# 19 LEARNING IN NEURAL AND BELIEF NETWORKS

*In which we see how to train complex networks of simple computing elements, thereby perhaps shedding some light on the workings of the brain.*

This chapter can be viewed in two ways. From a computational viewpoint, it is about a method of representing functions using networks of simple arithmetic computing elements, and about methods for learning such representations from examples. These networks represent functions in much the same way that circuits consisting of simple logic gates represent Boolean functions. Such representations are particularly useful for complex functions with continuous-valued outputs and large numbers of noisy inputs, where the logic-based techniques in Chapter 18 sometimes have difficulty.

From a biological viewpoint, this chapter is about a mathematical model for the operation of the brain. The simple arithmetic computing elements correspond to **neurons**—the cells that perform information processing in the brain—and the network as a whole corresponds to a collection of interconnected neurons. For this reason, the networks are called **neural networks**.[1] Besides their useful computational properties, neural networks may offer the best chance of understanding many psychological phenomena that arise from the specific structure and operation of the brain. We will therefore begin the chapter with a brief look at what is known about brains, because this provides much of the motivation for the study of neural networks. In a sense, we thereby depart from our intention, stated in Chapter 1, to concentrate on rational action rather than on imitating humans. These conflicting goals have characterized the study of neural networks ever since the very first paper on the topic by McCulloch and Pitts (1943). Methodologically speaking, the goals can be reconciled by acknowledging the fact that humans (and other animals) *do* think, and use their powers of thought to act quite successfully in complex domains where current computer-based agents would be lost. It is instructive to try to see how they do it.

Section 19.2 then presents the idealized models that are the main subject of study. Simple, single-layer networks called **perceptrons** are covered in Section 19.3, and general multilayer networks in Section 19.4. Section 19.5 illustrates the various uses of neural networks.

[1] Other names that have been used for the field include **connectionism**, **parallel distributed processing**, **neural computation**, **adaptive networks**, and **collective computation**. It should be emphasized that these are artificial neural networks; there is no attempt to build computing elements out of animal tissue.

The network theme is continued in Section 19.6, where we discuss methods for learning belief networks from examples. The connection is deeper than the superficial similarity implied by the word "network"—not only do the two fields share some learning methods, but in some cases, it can be shown that neural networks *are* belief networks.

## 9.1 How the Brain Works

The exact way in which the brain enables thought is one of the great mysteries of science. It has been appreciated for thousands of years that strong blows to the head can lead to unconsciousness, temporary loss of memory, or even permanent loss of mental capability. This suggests that the brain is somehow involved in thought. It has also long been known that the human brain is somehow different; in about 335 B.C. Aristotle wrote, "Of all the animals, man has the largest brain in proportion to his size."[2] Still, it was not until the middle of the eighteenth century that the brain was widely recognized as the seat of consciousness, and it was not until the late nineteenth century that the functional regions of animal brains began to be mapped out. Before the nineteenth century, candidate locations for the seat of consciousness included the heart, the spleen, and the pineal body, a small appendage of the brain present in all vertebrates.

EURON
OMA
ENDRITES
XON

We do know that the **neuron**, or nerve cell, is the fundamental functional unit of all nervous system tissue, including the brain. Each neuron consists of a cell body, or **soma**, that contains a cell nucleus. Branching out from the cell body are a number of fibers called **dendrites** and a single long fiber called the **axon**. Dendrites branch into a bushy network around the cell, whereas the axon stretches out for a long distance—usually about a centimeter (100 times the diameter of the cell body), and as far as a meter in extreme cases. Eventually, the axon also branches into strands and substrands that connect to the dendrites and cell bodies of other neurons. The

NAPSE

connecting junction is called a **synapse**. Each neuron forms synapses with anywhere from a dozen to a hundred thousand other neurons. Figure 19.1 shows the parts of a neuron.

Signals are propagated from neuron to neuron by a complicated electrochemical reaction. Chemical transmitter substances are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body. When the potential reaches a threshold,

TION POTENTIAL

an electrical pulse or **action potential** is sent down the axon. The pulse spreads out along the branches of the axon, eventually reaching synapses and releasing transmitters into the bodies of other cells. Synapses that increase the potential are called **excitatory**, and those that decrease it are called **inhibitory**. Perhaps the most significant finding is that synaptic connections exhibit **plasticity**—long-term changes in the strength of connections in response to the pattern of stimulation. Neurons also form new connections with other neurons, and sometimes entire collections of neurons can migrate from one place to another. These mechanisms are thought to form the basis for learning in the brain.

CITATORY
HIBITORY
ASTICITY

Most information processing goes on in the cerebral cortex, the outer layer of the brain. The basic organizational unit appears to be a barrel-shaped module of tissue about 0.5 mm in

[2] Since then, it has been discovered that some species of dolphins and whales have relatively larger brains. The large size of human brains is now thought to be enabled in part by recent improvements in its cooling system.

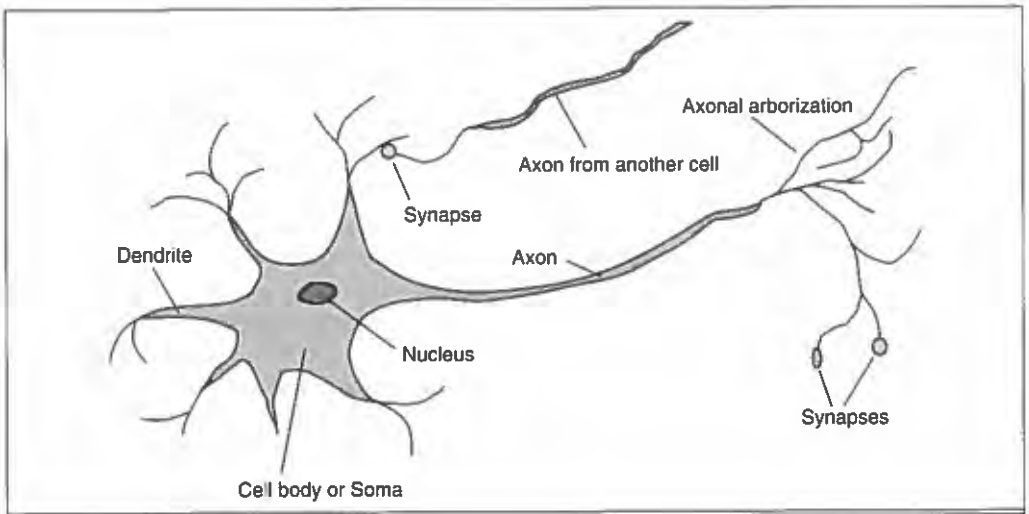


**Figure 19.1** The parts of a nerve cell or neuron. In reality, the length of the axon should be about 100 times the diameter of the cell body.

diameter, extending the full depth of the cortex, which is about 4 mm in humans. A module contains about 2000 neurons. It is known that certain areas of the brain have specific functions. In 1861, Pierre Paul Broca was able to demonstrate that the third left frontal convolution of the cerebral cortex is important for speech and language by his studies of patients with **aphasia**—an inability to speak, often brought on by brain damage. This soon led to surgical experiments on animals that mapped out the connection between areas of the cortex and specific motor controls. We now have some data on the mapping between areas of the brain and the parts of the body that they control, or from which they receive sensory input. Such mappings seem to be able to change radically over the course of a few weeks, and some animals seem to have multiple maps. Moreover, we do not fully understand how other areas can take over functions when one area is damaged. There is almost no theory about how an individual memory is stored.

The truly amazing thing is that *a collection of simple cells can lead to thought, action, and consciousness*. Neurobiology is a long way from a complete theory of consciousness, but even if there are some important electrical or chemical processes that have been overlooked, the amazing conclusion is the same: *brains cause minds* (Searle, 1992). The only real alternative theory is mysticism: that there is some mystical realm in which minds operate that is beyond physical science.

