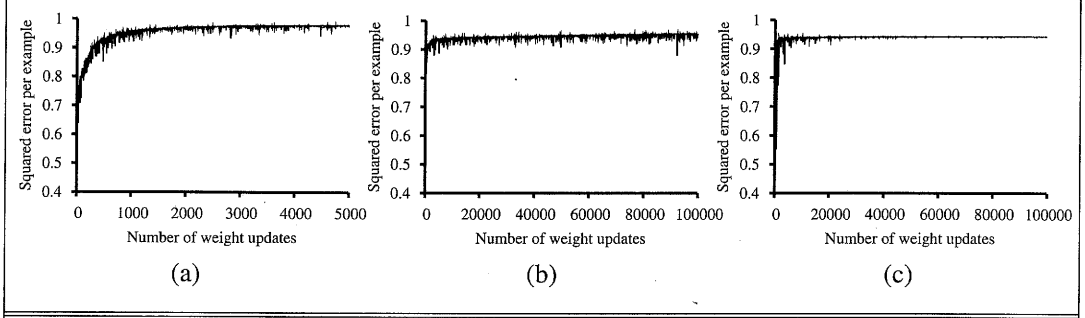

Figure 18.18 Repeat of the experiments in Figure 18.16 using logistic regression and squared error. The plot in (a) covers 5000 iterations rather than 1000, while (b) and (c) use the same scale.

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

so the weight update for minimizing the loss is

$$w_i \leftarrow w_i + \alpha\,(y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i\,. \tag{18.8}$$

Repeating the experiments of Figure 18.16 with logistic regression instead of the linear threshold classifier, we obtain the results shown in Figure 18.18. In (a), the linearly separable case, logistic regression is somewhat slower to converge, but behaves much more predictably. In (b) and (c), where the data are noisy and nonseparable, logistic regression converges far more quickly and reliably. These advantages tend to carry over into real-world applications and logistic regression has become one of the most popular classification techniques for problems in medicine, marketing and survey analysis, credit scoring, public health, and other applications.

18.7 Artificial Neural Networks

We turn now to what seems to be a somewhat unrelated topic: the brain. In fact, as we will see, the technical ideas we have discussed so far in this chapter turn out to be useful in building mathematical models of the brain's activity; conversely, thinking about the brain has helped in extending the scope of the technical ideas.

Chapter 1 touched briefly on the basic findings of neuroscience—in particular, the hypothesis that mental activity consists primarily of electrochemical activity in networks of brain cells called **neurons**. (Figure 1.2 on page 11 showed a schematic diagram of a typical neuron.) Inspired by this hypothesis, some of the earliest AI work aimed to create artificial **neural networks**. (Other names for the field include **connectionism**, **parallel distributed processing**, and **neural computation**.) Figure 18.19 shows a simple mathematical model of the neuron devised by McCulloch and Pitts (1943). Roughly speaking, it "fires" when a linear combination of its inputs exceeds some (hard or soft) threshold—that is, it implements

NEURAL NETWORK

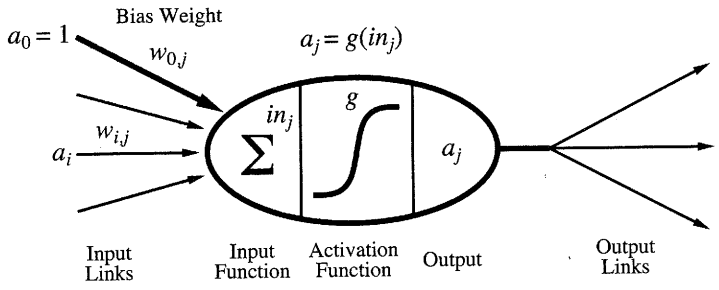

Figure 18.19 A simple mathematical model for a neuron. The unit's output activation is $a_j = g(\sum_{i=0}^{n} w_{i,j} a_i)$, where a_i is the output activation of unit i and $w_{i,j}$ is the weight on the link from unit i to this unit.

a linear classifier of the kind described in the preceding section. A neural network is just a collection of units connected together; the properties of the network are determined by its topology and the properties of the "neurons."

