbooks are devoted to the topic of neural network learning. An early but still useful book on parameter learning methods for pattern recognition is Duda and Hart (1973). The text by Widrow and Stearns (1985) covers perceptrons and related single-layer networks and their applications. Rumelhart and McClelland (1986) produced an edited collection of papers that helped generate the increased interest in these methods beginning in the mid-1980s. Recent books on neural network learning include Bishop (1996); Chauvin and Rumelhart (1995); Freeman and Skapina (1991); Fu (1994); Hecht-Nielsen (1990); and Hertz et al. (1991).

EXERCISES

- 4.1. What are the values of weights w_0 , w_1 , and w_2 for the perceptron whose decision surface is illustrated in Figure 4.3? Assume the surface crosses the x_1 axis at -1 , and the x_2 axis at 2.
- 4.2. Design a two-input perceptron that implements the boolean function $A \wedge \neg B$. Design a two-layer network of perceptrons that implements $A \text{ XOR } B$.
- 4.3. Consider two perceptrons defined by the threshold expression $w_0 + w_1x_1 + w_2x_2 > 0$. Perceptron A has weight values

$$w_0 = 1, \quad w_1 = 2, \quad w_2 = 1$$

and perceptron B has the weight values

$$w_0 = 0, \quad w_1 = 2, \quad w_2 = 1$$

True or false? Perceptron A is *more general than* perceptron B. (*more general than* is defined in Chapter 2.)

- 4.4. Implement the delta training rule for a two-input linear unit. Train it to fit the target concept $-2 + x_1 + 2x_2 > 0$. Plot the error E as a function of the number of training iterations. Plot the decision surface after 5, 10, 50, 100, \dots , iterations.
 - (a) Try this using various constant values for η and using a decaying learning rate of η_0/i for the i th iteration. Which works better?
 - (b) Try incremental and batch learning. Which converges more quickly? Consider both number of weight updates and total execution time.
- 4.5. Derive a gradient descent training rule for a single unit with output o , where

$$o = w_0 + w_1x_1 + w_1x_1^2 + \dots + w_nx_n + w_nx_n^2$$

- 4.6. Explain informally why the delta training rule in Equation (4.10) is only an approximation to the true gradient descent rule of Equation (4.7).
- 4.7. Consider a two-layer feedforward ANN with two inputs a and b , one hidden unit c , and one output unit d . This network has five weights (w_{ca} , w_{cb} , w_{cd} , w_{dc} , w_{d0}), where w_{x0} represents the threshold weight for unit x . Initialize these weights to the values $(.1, .1, .1, .1, .1)$, then give their values after each of the first two training iterations of the BACKPROPAGATION algorithm. Assume learning rate $\eta = .3$, momentum $\alpha = 0.9$, incremental weight updates, and the following training examples:

a	b	d
1	0	1
0	1	0

- 4.8. Revise the BACKPROPAGATION algorithm in Table 4.2 so that it operates on units using the squashing function \tanh in place of the sigmoid function. That is, assume the output of a single unit is $o = \tanh(\vec{w} \cdot \vec{x})$. Give the weight update rule for output layer weights and hidden layer weights. Hint: $\tanh'(x) = 1 - \tanh^2(x)$.
- 4.9. Recall the $8 \times 3 \times 8$ network described in Figure 4.7. Consider trying to train a $8 \times 1 \times 8$ network for the same task; that is, a network with just one hidden unit. Notice the eight training examples in Figure 4.7 could be represented by eight distinct values for the single hidden unit (e.g., 0.1, 0.2, ..., 0.8). Could a network with just one hidden unit therefore learn the identity function defined over these training examples? Hint: Consider questions such as “do there exist values for the hidden unit weights that can create the hidden unit encoding suggested above?” “do there exist values for the output unit weights that could correctly decode this encoding of the input?” and “is gradient descent likely to find such weights?”
- 4.10. Consider the alternative error function described in Section 4.8.1

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

Derive the gradient descent update rule for this definition of E . Show that it can be implemented by multiplying each weight by some constant before performing the standard gradient descent update given in Table 4.2.