## Comparing brains with digital computers

Brains and digital computers perform quite different tasks, and have different properties. Figure 19.2 shows that there are more neurons in the typical human brain than there are bits in a typical high-end computer workstation. We can predict that this will not hold true for long, because the human brain is evolving very slowly, whereas computer memories are growing rapidly. In any

| | Computer | Human Brain |
|---|---|---|
| Computational units | 1 CPU, $10^5$ gates | $10^{11}$ neurons |
| Storage units | $10^9$ bits RAM, $10^{10}$ bits disk | $10^{11}$ neurons, $10^{14}$ synapses |
| Cycle time | $10^{-8}$ sec | $10^{-3}$ sec |
| Bandwidth | $10^9$ bits/sec | $10^{14}$ bits/sec |
| Neuron updates/sec | $10^5$ | $10^{14}$ |

**Figure 19.2** A crude comparison of the raw computational resources available to computers (*circa* 1994) and brains.

case, the difference in storage capacity is minor compared to the difference in switching speed and in parallelism. Computer chips can execute an instruction in tens of nanoseconds, whereas neurons require milliseconds to fire. Brains more than make up for this, however, because all the neurons and synapses are active simultaneously, whereas most current computers have only one or at most a few CPUs. A neural network running on a serial computer requires hundreds of cycles to decide if a single neuron-like unit will fire, whereas in a real brain, *all* the neurons do this in a single step. Thus, *even though a computer is a million times faster in raw switching speed, the brain ends up being a billion times faster at what it does.* One of the attractions of the neural network approach is the hope that a device could be built that combines the parallelism of the brain with the switching speed of the computer. Full-scale hardware development will depend on finding a family of neural network algorithms that provides a basis for long-term investment.

A brain can perform a complex task—recognize a face, for example—in less than a second, which is only enough time for a few hundred cycles. A serial computer requires billions of cycles to perform the same task less well. Clearly, there *is* an opportunity for massive parallelism here. Neural networks may provide a model for massively parallel computation that is more successful than the approach of "parallelizing" traditional serial algorithms.

Brains are more fault-tolerant than computers. A hardware error that flips a single bit can doom an entire computation, but brain cells die all the time with no ill effect to the overall functioning of the brain. It is true that there are a variety of diseases and traumas that can affect a brain, but for the most part, brains manage to muddle through for 70 or 80 years with no need to replace a memory card, call the manufacturer's service line, or reboot. In addition, brains are constantly faced with novel input, yet manage to do something with it. Computer programs rarely work as well with novel input, unless the programmer has been exceptionally careful. The third attraction of neural networks is **graceful degradation**: they tend to have a gradual rather than sharp drop-off in performance as conditions worsen.

ACEFUL
GRADATION

The final attraction of neural networks is that they are designed to be trained using an inductive learning algorithm. (Contrary to the impression given by the popular media, of course, neural networks are far from being the only AI systems capable of learning.) After the network is initialized, it can be modified to improve its performance on input/output pairs. To the extent that the learning algorithms can be made general and efficient, this increases the value of neural networks as psychological models, and makes them useful tools for creating a wide variety of high-performance applications.

## 9.2 NEURAL NETWORKS

NITS
NKS
EIGHT

A neural network is composed of a number of nodes, or **units**, connected by **links**. Each link has a numeric **weight** associated with it. Weights are the primary means of long-term storage in neural networks, and learning usually takes place by updating the weights. Some of the units are connected to the external environment, and can be designated as input or output units. The weights are modified so as to try to bring the network's input/output behavior more into line with that of the environment providing the inputs.

CTIVATION LEVEL

Each unit has a set of input links from other units, a set of output links to other units, a current **activation level**, and a means of computing the activation level at the next step in time, given its inputs and weights. The idea is that each unit does a local computation based on inputs from its neighbors, but without the need for any global control over the set of units as a whole. In practice, most neural network implementations are in software and use synchronous control to update all the units in a fixed sequence.

To build a neural network to perform some task, one must first decide how many units are to be used, what kind of units are appropriate, and how the units are to be connected to form a network. One then initializes the weights of the network, and trains the weights using a learning algorithm applied to a set of training examples for the task.[3] The use of examples also implies that one must decide how to encode the examples in terms of inputs and outputs of the network.

### Notation

Neural networks have lots of pieces, and to refer to them we will need to introduce a variety of mathematical notations. For convenience, these are summarized in Figure 19.3.

### Simple computing elements

Figure 19.4 shows a typical unit. Each unit performs a simple computation: it receives signals from its input links and computes a new activation level that it sends along each of its output links. The computation of the activation level is based on the values of each input signal received from a neighboring node, and the weights on each input link. The computation is split into two components. First is a *linear* component, called the **input function**, $in_i$, that computes the weighted sum of the unit's input values. Second is a *nonlinear* component called the **activation function**, $g$, that transforms the weighted sum into the final value that serves as the unit's activation value, $a_i$. Usually, all units in a network use the same activation function. Exercise 19.3 explains why it is important to have a nonlinear component.

PUT FUNCTION

TIVATION
INCTION

The total weighted input is the sum of the input activations times their respective weights:

$$in_i = \sum_j W_{j,i} a_j = \mathbf{W}_i \cdot \mathbf{a}_i$$

[3] In this chapter, we will assume that all examples are labelled with the correct outputs. In Chapter 20, we will see how to relax this assumption.

| Notation | Meaning |
|---|---|
| $a_i$ | Activation value of unit $i$ (also the output of the unit) |
| $\mathbf{a}_i$ | Vector of activation values for the inputs to unit $i$ |
| $g$ | Activation function |
| $g'$ | Derivative of the activation function |
| $Err_i$ | Error (difference between output and target) for unit $i$ |
| $Err^e$ | Error for example $e$ |
| $I_i$ | Activation of a unit $i$ in the input layer |
| $\mathbf{I}$ | Vector of activations of all input units |
| $\mathbf{I}^e$ | Vector of inputs for example $e$ |
| $in_i$ | Weighted sum of inputs to unit $i$ |
| $N$ | Total number of units in the network |
| $O$ | Activation of the single output unit of a perceptron |
| $O_i$ | Activation of a unit $i$ in the output layer |
| $\mathbf{O}$ | Vector of activations of all units in the output layer |
| $t$ | Threshold for a step function |
| $T$ | Target (desired) output for a perceptron |
| $\mathbf{T}$ | Target vector when there are several output units |
| $\mathbf{T}^e$ | Target vector for example $e$ |
| $W_{j,i}$ | Weight on the link from unit $j$ to unit $i$ |
| $W_i$ | Weight from unit $i$ to the output in a perceptron |
| $\mathbf{W}_i$ | Vector of weights leading into unit $i$ |
| $\mathbf{W}$ | Vector of all weights in the network |

**Figure 19.3** Neural network notation. Subscripts denote units; superscripts denote examples.

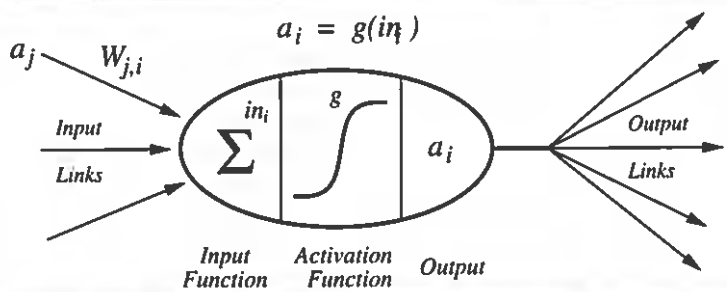


**Figure 19.4** A unit.

where the final expression illustrates the use of vector notation. In this notation, the weights on links into node $i$ are denoted by $\mathbf{W}$, the set of input values is called $\mathbf{a}_i$, and the dot product denotes the sum of the pairwise products.