COMPUTATIONAL NEUROSCIENCE

Since 1943, much more detailed and realistic models have been developed, both for neurons and for larger systems in the brain, leading to the modern field of **computational neuroscience**. On the other hand, researchers in AI and statistics became interested in the more abstract properties of neural networks, such as their ability to perform distributed computation, to tolerate noisy inputs, and to learn. Although we understand now that other kinds of systems—including Bayesian networks—have these properties, neural networks remain one of the most popular and effective forms of learning system and are worthy of study in their own right.

18.7.1 Neural network structures

UNIT
LINK
ACTIVATION
WEIGHT

Neural networks are composed of nodes or **units** (see Figure 18.19) connected by directed **links**. A link from unit i to unit j serves to propagate the **activation** a_i from i to j.[8] Each link also has a numeric **weight** $w_{i,j}$ associated with it, which determines the strength and sign of the connection. Just as in linear regression models, each unit has a dummy input $a_0 = 1$ with an associated weight $w_{0,j}$. Each unit j first computes a weighted sum of its inputs:

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i \ .$$

ACTIVATION FUNCTION

Then it applies an **activation function** g to this sum to derive the output:

$$a_j = g(in_j) = g\left(\sum_{i=0}^{n} w_{i,j} a_i\right) \ . \tag{18.9}$$

[8] A note on notation: for this section, we are forced to suspend our usual conventions. Input attributes are still indexed by i , so that an "external" activation a_i is given by input x_i; but index j will refer to internal units rather than examples. Throughout this section, the mathematical derivations concern a single generic example $\mathbf{x}$, omitting the usual summations over examples to obtain results for the whole data set.

PERCEPTRON
SIGMOID PERCEPTRON

The activation function g is typically either a hard threshold (Figure 18.17(a)), in which case the unit is called a **perceptron**, or a logistic function (Figure 18.17(b)), in which case the term **sigmoid perceptron** is sometimes used. Both of these nonlinear activation function ensure the important property that the entire network of units can represent a nonlinear function (see Exercise 18.26). As mentioned in the discussion of logistic regression (page 725), the logistic activation function has the added advantage of being differentiable.

FEED-FORWARD NETWORK

Having decided on the mathematical model for individual "neurons," the next task is to connect them together to form a network. There are two fundamentally distinct ways to do this. A **feed-forward network** has connections only in one direction—that is, it forms a directed acyclic graph. Every node receives input from "upstream" nodes and delivers output to "downstream" nodes; there are no loops. A feed-forward network represents a function of its current input; thus, it has no internal state other than the weights themselves. A **recurrent network**, on the other hand, feeds its outputs back into its own inputs. This means that the activation levels of the network form a dynamical system that may reach a stable state or exhibit oscillations or even chaotic behavior. Moreover, the response of the network to a given input depends on its initial state, which may depend on previous inputs. Hence, recurrent networks (unlike feed-forward networks) can support short-term memory. This makes them more interesting as models of the brain, but also more difficult to understand. This section will concentrate on feed-forward networks; some pointers for further reading on recurrent networks are given at the end of the chapter.

RECURRENT NETWORK

LAYERS

Feed-forward networks are usually arranged in **layers**, such that each unit receives input only from units in the immediately preceding layer. In the next two subsections, we will look at single-layer networks, in which every unit connects directly from the network's inputs to its outputs, and multilayer networks, which have one or more layers of **hidden units** that are not connected to the outputs of the network. So far in this chapter, we have considered only learning problems with a single output variable y, but neural networks are often used in cases where multiple outputs are appropriate. For example, if we want to train a network to add two input bits, each a 0 or a 1, we will need one output for the sum bit and one for the carry bit. Also, when the learning problem involves classification into more than two classes—for example, when learning to categorize images of handwritten digits—it is common to use one output unit for each class.