- 4.11. Apply BACKPROPAGATION to the task of face recognition. See World Wide Web URL <http://www.cs.cmu.edu/~tom/book.html> for details, including face-image data, BACKPROPAGATION code, and specific tasks.
- 4.12. Consider deriving a gradient descent algorithm to learn target concepts corresponding to rectangles in the x, y plane. Describe each hypothesis by the x and y coordinates of the lower-left and upper-right corners of the rectangle — llx, lly, urx , and ury respectively. An instance $\langle x, y \rangle$ is labeled positive by hypothesis $\langle llx, lly, urx, ury \rangle$ if and only if the point $\langle x, y \rangle$ lies inside the corresponding rectangle. Define error E as in the chapter. Can you devise a gradient descent algorithm to learn such rectangle hypotheses? Notice that E is not a continuous function of llx, lly, urx , and ury , just as in the case of perceptron learning. (Hint: Consider the two solutions used for perceptrons: (1) changing the classification rule to make output predictions *continuous* functions of the inputs, and (2) defining an alternative error—such as distance to the rectangle center—as in using the delta rule to train perceptrons.) Does your algorithm converge to the minimum error hypothesis when the positive and negative examples are separable by a rectangle? When they are not? Do you

have problems with local minima? How does your algorithm compare to symbolic methods for learning conjunctions of feature constraints?

REFERENCES

- Bishop, C. M. (1996). *Neural networks for pattern recognition*. Oxford, England: Oxford University Press.
- Chauvin, Y., & Rumelhart, D. (1995). *BACKPROPAGATION: Theory, architectures, and applications* (edited collection). Hillsdale, NJ: Lawrence Erlbaum Assoc.
- Churchland, P. S., & Sejnowski, T. J. (1992). *The computational brain*. Cambridge, MA: The MIT Press.
- Cybenko, G. (1988). Continuous valued neural networks with two hidden layers are sufficient (Technical Report). Department of Computer Science, Tufts University, Medford, MA.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2, 303–314.
- Cottrell, G. W. (1990). Extracting features from faces using compression networks: Face, identity, emotion and gender recognition using holons. In D. Touretzky (Ed.), *Connection Models: Proceedings of the 1990 Summer School*. San Mateo, CA: Morgan Kaufmann.
- Dietterich, T. G., Hild, H., & Bakiri, G. (1995). A comparison of ID3 and BACKPROPAGATION for English text-to-speech mapping. *Machine Learning*, 18(1), 51–80.
- Duda, R., & Hart, P. (1973). *Pattern classification and scene analysis*. New York: John Wiley & Sons.
- Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14, 179–211.
- Fahlman, S., & Lebiere, C. (1990). *The CASCADE-CORRELATION learning architecture* (Technical Report CMU-CS-90-100). Computer Science Department, Carnegie Mellon University, Pittsburgh, PA.
- Freeman, J. A., & Skapura, D. M. (1991). *Neural networks*. Reading, MA: Addison Wesley.
- Fu, L. (1994). *Neural networks in computer intelligence*. New York: McGraw Hill.
- Gabriel, M., & Moore, J. (1990). *Learning and computational neuroscience: Foundations of adaptive networks* (edited collection). Cambridge, MA: The MIT Press.
- Hecht-Nielsen, R. (1990). *Neurocomputing*. Reading, MA: Addison Wesley.
- Hertz, J., Krogh, A., & Palmer, R.G. (1991). *Introduction to the theory of neural computation*. Reading, MA: Addison Wesley.
- Hornick, K., Stinchcombe, M., & White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2, 359–366.
- Huang, W. Y., & Lippmann, R. P. (1988). Neural net and traditional classifiers. In Anderson (Ed.), *Neural Information Processing Systems* (pp. 387–396).
- Jordan, M. (1986). Attractor dynamics and parallelism in a connectionist sequential machine. *Proceedings of the Eighth Annual Conference of the Cognitive Science Society* (pp. 531–546).
- Kohonen, T. (1984). *Self-organization and associative memory*. Berlin: Springer-Verlag.
- Lang, K. J., Waibel, A. H., & Hinton, G. E. (1990). A time-delay neural network architecture for isolated word recognition. *Neural Networks*, 3, 33–43.
- LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L.D. (1989). BACKPROPAGATION applied to handwritten zip code recognition. *Neural Computation*, 1(4).
- LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems* (Vol. 2, pp. 598–605). San Mateo, CA: Morgan Kaufmann.
- Manke, S., Finke, M., & Waibel, A. (1995). NPEN++: a writer independent, large vocabulary online cursive handwriting recognition system. *Proceedings of the International Conference on Document Analysis and Recognition*. Montreal, Canada: IEEE Computer Society.
- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 115–133.