The elementary computation step in each unit computes the new activation value for the unit by applying the activation function, $g$, to the result of the input function:

$$a_i \leftarrow g(in_i) = g\left(\sum_j W_{j,i} a_j\right)$$

Different models are obtained by using different mathematical functions for $g$. Three common choices are the step, sign, and sigmoid functions, illustrated in Figure 19.5. The step function has a threshold $t$ such that it outputs a 1 when the input is greater than its threshold, and outputs a 0 otherwise. The biological motivation is that a 1 represents the firing of a pulse down the axon, and a 0 represents no firing. The threshold represents the minimum total weighted input necessary to cause the neuron to fire. Threshold versions of the sign and sigmoid functions can also be defined.

In most cases, we will find it mathematically convenient to replace the threshold with an extra input weight. This allows for a simpler learning element because it need only worry about adjusting weights, rather than adjusting both weights and thresholds. Thus, instead of having a threshold $t$ for each unit, we add an extra input whose activation $a_0$ is fixed at $-1$. The extra weight $W_{0,i}$ associated with $a_0$ serves the function of a threshold at $t$, provided that $W_{0,i}a_0 = -t$. Then all units can have a fixed threshold at 0. Mathematically, the two representations for thresholds are entirely equivalent:

$$a_i = \text{step}_t\left(\sum_{j=1}^{n} W_{j,i} a_j\right) = \text{step}_0\left(\sum_{j=0}^{n} W_{j,i} a_j\right) \text{ where } W_{0,i} = t \text{ and } a_0 = -1$$

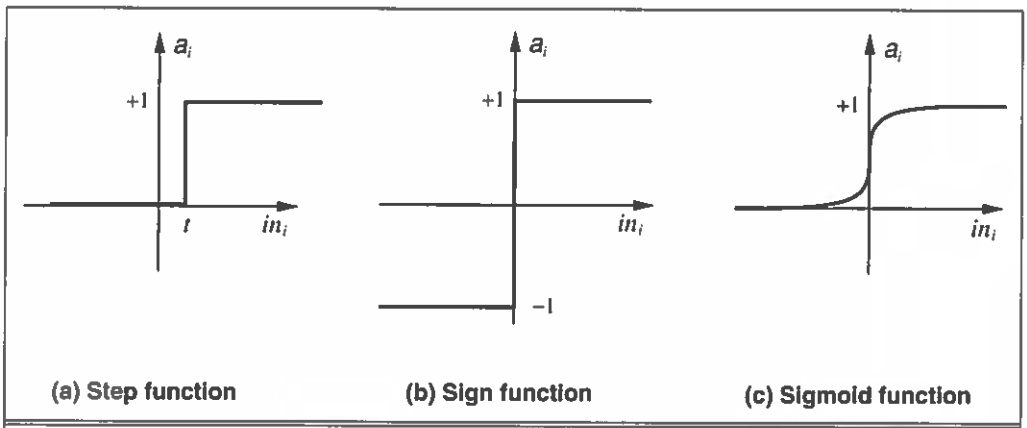


**Figure 19.5** Three different activation functions for units.

$$\text{step}_t(x) = \begin{cases} 1, & \text{if } x \geq t \\ 0, & \text{if } x < t \end{cases} \quad \text{sign}(x) = \begin{cases} +1, & \text{if } x \geq 0 \\ -1, & \text{if } x < 0 \end{cases} \quad \text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

We can get a feel for the operation of individual units by comparing them with logic gates. One of the original motivations for the design of individual units (McCulloch and Pitts, 1943) was their ability to represent basic Boolean functions. Figure 19.6 shows how the Boolean functions *AND*, *OR*, and *NOT* can be represented by units with suitable weights and thresholds. This is important because it means we can use these units to build a network to compute any Boolean function of the inputs.

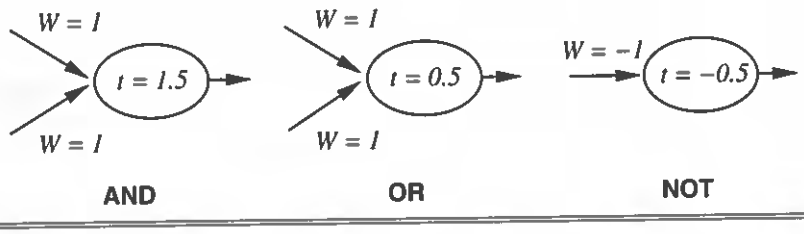


**Figure 19.6** Units with a step function for the activation function can act as logic gates, given appropriate thresholds and weights.

## Network structures

·FORWARD
RRENT

There are a variety of kinds of network structure, each of which results in very different computational properties. The main distinction to be made is between **feed-forward** and **recurrent** networks. In a feed-forward network, links are unidirectional, and there are no cycles. In a recurrent network, the links can form arbitrary topologies. Technically speaking, a feed-forward network is a directed acyclic graph (DAG). We will usually be dealing with networks that are arranged in layers. In a layered feed-forward network, each unit is linked only to units in the next layer; there are no links between units in the same layer, no links backward to a previous layer, and no links that skip a layer. Figure 19.7 shows a very simple example of a layered feed-forward network. This network has *two* layers; because the input units (square nodes) simply serve to pass activation to the next layer, they are not counted (although some authors would describe this as a three-layer network).

The significance of the lack of cycles is that computation can proceed uniformly from input units to output units. The activation from the previous time step plays no part in the computation, because it is not fed back to an earlier unit. Hence, a feed-forward network simply computes a function of the input values that depends on the weight settings—it has *no internal state* other than the weights themselves. Such networks can implement adaptive versions of simple reflex agents or they can function as components of more complex agents. In this chapter, we will focus on feed-forward networks because they are relatively well-understood.

Obviously, the brain cannot be a feed-forward network, else we would have no short-term memory. Some regions of the brain are largely feed-forward and somewhat layered, but there are rampant back-connections. In our terminology, the brain is a recurrent network. Because activation is fed back to the units that caused it, recurrent networks have internal state stored in the activation levels of the units. This also means that computation can be much less orderly

than in feed-forward networks. Recurrent networks can become unstable, or oscillate, or exhibit chaotic behavior. Given some input values, it can take a long time to compute a stable output, and learning is made more difficult. On the other hand, recurrent networks can implement more complex agent designs and can model systems with state. Because recurrent networks require some quite advanced mathematical methods, we can only provide a few pointers here.

HOPFIELD NETWORKS

**Hopfield networks** are probably the best-understood class of recurrent networks. They use *bidirectional* connections with *symmetric* weights (i.e., $W_{i,j} = W_{j,i}$); all of the units are both input and output units; the activation function $g$ is the sign function; and the activation levels can only be $\pm 1$. A Hopfield network functions as an **associative memory**—after training on a set of examples, a new stimulus will cause the network to settle into an activation pattern corresponding to the example in the training set that *most closely resembles* the new stimulus. For example, if the training set consists of a set of photographs, and the new stimulus is a small piece of one of the photographs, then the network activation levels will reproduce the photograph from which the piece was taken. Notice that the original photographs are not stored separately in the network; each weight is a partial encoding of all the photographs. One of the most interesting theoretical results is that Hopfield networks can reliably store up to $0.138N$ training examples, where $N$ is the number of units in the network.