HIDDEN UNIT

18.7.2 Single-layer feed-forward neural networks (perceptrons)

PERCEPTRON NETWORK

A network with all the inputs connected directly to the outputs is called a **single-layer neural network**, or a **perceptron network**. Figure 18.20 shows a simple two-input, two-output perceptron network. With such a network, we might hope to learn the two-bit adder function, for example. Here are all the training data we will need:

x_1	x_2	y_3 (carry)	y_4 (sum)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

The first thing to notice is that a perceptron network with m outputs is really m separate networks, because each weight affects only one of the outputs. Thus, there will be m separate training processes. Furthermore, depending on the type of activation function used, the training processes will be either the **perceptron learning rule** (Equation (18.7) on page 724) or gradient descent rule for the **logistic regression** (Equation (18.8) on page 727).

If you try either method on the two-bit-adder data, something interesting happens. Unit 3 learns the carry function easily, but unit 4 completely fails to learn the sum function. No, unit 4 is not defective! The problem is with the sum function itself. We saw in Section 18.6 that linear classifiers (whether hard or soft) can represent linear decision boundaries in the input space. This works fine for the carry function, which is a logical AND (see Figure 18.21(a)). The sum function, however, is an XOR (exclusive OR) of the two inputs. As Figure 18.21(c) illustrates, this function is not linearly separable so the perceptron cannot learn it.

The linearly separable functions constitute just a small fraction of all Boolean functions; Exercise 18.23 asks you to quantify this fraction. The inability of perceptrons to learn even such simple functions as XOR was a significant setback to the nascent neural network

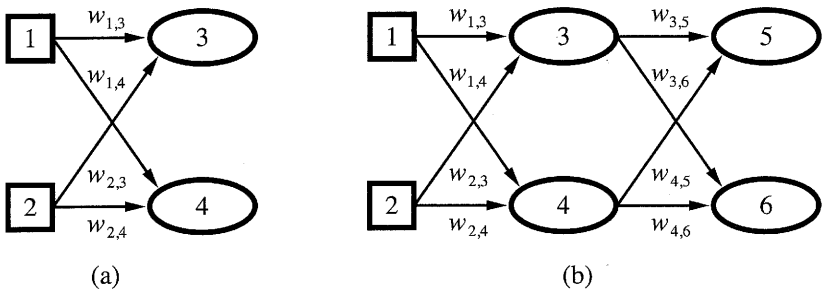

Figure 18.20 (a) A perceptron network with two inputs and two output units. (b) A neural network with two inputs, one hidden layer of two units, and one output unit. Not shown are the dummy inputs and their associated weights.

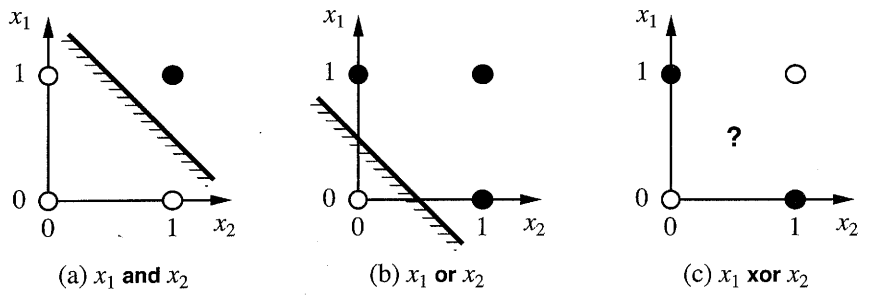

Figure 18.21 Linear separability in threshold perceptrons. Black dots indicate a point in the input space where the value of the function is 1, and white dots indicate a point where the value is 0. The perceptron returns 1 on the region on the non-shaded side of the line. In (c), no such line exists that correctly classifies the inputs.

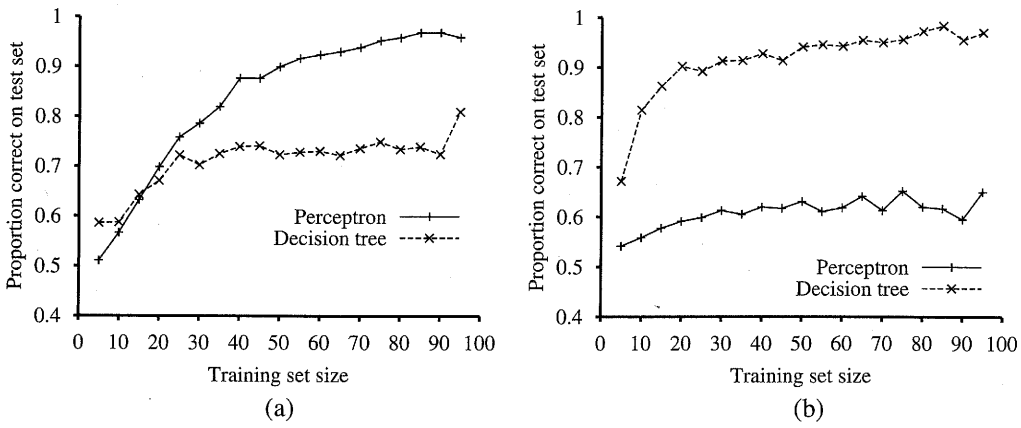

Figure 18.22 Comparing the performance of perceptrons and decision trees. (a) Perceptrons are better at learning the majority function of 11 inputs. (b) Decision trees are better at learning the *WillWait* predicate in the restaurant example.

community in the 1960s. Perceptrons are far from useless, however. Section 18.6.4 noted that logistic regression (i.e., training a sigmoid perceptron) is even today a very popular and effective tool. Moreover, a perceptron can represent some quite "complex" Boolean functions very compactly. 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 $w_i = 1$ and with $w_0 = -n/2$. A decision tree would need exponentially many nodes to represent this function.

Figure 18.22 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, we have the restaurant example. The solution problem is easily represented as a decision tree, but is not linearly separable. The best plane through the data correctly classifies only 65%.

18.7.3 Multilayer feed-forward neural networks

(McCulloch and Pitts, 1943) were well aware that a single threshold unit would not solve all their problems. In fact, their paper proves that such a unit can represent the basic Boolean functions AND, OR, and NOT and then goes on to argue that any desired functionality can be obtained by connecting large numbers of units into (possibly recurrent) networks of arbitrary depth. The problem was that nobody knew how to train such networks.

This turns out to be an easy problem if we think of a network the right way: as a function $h_{\mathbf{w}}(\mathbf{x})$ parameterized by the weights $\mathbf{w}$. Consider the simple network shown in Figure 18.20(b), which has two input units, two hidden units, and two output unit. (In addition, each unit has a dummy input fixed at 1.) Given an input vector $\mathbf{x} = (x_1, x_2)$, the activations