ASSOCIATIVE MEMORY

BOLTZMANN MACHINES

**Boltzmann machines** also use symmetric weights, but include units that are neither input nor output units (cf. the units labelled $H_3$ and $H_4$ in Figure 19.7). They also use a *stochastic* activation function, such that the probability of the output being 1 is some function of the total weighted input. Boltzmann machines therefore undergo state transitions that resemble a simulated annealing search for the configuration that best approximates the training set (see Chapter 4). It turns out that Boltzmann machines are formally identical to a special case of belief networks evaluated with a stochastic simulation algorithm (see Section 15.4).

Returning to feed-forward networks, there is one more important distinction to be made. Examine Figure 19.7, which shows the topology of a very simple neural network. On the left are the **input units**. The activation value of each of these units is determined by the environment. At the right-hand end of the network are four **output units**. In between, the nodes labelled $H_3$ and $H_4$ have no direct connection to the outside world. These are called **hidden units**, because they cannot be directly observed by noting the input/output behavior of the network. Some networks, called **perceptrons**, have no hidden units. This makes the learning problem much simpler, but it means that perceptrons are very limited in what they can represent. Networks with one or more layers of hidden units are called **multilayer networks**. With one (sufficiently large) layer of hidden units, it is possible to represent any continuous function of the inputs; with two layers, even discontinuous functions can be represented.

INPUT UNITS

OUTPUT UNITS

HIDDEN UNITS

PERCEPTRONS

MULTILAYER NETWORKS

With a fixed structure and fixed activation functions $g$, the functions representable by a feed-forward network are restricted to have a specific parameterized structure. The weights chosen for the network determine which of these functions is actually represented. For example, the network in Figure 19.7 calculates the following function:

$$\begin{aligned} a_5 &= g(W_{3,5}a_3 + W_{4,5}a_4) \\ &= g(W_{3,5}g(W_{1,3}a_1 + W_{2,3}a_2) + W_{4,5}g(W_{1,4}a_1 + W_{2,4}a_2)) \end{aligned} \tag{19.1}$$

where $g$ is the activation function, and $a_i$ is the output of node $i$. Notice that because the activation functions $g$ are nonlinear, the whole network represents a complex nonlinear function. If you

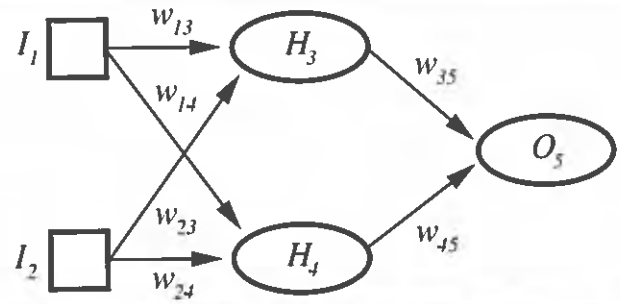


**Figure 19.7** A very simple, two-layer, feed-forward network with two inputs, two hidden nodes, and one output node.

LINEAR RESSION

think of the weights as parameters or coefficients of this function, then *learning just becomes a process of tuning the parameters to fit the data in the training set*—a process that statisticians call **nonlinear regression**. From the statistical viewpoint, this is what neural networks do.

## Optimal network structure

So far we have considered networks with a fixed structure, determined by some outside authority. This is a potential weak point, because the wrong choice of network structure can lead to poor performance. If we choose a network that is too small, then the model will be incapable of representing the desired function. If we choose a network that is too big, it will be able to memorize all the examples by forming a large lookup table, but will not generalize well to inputs that have not been seen before. In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters (i.e., weights) in the model. We saw this in Figure 18.2 (page 530), where the high-parameter models (b) and (c) fit all the data, but may not generalize as well as the low-parameter model (d).

It is known that a feed-forward network with one hidden layer can approximate any continuous function of the inputs, and a network with two hidden layers can approximate any function at all. However, the number of units in each layer may grow exponentially with the number of inputs. As yet, we have no good theory to characterize **NERFs**, or Network Efficiently Representable Functions—functions that can be approximated with a small number of units.

FS

We can think of the problem of finding a good network structure as a search problem. One approach that has been used is to use a **genetic algorithm** (Chapter 20) to search the space of network structures. However, this is a very large space, and evaluating a state in the space means running the whole neural network training protocol, so this approach is very CPU-intensive. Therefore, it is more common to see hill-climbing searches that selectively modify an existing network structure. There are two ways to do this: start with a big network and make it smaller, or start with a small one and make it bigger.

The zip code reading network described on page 586 uses an approach called **optimal brain damage** to remove weights from the initial fully-connected model. After the network is

initially trained, an information theoretic approach identifies an optimal selection of connections that can be dropped (i.e., the weights are set to zero). The network is then retrained, and if it is performing as well or better, the process is repeated. This process was able to eliminate 3/4 of the weights, and improve overall performance on test data. In addition to removing connections, it is also possible to remove units that are not contributing much to the result.

Several algorithms have been proposed for growing a larger network from a smaller one. The **tiling algorithm** (Mézard and Nadal, 1989) is interesting because it is similar to the decision tree learning algorithm. The idea is to start with a single unit that does its best to produce the correct output on as many of the training examples as possible. Subsequent units are added to take care of the examples that the first unit got wrong. The algorithm adds only as many units as are needed to cover all the examples.

The **cross-validation** techniques of Chapter 18 are useful for deciding when we have found a network of the right size.

## 19.3 PERCEPTRONS

Layered feed-forward networks were first studied in the late 1950s under the name **perceptrons**. Although networks of all sizes and topologies were considered, the only effective learning element at the time was for single-layered networks, so that is where most of the effort was spent. Today, the name perceptron is used as a synonym for a single-layer, feed-forward network. The left-hand side of Figure 19.8 shows such a perceptron network. Notice that each output unit is independent of the others—each weight only affects one of the outputs. That means that we can limit our study to perceptrons with a single output unit, as in the right-hand side of Figure 19.8, and use several of them to build up a multi-output perceptron. For convenience, we can drop subscripts, denoting the output unit as $O$ and the weight from input unit $j$ to $O$ as $W_j$. The activation of input unit $j$ is given by $I_j$. The activation of the output unit is therefore

$$O = Step_0\left(\sum_j W_j I_j\right) = Step_0(\mathbf{W} \cdot \mathbf{I}) \tag{19.2}$$

where, as discussed earlier, we have assumed an additional weight $W_0$ to provide a threshold for the step function, with $I_0 = -1$.

### What perceptrons can represent

We saw in Figure 19.6 that units can represent the simple Boolean functions AND, OR, and NOT, and that therefore a feed-forward network of units can represent any Boolean function, if we allow for enough layers and units. But what Boolean functions can be represented with a single-layer perceptron?

Some complex Boolean functions can be represented. For example, the **majority function**, which outputs a 1 only if more than half of its $n$ inputs are 1, can be represented by a perceptron with each $W_j = 1$ and threshold $t = n/2$. This would require a decision tree with $O(2^n)$ nodes.