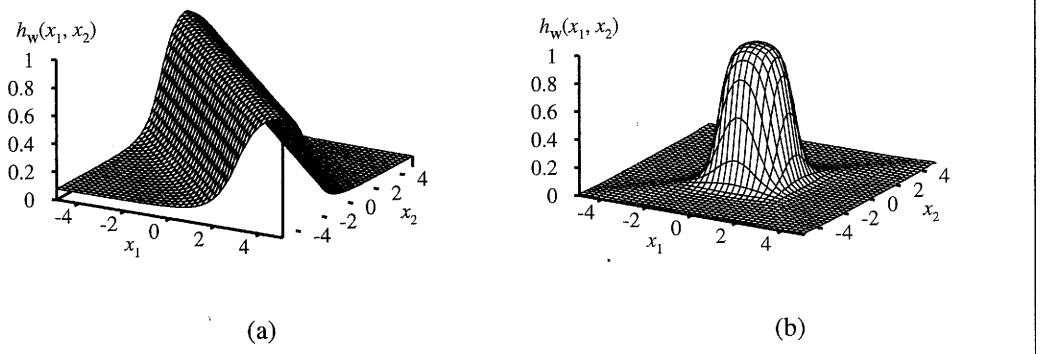

Figure 18.23 (a) The result of combining two opposite-facing soft threshold functions to produce a ridge. (b) The result of combining two ridges to produce a bump.

of the input units are set to $(a_1, a_2) = (x_1, x_2)$. The output at unit 5 is given by

$$\begin{aligned} a_5 &= g(w_{0,5,+} w_{3,5}\, a_3 + w_{4,5}\, a_4) \\ &= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, a_1 + w_{2,3}\, a_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, a_1 + w_{2,4}\, a_2)) \\ &= g(w_{0,5,+} w_{3,5}\, g(w_{0,3} + w_{1,3}\, x_1 + w_{2,3}\, x_2) + w_{4,5}\, g(w_0 4 + w_{1,4}\, x_1 + w_{2,4}\, x_2)). \end{aligned}$$

Thus, we have the output expressed as a function of the inputs and the weights. A similar expression holds for unit 6. As long as we can calculate the derivatives of such expressions with respect to the weights, we can use the gradient-descent loss-minimization method to train the network. Section 18.7.4 shows exactly how to do this. And because the function represented by a network can be highly nonlinear—composed, as it is, of nested nonlinear soft threshold functions—we can see neural networks as a tool for doing **nonlinear regression**.

NONLINEAR REGRESSION

Before delving into learning rules, let us look at the ways in which networks generate complicated functions. First, remember that each unit in a sigmoid network represents a soft threshold in its input space, as shown in Figure 18.17(c) (page 726). With one hidden layer and one output layer, as in Figure 18.20(b), each output unit computes a soft-thresholded linear combination of several such functions. For example, by adding two opposite-facing soft threshold functions and thresholding the result, we can obtain a "ridge" function as shown in Figure 18.23(a). Combining two such ridges at right angles to each other (i.e., combining the outputs from four hidden units), we obtain a "bump" as shown in Figure 18.23(b).

With more hidden units, we can produce more bumps of different sizes in more places. In fact, with a single, sufficiently large hidden layer, it is possible to represent any continuous function of the inputs with arbitrary accuracy; with two layers, even discontinuous functions can be represented.[9] Unfortunately, for any *particular* network structure, it is harder to characterize exactly which functions can be represented and which ones cannot.

[9] The proof is complex, but the main point is that the required number of hidden units grows exponentially with the number of inputs. For example, $2^n/n$ hidden units are needed to encode all Boolean functions of n inputs.

18.7.4 Learning in multilayer networks

First, let us dispense with one minor complication arising in multilayer networks: interactions among the learning problems when the network has multiple outputs. In such cases, we should think of the network as implementing a vector function $\mathbf{h_w}$ rather than a scalar function $h_\mathbf{w}$; for example, the network in Figure 18.20(b) returns a vector $[a_5, a_6]$. Similarly, the target output will be a vector $\mathbf{y}$. Whereas a perceptron network decomposes into m separate learning problems for an m-output problem, this decomposition fails in a multilayer network. For example, both a_5 and a_6 in Figure 18.20(b) depend on all of the input-layer weights, so updates to those weights will depend on errors in both a_5 and a_6. Fortunately, this dependency is very simple in the case of any loss function that is *additive* across the components of the error vector $\mathbf{y} - \mathbf{h_w}(\mathbf{x})$. For the L_2 loss, we have, for any weight w,

$$\frac{\partial}{\partial w} Loss(\mathbf{w}) = \frac{\partial}{\partial w} |\mathbf{y} - \mathbf{h_w}(\mathbf{x})|^2 = \frac{\partial}{\partial w} \sum_k (y_k - a_k)^2 = \sum_k \frac{\partial}{\partial w} (y_k - a_k)^2 \quad (18.10)$$

where the index k ranges over nodes in the output layer. Each term in the final summation is just the gradient of the loss for the kth output, computed as if the other outputs did not exist. Hence, we can decompose an m-output learning problem into m learning problems, provided we remember to add up the gradient contributions from each of them when updating the weights.

The major complication comes from the addition of hidden layers to the network. Whereas the error $\mathbf{y} - \mathbf{h_w}$ at the output layer is clear, the error at the hidden layers seems mysterious because the training data do not say what value the hidden nodes should have. Fortunately, it turns out that we can **back-propagate** the error from the output layer to the hidden layers. The back-propagation process emerges directly from a derivation of the overall error gradient. First, we will describe the process with an intuitive justification; then, we will show the derivation.

BACK-PROPAGATION

At the output layer, the weight-update rule is identical to Equation (18.8). We have multiple output units, so let Err_k be the kth component of the error vector $\mathbf{y} - \mathbf{h_w}$. We will also find it convenient to define a modified error $\Delta_k = Err_k \times g'(in_k)$, so that the weight-update rule becomes

$$w_{j,k} \leftarrow w_{j,k} + \alpha \times a_j \times \Delta_k \,. \quad (18.11)$$

To update 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 Δ_k in each of the output nodes to which it connects. Thus, the Δ_k values are divided according to the strength of the connection between the hidden node and the output node and are propagated back to provide the Δ_j values for the hidden layer. The propagation rule for the Δ values is the following:

$$\Delta_j = g'(in_j) \sum_k w_{j,k} \Delta_k \,. \quad (18.12)$$

```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights w_{i,j}, activation function g
  local variables: Δ, a vector of errors, indexed by network node

  repeat
    for each weight w_{i,j} in network do
      w_{i,j} ← a small random number
    for each example (x, y) in examples do
      /* Propagate the inputs forward to compute the outputs */
      for each node i in the input layer do
        a_i ← x_i
      for ℓ = 2 to L do
        for each node j in layer ℓ do
          in_j ← Σ_i w_{i,j} a_i
          a_j ← g(in_j)
      /* Propagate deltas backward from output layer to input layer */
      for each node j in the output layer do
        Δ[j] ← g'(in_j) × (y_j − a_j)
      for ℓ = L − 1 to 1 do
        for each node i in layer ℓ do
          Δ[i] ← g'(in_i) Σ_j w_{i,j} Δ[j]
      /* Update every weight in network using deltas */
      for each weight w_{i,j} in network do
        w_{i,j} ← w_{i,j} + α × a_i × Δ[j]
  until some stopping criterion is satisfied
  return network
```

Figure 18.24 The back-propagation algorithm for learning in multilayer networks.

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

$$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta_j \ .$$

The back-propagation process can be summarized as follows:

- Compute the Δ 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 Δ values back to the previous layer.
  - Update the weights between the two layers.

The detailed algorithm is shown in Figure 18.24.

For the mathematically inclined, we will now derive the back-propagation equations from first principles. The derivation is quite similar to the gradient calculation for logistic

regression (leading up to Equation (18.8) on page 727), except that we have to use the chain rule more than once.