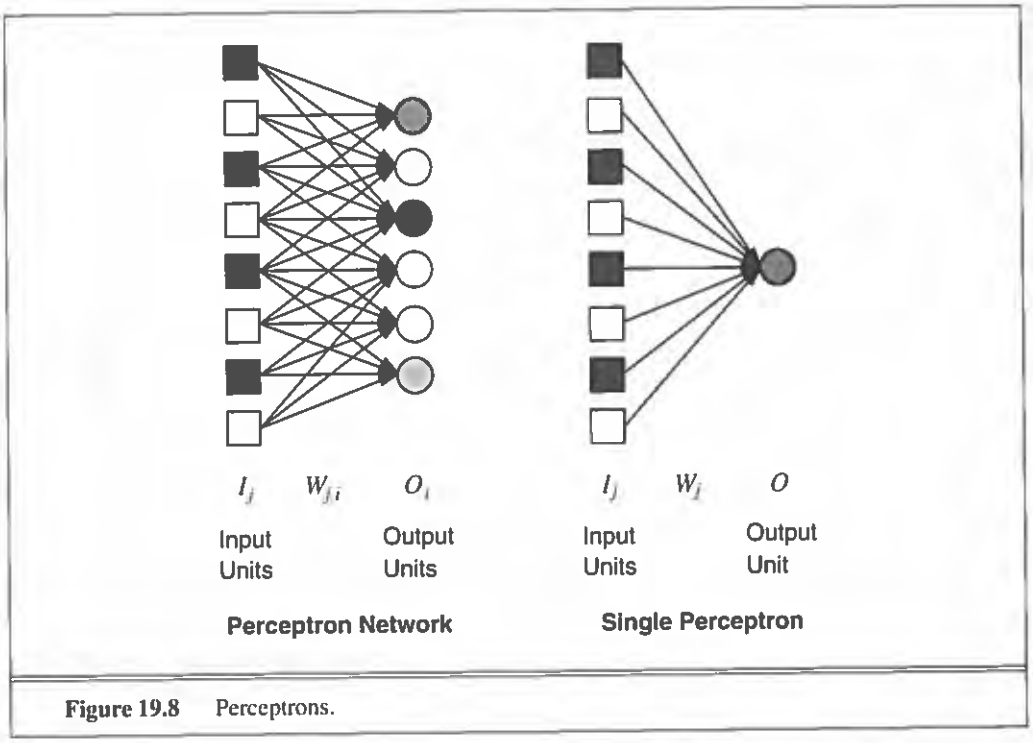


**Figure 19.8** Perceptrons.

The perceptron, with 1 unit and $n$ weights, gives a much more compact representation of this function. In accordance with Ockham's razor, we would expect the perceptron to do a much better job of learning a majority function, as we will soon see.

Unfortunately, it turns out that perceptrons are severely limited in the Boolean functions they can represent. The problem is that any input $I_j$ can only influence the final output in one direction, no matter what the other input values are. Consider some input vector **a**. Suppose that this vector has $a_j = 0$ and that the vector produces a 0 as output. Furthermore, suppose that when $a_j$ is replaced with 1, the output changes to 1. This implies that $W_j$ must be positive. It also implies that there can be no input vector **b** for which the output is 1 when $b_j = 0$, but the output is 0 when $b_j$ is replaced with 1. Because this limitation applies to each input, the result is a severe limitation in the total number of functions that can be represented. For example, the perceptron is unable to represent the function for deciding whether or not to wait for a table at a restaurant (shown as a decision tree in Figure 18.4).

A little geometry helps make clear what is going on. Figure 19.9 shows three different Boolean functions of two inputs, the AND, OR, and XOR functions. Each function is represented as a two-dimensional plot, based on the values of the two inputs. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. As we will explain shortly, a perceptron can represent a function only if there is some line that separates all the white dots from the black dots. Such functions are called **linearly separable**. Thus, a perceptron can represent AND and OR, but not XOR.

LINEARLY SEPARABLE

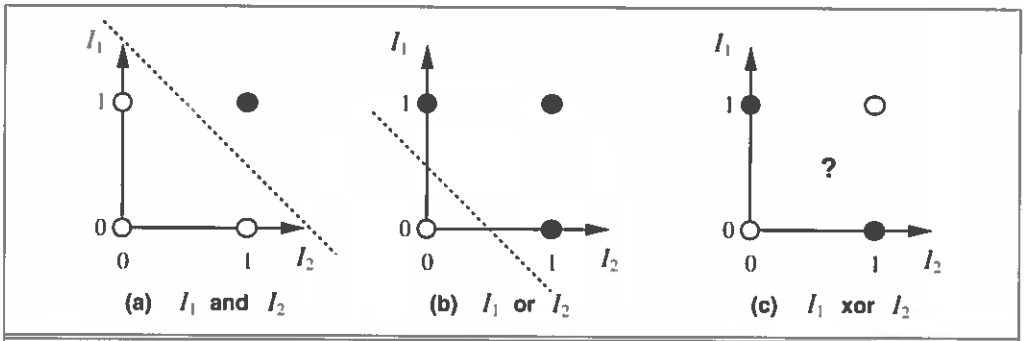


**Figure 19.9** Linear separability in perceptrons.

The fact that a perceptron can only represent linearly separable functions follows directly from Equation (19.2), which defines the function computed by a perceptron. A perceptron outputs a 1 only if $\mathbf{W} \cdot \mathbf{I} > 0$. This means that the entire input space is divided in two along a boundary defined by $\mathbf{W} \cdot \mathbf{I} = 0$, that is, a plane in the input space with coefficients given by the weights. With $n$ inputs, the input space is $n$-dimensional, and linear separability can be rather hard to visualize if $n$ is too large. It is easiest to understand for the case where $n = 2$. In Figure 19.9(a), one possible separating "plane" is the dotted line defined by the equation

$$I_1 = -I_2 + 1.5 \qquad \text{or} \qquad I_1 + I_2 = 1.5$$

The region above the line, where the output is 1, is therefore given by

$$-1.5 + I_1 + I_2 > 0$$

or, in vector notation,

$$\mathbf{W} \cdot \mathbf{I} = \langle 1.5, 1, 1 \rangle \cdot \langle -1, I_1, I_2 \rangle > 0$$

With three inputs, the separating plane can still be visualized. Figure 19.10(a) shows an example in three dimensions. The function we are trying to represent is true if and only if a minority of its three inputs are true. The shaded separating plane is defined by the equation

$$I_1 + I_2 + I_3 = 1.5$$

This time the positive outputs lie below the plane, in the region

$$(-I_1) + (-I_2) + (-I_3) > -1.5$$

Figure 19.10(b) shows a unit to implement the function.

## Learning linearly separable functions

As with any performance element, the question of what perceptrons can represent is prior to the question of what they can learn. We have just seen that a function can be represented by a perceptron if and only if it is linearly separable. That is relatively bad news, because there are not many linearly separable functions. The (relatively) good news is that *there is a perceptron algorithm that will learn any linearly separable function, given enough training examples.*

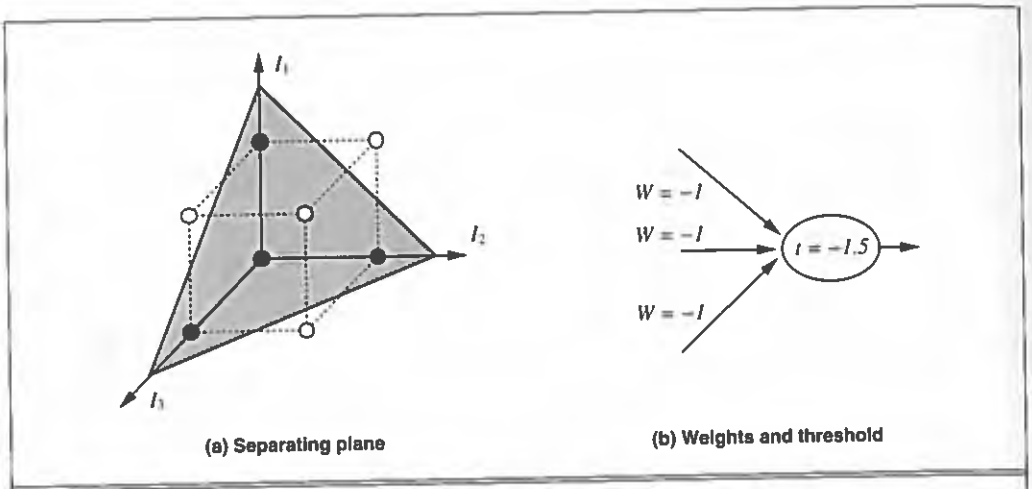


**Figure 19.10** Linear separability in three dimensions—representing the "minority" function.

Most neural network learning algorithms, including the perceptron learning method, follow the current-best-hypothesis (CBH) scheme described in Chapter 18. In this case, the hypothesis is a network, defined by the current values of the weights. The initial network has randomly assigned weights, usually from the range [–0.5,0.5]. The network is then updated to try to make it consistent with the examples. This is done by making small adjustments in the weights to reduce the difference between the observed and predicted values. The main difference from the logical algorithms is the need to repeat the update phase several times for each example in order to achieve convergence. Typically, the updating process is divided into **epochs**. Each epoch involves updating all the weights for all the examples. The general scheme is shown as NEURAL-NETWORK-LEARNING in Figure 19.11.