Following Equation (18.10), we compute just the gradient for $Loss_k = (y_k - a_k)^2$ at the kth output. The gradient of this loss with respect to weights connecting the hidden layer to the output layer will be zero except for weights $w_{j,k}$ that connect to the kth output unit. For those weights, we have

$$\begin{aligned}\frac{\partial Loss_k}{\partial w_{j,k}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{j,k}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{j,k}} \\ &= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{j,k}} = -2(y_k - a_k)g'(in_k)\frac{\partial}{\partial w_{j,k}}\left(\sum_j w_{j,k}a_j\right) \\ &= -2(y_k - a_k)g'(in_k)a_j = -a_j\Delta_k\ ,\end{aligned}$$

with Δ_k defined as before. To obtain the gradient with respect to the $w_{i,j}$ weights connecting the input layer to the hidden ¡layer, we have to expand out the activations a_j and reapply the chain rule. We will show the derivation in gory detail because it is interesting to see how the derivative operator propagates back through the network:

$$\begin{aligned}\frac{\partial Loss_k}{\partial w_{i,j}} &= -2(y_k - a_k)\frac{\partial a_k}{\partial w_{i,j}} = -2(y_k - a_k)\frac{\partial g(in_k)}{\partial w_{i,j}} \\ &= -2(y_k - a_k)g'(in_k)\frac{\partial in_k}{\partial w_{i,j}} = -2\Delta_k\frac{\partial}{\partial w_{i,j}}\left(\sum_j w_{j,k}a_j\right) \\ &= -2\Delta_k w_{j,k}\frac{\partial a_j}{\partial w_{i,j}} = -2\Delta_k w_{j,k}\frac{\partial g(in_j)}{\partial w_{i,j}} \\ &= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial in_j}{\partial w_{i,j}} \\ &= -2\Delta_k w_{j,k}g'(in_j)\frac{\partial}{\partial w_{i,j}}\left(\sum_i w_{i,j}a_i\right) \\ &= -2\Delta_k w_{j,k}g'(in_j)a_i = -a_i\Delta_j\ ,\end{aligned}$$

where Δ_j is defined as before. Thus, we obtain the update rules obtained earlier from intuitive considerations. It is also clear that the process can be continued for networks with more than one hidden layer, which justifies the general algorithm given in Figure 18.24.

Having made it through (or skipped over) all the mathematics, let's see how a single-hidden-layer network performs on the restaurant problem. First, we need to determine the structure of the network. We have 10 attributes describing each example, so we will need 10 input units. Should we have one hidden layer or two? How many nodes in each layer? Should they be fully connected? There is no good theory that will tell us the answer. (See the next section.) As always, we can use cross-validation: try several different structures and see which one works best. It turns out that a network with one hidden layer containing four nodes is about right for this problem. In Figure 18.25, we show two curves. The first is a training curve showing the mean squared error on a given training set of 100 restaurant examples

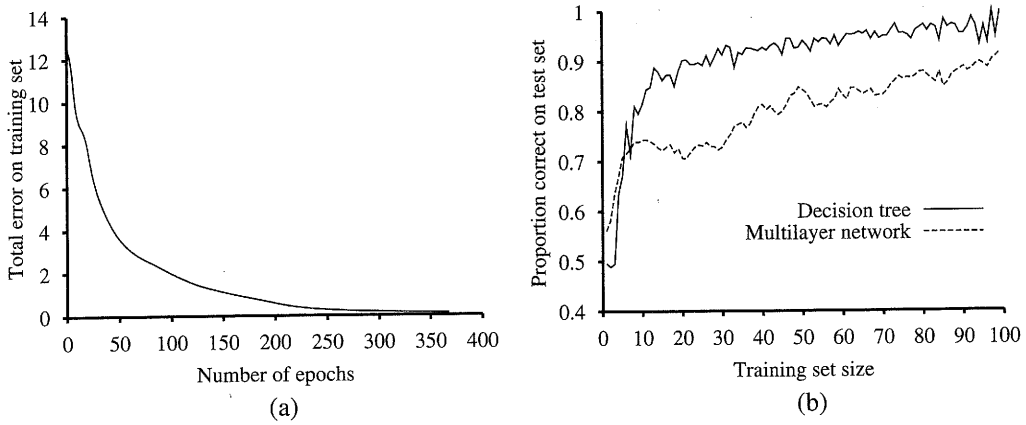

Figure 18.25 (a) Training curve showing the gradual reduction in error as weights are modified over several epochs, for a given set of examples in the restaurant domain. (b) Comparative learning curves showing that decision-tree learning does slightly better on the restaurant problem than back-propagation in a multilayer network.