EPOCHS

For perceptrons, the weight update rule is particularly simple. If the predicted output for the single output unit is $O$, and the correct output should be $T$, then the error is given by

$$Err = T - O$$

If the error is positive, then we need to increase $O$; if it is negative, we need to decrease $O$. Now each input unit contributes $W_j I_j$ to the total input, so if $I_j$ is positive, an increase in $W_j$ will tend to increase $O$, and if $I_j$ is negative, an increase in $W_j$ will tend to decrease $O$. Thus, we can achieve the effect we want with the following rule:

$$W_j \leftarrow W_j + \alpha \times I_j \times Err$$

LEARNING RATE
PERCEPTRON LEARNING RULE

where the term $\alpha$ is a constant called the **learning rate**. This rule is a slight variant of the **perceptron learning rule** proposed by Frank Rosenblatt in 1960. Rosenblatt proved that a learning system using the perceptron learning rule will converge to a set of weights that correctly represents the examples, as long as the examples represent a linearly separable function.

The perceptron convergence theorem created a good deal of excitement when it was announced. People were amazed that such a simple procedure could correctly learn any representable function, and there were great hopes that intelligent machines could be built from

```
function NEURAL-NETWORK-LEARNING(examples) returns network

  network ← a network with randomly assigned weights
  repeat
    for each e in examples do
      O ← NEURAL-NETWORK-OUTPUT(network, e)
      T ← the observed output values from e
      update the weights in network based on e, O, and T
    end
  until all examples correctly predicted or stopping criterion is reached
  return network
```

**Figure 19.11** The generic neural network learning method: adjust the weights until predicted output values **O** and true values **T** agree.

perceptrons. It was not until 1969 that Minsky and Papert undertook what should have been the first step: analyzing the class of representable functions. Their book *Perceptrons* (Minsky and Papert, 1969) clearly demonstrated the limits of linearly separable functions.

In retrospect, the perceptron convergence theorem should not have been surprising. The perceptron is doing a **gradient descent** search through weight space (see Chapter 4). It is fairly easy to show that the weight space has no local minima. Provided the learning rate parameter is not so large as to cause "overshooting," the search will converge on the correct weights. In short, perceptron learning is easy because the space of representable functions is simple.

We can examine the learning behavior of perceptrons using the method of constructing learning curves, as described in Chapter 18. There is a slight difference between the example descriptions used for neural networks and those used for other attribute-based methods such as decision trees. In a neural network, all inputs are real numbers in some fixed range, whereas decision trees allow for multivalued attributes with a discrete set of values. For example, the attribute for the number of patrons in the restaurant has values *None*, *Some*, and *Full*. There are two ways to handle this. In a **local encoding**, we use a single input unit and pick an appropriate number of distinct values to correspond to the discrete attribute values. For example, we can use *None* = 0.0, *Some* = 0.5, and *Full* = 1.0. In a **distributed encoding**, we use one input unit for each value of the attribute, turning on the unit that corresponds to the correct value.

LOCAL ENCODING

DISTRIBUTED ENCODING

Figure 19.12 shows the learning curve for a perceptron on two different problems. On the left, we show the curve for learning the majority function with 11 Boolean inputs (i.e., the function outputs a 1 if 6 or more inputs are 1). As we would expect, the perceptron learns the function quite quickly because the majority function is linearly separable. On the other hand, the decision tree learner makes no progress, because the majority function is very hard (although not impossible) to represent as a decision tree. On the right of the figure, we have the opposite situation. The *WillWait* problem is easily represented as a decision tree, but is not linearly separable. The perceptron algorithm draws the best plane it can through the data, but can manage no more than 65% accuracy.

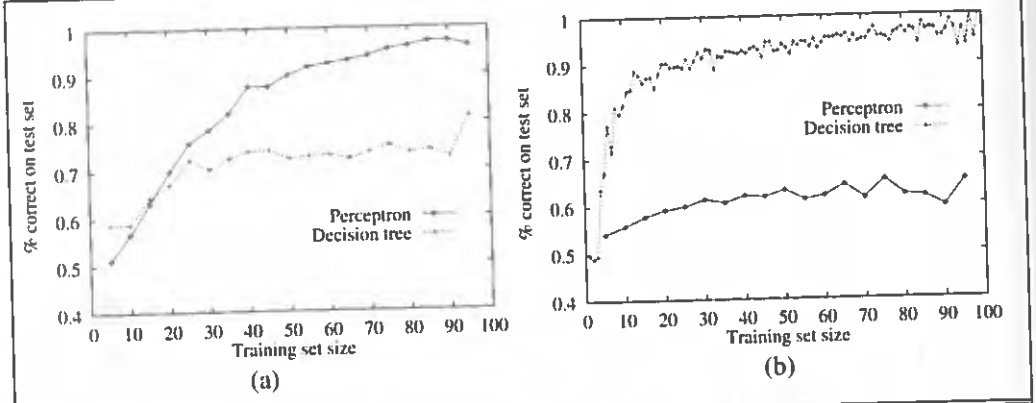


**Figure 19.12** Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate for the restaurant example.

## 19.4 MULTILAYER FEED-FORWARD NETWORKS

Rosenblatt and others described multilayer feed-forward networks in the late 1950s, but concentrated their research on single-layer perceptrons. This was mainly because of the difficulty of finding a sensible way to update the weights between the inputs and the hidden units; whereas an error signal can be calculated for the output units, it is harder to see what the error signal should be for the hidden units. When the book *Perceptrons* was published, Minsky and Papert (1969) stated that it was an "important research problem" to investigate multilayer networks more thoroughly, although they speculated that "there is no reason to suppose that any of the virtues [of perceptrons] carry over to the many-layered version." In a sense, they were right. Learning algorithms for multilayer networks are neither efficient nor guaranteed to converge to a global optimum. On the other hand, the results of computational learning theory tell us that learning general functions from examples is an intractable problem in the worst case, regardless of the method, so we should not be too dismayed.

BACK-PROPAGATION

The most popular method for learning in multilayer networks is called **back-propagation**. It was first invented in 1969 by Bryson and Ho, but was more or less ignored until the mid-1980s. The reasons for this may be sociological, but may also have to do with the computational requirements of the algorithm on nontrivial problems.

### Back-Propagation Learning

Suppose we want to construct a network for the restaurant problem. We have already seen that a perceptron is inadequate, so we will try a two-layer network. We have ten attributes

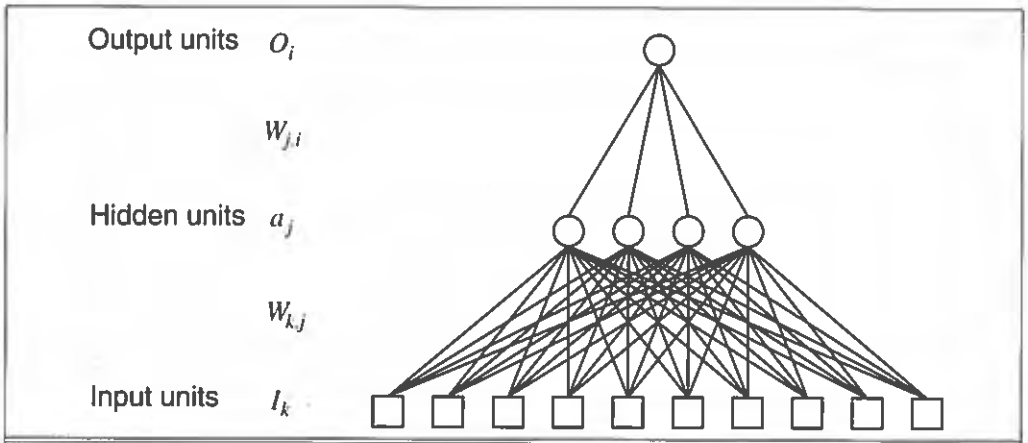


**Figure 19.13** A two-layer feed-forward network for the restaurant problem.

describing each example, so we will need ten input units. How many hidden units are needed? In Figure 19.13, we show a network with four hidden units. This turns out to be about right for this problem. The problem of choosing the right number of hidden units in advance is still not well-understood. We cover what is known on page 572.

Learning in such a network proceeds the same way as for perceptrons: example inputs are presented to the network, and if the network computes an output vector that matches the target, nothing is done. If there is an error (a difference between the output and target), then the weights are adjusted to reduce this error. *The trick is to assess the blame for an error and divide it among the contributing weights.* In perceptrons, this is easy, because there is only one weight between each input and the output. But in multilayer networks, there are many weights connecting each input to an output, and each of these weights contributes to more than one output.

The back-propagation algorithm is a sensible approach to dividing the contribution of each weight. As in the perceptron learning algorithm, we try to minimize the error between each target output and the output actually computed by the network.[4] At the output layer, the weight update rule is very similar to the rule for the perceptron. There are two differences: the activation of the hidden unit $a_j$ is used instead of the input value; and the rule contains a term for the gradient of the activation function. If $Err_i$ is the error $(T_i - O_i)$ at the output node, then the weight update rule for the link from unit $j$ to unit $i$ is

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times Err_i \times g'(in_i)$$

where $g'$ is the derivative of the activation function $g$. We will find it convenient to define a new error term $\Delta_i$, which for output nodes is defined as $\Delta_i = Err_i g'(in_i)$. The update rule then becomes

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i \tag{19.3}$$

For updating the connections between the input units and the hidden units, we need to define a quantity analogous to the error term for output nodes. Here is where we do the error back-propagation. The idea is that hidden node $j$ is "responsible" for some fraction of the error $\Delta_i$ in

[4] Actually, we minimize the square of the error; Section 19.4 explains why, but the result is almost the same.

each of the output nodes to which it connects. Thus, the $\Delta_i$ values are divided according to the strength of the connection between the hidden node and the output node, and propagated back to provide the $\Delta_j$ values for the hidden layer. The propagation rule for the $\Delta$ values is the following:

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \tag{19.4}$$

Now the weight update rule for the weights between the inputs and the hidden layer is almost identical to the update rule for the output layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times I_k \times \Delta_j$$

The detailed algorithm is shown in Figure 19.14. It can be summarized as follows:

- Compute the $\Delta$ values for the output units using the observed error.
- Starting with output layer, repeat the following for each layer in the network, until the earliest hidden layer is reached:
  - Propagate the $\Delta$ values back to the previous layer.
  - Update the weights between the two layers.

Recall that in computing the observed error for a given example, NEURAL-NETWORK-LEARNING first feeds the example to the network inputs in order to calculate the predicted output values. During this computation, it is a good idea to save some of the intermediate values computed in each unit. In particular, caching the activation gradient $g'(in_i)$ in each unit speeds up the subsequent back-propagation phase enormously.

Now that we have a learning method for multilayer networks, we can test our claim that adding a hidden layer makes the network more expressive. In Figure 19.15, we show two curves. The first is a **training curve**, which shows the mean squared error on a given training set of 100 restaurant examples during the weight-updating process. This demonstrates that the network does indeed converge to a perfect fit to the training data. The second curve is the standard learning curve for the restaurant data, with one minor exception: the $y$-axis is no longer the proportion of correct answers on the test set, because sigmoid units do not give 0/1 outputs. Instead, we use the mean squared error on the test set, which happens to coincide with the proportion of correct answers in the 0/1 case. The curve clearly shows that the network is capable of learning in the restaurant domain; indeed, the curve is very similar to that for decision-tree learning, albeit somewhat shallower.

NING CURVE

## Back-propagation as gradient descent search

We have given some suggestive reasons why the back-propagation equations are reasonable. It turns out that the equations also can be given a very simple interpretation as a method for performing **gradient descent** in weight space. In this case, the gradient is on the **error surface**: the surface that describes the error on each example as a function of the all the weights in the network. An example error surface is shown in Figure 19.16. The current set of weights defines a point on this surface. At that point, we look at the slope of the surface along the axis formed by each weight. This is known as the *partial derivative* of the surface with respect to each weight—how much the error would change if we made a small change in weight. We then alter

IOR SURFACE

```
function BACK-PROP-UPDATE(network, examples, α) returns a network with modified weights
  inputs: network, a multilayer network
          examples, a set of input/output pairs
          α, the learning rate

  repeat
    for each e in examples do
      /* Compute the output for this example */
        O ← RUN-NETWORK(network, I^e)
      /* Compute the error and Δ for units in the output layer */
        Err^e ← T^e − O
      /* Update the weights leading to the output layer */
        W_{j,i} ← W_{j,i} + α × a_j × Err_i^e × g'(in_i)
      for each subsequent layer in network do
        /* Compute the error at each node */
          Δ_j ← g'(in_j) Σ_i W_{j,i} Δ_i
        /* Update the weights leading into the layer */
          W_{k,j} ← W_{k,j} + α × I_k × Δ_j
      end
    end
  until network has converged
  return network
```

**Figure 19.14** The back-propagation algorithm for updating weights in a multilayer network.

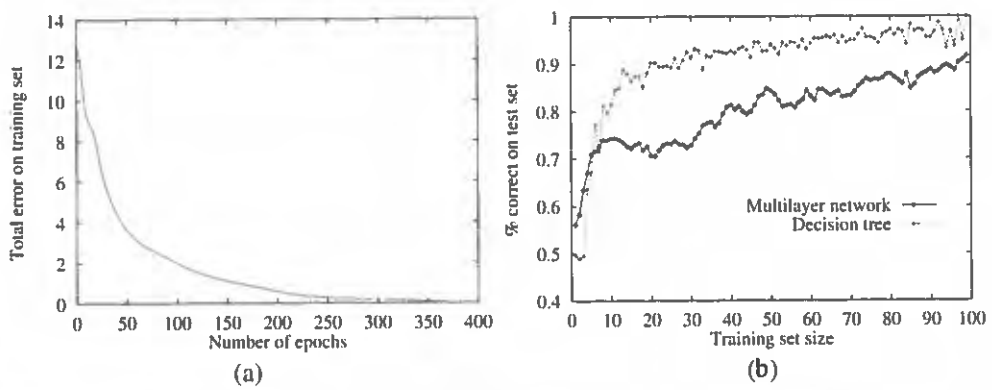


**Figure 19.15** (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves for a back-propagation and decision-tree learning.