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. The neural network does learn well, although not quite as fast as decision-tree learning; this is perhaps not surprising, because the data were generated from a simple decision tree in the first place.

Neural networks are capable of far more complex learning tasks of course, although it must be said that a certain amount of twiddling is needed to get the network structure right and to achieve convergence to something close to the global optimum in weight space. There are literally tens of thousands of published applications of neural networks. Section 18.11.1 looks at one such application in more depth.

18.7.5 Learning neural network structures

So far, we have considered the problem of learning weights, given a fixed network structure; just as with Bayesian networks, we also need to understand how to find the best network structure. 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 necessarily generalize well to inputs that have not been seen before.[10] In other words, like all statistical models, neural networks are subject to **overfitting** when there are too many parameters in the model. We saw this in Figure 18.1 (page 696), where the high-parameter models in (b) and (c) fit all the data, but might not generalize as well as the low-parameter models in (a) and (d).

If we stick to fully connected networks, the only choices to be made concern the number

[10] It has been observed that very large networks *do* generalize well *as long as the weights are kept small.* This restriction keeps the activation values in the *linear* region of the sigmoid function $g(x)$ where x is close to zero. This, in turn, means that the network behaves like a linear function (Exercise 18.26) with far fewer parameters.

of hidden layers and their sizes. The usual approach is to try several and keep the best. The **cross-validation** techniques of Chapter 18 are needed if we are to avoid **peeking** at the test set. That is, we choose the network architecture that gives the highest prediction accuracy on the validation sets.

If we want to consider networks that are not fully connected, then we need to find some effective search method through the very large space of possible connection topologies.

OPTIMAL BRAIN DAMAGE

The **optimal brain damage** algorithm begins with a fully connected network and removes connections from it. After the network is trained for the first time, an information-theoretic approach identifies an optimal selection of connections that can be dropped. The network is then retrained, and if its performance has not decreased then the process is repeated. 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.

TILING

One, the **tiling** algorithm, resembles decision-list learning. 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.

18.8 NONPARAMETRIC MODELS

Linear regression and neural networks use the training data to estimate a fixed set of parameters $\mathbf{w}$. That defines our hypothesis $h_{\mathbf{w}}(\mathbf{x})$, and at that point we can throw away the training data, because they are all summarized by $\mathbf{w}$. A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a

PARAMETRIC MODEL

parametric model.

No matter how much data you throw at a parametric model, it won't change its mind about how many parameters it needs. When data sets are small, it makes sense to have a strong restriction on the allowable hypotheses, to avoid overfitting. But when there are thousands or millions or billions of examples to learn from, it seems like a better idea to let the data speak for themselves rather than forcing them to speak through a tiny vector of parameters. If the data say that the correct answer is a very wiggly function, we shouldn't restrict ourselves to linear or slightly wiggly functions.

NONPARAMETRIC MODEL

A **nonparametric model** is one that cannot be characterized by a bounded set of parameters. For example, suppose that each hypothesis we generate simply retains within itself all of the training examples and uses all of them to predict the next example. Such a hypothesis family would be nonparametric because the effective number of parameters is unbounded—it grows with the number of examples. This approach is called **instance-based learning** or **memory-based learning**. The simplest instance-based learning method is **table lookup**: take all the training examples, put them in a lookup table, and then when asked for $h(\mathbf{x})$, see if $\mathbf{x}$ is in the table; if it is, return the corresponding y. The problem with this method is that it does not generalize well: when $\mathbf{x}$ is not in the table all it can do is return some default value.

INSTANCE-BASED LEARNING

TABLE LOOKUP