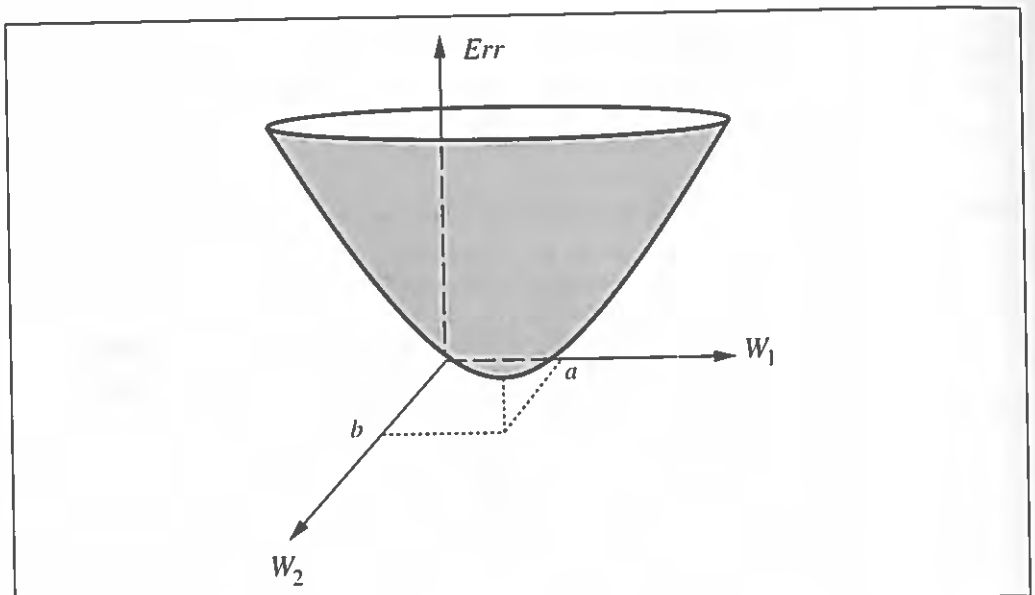


**Figure 19.16** An error surface for gradient descent search in weight space. When $w_1 = a$ and $w_2 = b$, the error on the training set is minimized.

the weights in an amount proportional to the slope in each direction. This moves the network as a whole in the direction of steepest descent on the error surface.

Basically, this is the key: *back-propagation provides a way of dividing the calculation of the gradient among the units, so the change in each weight can be calculated by the unit to which the weight is attached, using only local information.* Like any gradient descent search, back-propagation has problems with efficiency and convergence, as we will discuss shortly. Nonetheless, the decomposition of the learning algorithm is a major step towards parallelizable and biologically plausible learning mechanisms.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. We begin with the error function itself. Because of its convenient form, we use the sum of squared errors over the output values:

$$E = \frac{1}{2}\sum_i (T_i - O_i)^2$$

The key insight, again, is that the output values $O_i$ are a function of the weights (see Equation (19.1), for example). For a general two-layer network, we can write

$$\begin{aligned} E(\mathbf{W}) &= \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j W_{j,i} a_j\right)\right)^2 \\ &= \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j W_{j,i}\, g\left(\sum_k W_{k,j} I_k\right)\right)\right)^2 \end{aligned} \tag{19.5}$$

Notice that although the $a_j$ term in the first line represents a complex expression, it does not depend on $W_{j,i}$. Also, only one of the terms in the summation over $i$ and $j$ depends on a particular $W_{j,i}$, so all the other terms are treated as constants with respect to $W_{j,i}$ and will disappear when differentiated. Hence, when we differentiate the first line with respect to $W_{j,i}$, we obtain

$$\begin{aligned}\frac{\partial E}{\partial W_{j,i}} &= -a_j(T_i - O_i)g'\left(\sum_j W_{j,i}a_j\right) \\ &= -a_j(T_i - O_i)g'(in_i) = -a_j\Delta_i\end{aligned}$$

with $\Delta_i$ defined as before. The derivation of the gradient with respect to $W_{k,j}$ is slightly more complex, but has a similar result:

$$\frac{\partial E}{\partial W_{k,j}} = -I_k\Delta_j$$

To obtain the update rules for the weights, we have to remember that the object is to *minimize* the error, so we need to take a small step in the direction *opposite* to the gradient.

There is one minor technical observation to make about these update rules. They require the derivative of the activation function $g$, so we cannot use the sign or step functions. Back-propagation networks usually use the sigmoid function, or some variant thereof. The sigmoid also has the convenient property that the derivative $g' = g(1 - g)$, so that little extra calculation is needed to find $g'(in_i)$.

## Discussion

Let us stand back for a moment from the delightful mathematics and the fascinating biology, and ask whether back-propagation learning in multilayer networks is a good method for machine learning. We can examine the same set of issues that arose in Chapter 18:

- **Expressiveness**: Neural networks are clearly an attribute-based representation, and do not have the expressive power of general logical representations. They are well-suited for continuous inputs and outputs, unlike most decision tree systems. The class of multilayer networks *as a whole* can represent any desired function of a set of attributes, but any *particular* network may have too few hidden units. It turns out that $2^n/n$ hidden units are needed to represent all Boolean functions of $n$ inputs. This should not be too surprising. Such a network has $O(2^n)$ weights, and we need at least $2^n$ bits to specify a Boolean function. In practice, most problems can be solved with many fewer weights. Designing a good topology is, however, a black art.
- **Computational efficiency**: Computational efficiency depends on the amount of computation time required to train the network to fit a given set of examples. If there are $m$ examples, and $|\mathbf{W}|$ weights, each epoch takes $O(m|\mathbf{W}|)$ time. However, work in computational learning theory has shown that the worst-case number of epochs can be exponential in $n$, the number of inputs. In practice, time to convergence is highly variable, and a vast array of techniques have been developed to try to speed up the process using an assortment of tunable parameters. Local minima in the error surface are also a problem. Networks quite often converge to give a constant "yes" or "no" output, whichever is most common

in the training set. At the cost of some additional computation, the simulated annealing method (Chapter 4) can be used to assure convergence to a global optimum.

- ◇ **Generalization:** As we have seen in our experiments on the restaurant data, neural networks can do a good job of generalization. One can say, somewhat circularly, that they will generalize well on functions for which they are well-suited. These seem to be functions in which the interactions between inputs are not too intricate, and for which the output varies smoothly with the input. There is no theorem to be proved here, but it does seem that neural networks have had reasonable success in a number of real-world problems.
- ◇ **Sensitivity to noise**: Because neural networks are essentially doing nonlinear regression, they are very tolerant of noise in the input data. They simply find the best fit given the constraints of the network topology. On the other hand, it is often useful to have some idea of the degree of certainty of the output values. Neural networks do not provide probability distributions on the output values. For this purpose, belief networks seem more appropriate.
- ◇ **Transparency**: Neural networks are essentially black boxes. Even if the network does a good job of predicting new cases, many users will still be dissatisfied because they will have no idea *why* a given output value is reasonable. If the output value represents, for example, a decision to perform open heart surgery, then an explanation is clearly in order. With decision trees and other logical representations, the output can be explained as a logical derivation and by appeal to a specific set of cases that supports the decision. This is not currently possible with neural networks.
- ◇ **Prior knowledge**: As we mentioned in Chapter 18, learning systems can often benefit from prior knowledge that is available to the user or expert. Prior knowledge can mean the difference between learning from a few well-chosen examples and failing to learn anything at all. Unfortunately, because of the lack of transparency, it is quite hard to use one's knowledge to "prime" a network to learn better. Some tailoring of the network topology can be done—for example, when training on visual images it is common to connect only small sets of nearby pixels to any given unit in the first hidden layer. On the other hand, such "rules of thumb" do not constitute a *mechanism* by which previously accumulated knowledge can be used to learn from subsequent experience. It is possible that learning methods for belief networks can overcome this problem (see Section 19.6).

All these considerations suggest that simple feed-forward networks, although very promising as construction tools for learning complex input/output mappings, do not fulfill our needs for a comprehensive theory of learning in their present form. Researchers in AI, psychology, theoretical computer science, statistics, physics, and biology are working hard to overcome the difficulties.