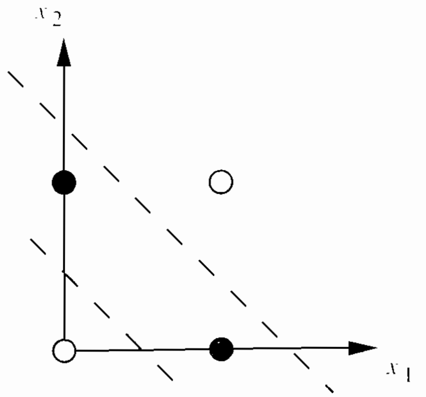

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Figure 18.12: A Classification Problem, XOR, That Is Not Linearly Separable

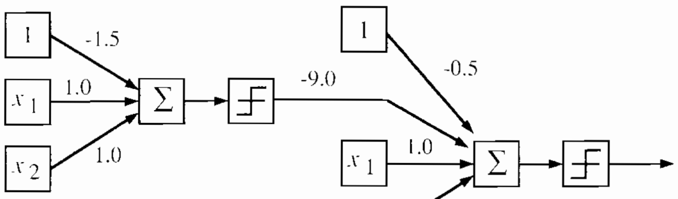

Figure 18.13: A Multilayer Perceptron That Solves the XOR Problem

> The perceptron ...has many features that attract attention: its linearity, its intriguing learning theorem ... there is no reason to suppose that any of these virtues carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile.

Despite the identification of this "important research problem," actual research in perceptron learning came to a halt in the 1970s. The field saw little interest until the 1980s, when several learning procedures for multilayer perceptrons—also called multilayer networks—were proposed. The next few sections are devoted to such learning procedures.

18.2.2 Backpropagation Networks

Rich & Knight, Artificial Intelligence, 1991.

As suggested by Figure 18.8 and the *Perceptrons* critique, the ability to train multilayer networks is an important step in the direction of building intelligent machines from neuronlike components. Let's reflect for a moment on why this is so. Our goal is to take a relatively amorphous mass of neuronlike elements and teach it to perform useful tasks. We would like it to be fast and resistant to damage. We would like it to generalize from the inputs it sees. We would like to build these neural masses on a very large scale, and we would like them to be able to learn efficiently. Perceptrons got us part of the way there, but we saw that they were too weak computationally. So we turn to more complex, multilayer networks.

What can a multilayer network compute? The simple answer is: *anything*! Given a set of inputs, we can use summation-threshold units as simple AND, OR, and NOT gates by appropriately setting the threshold and connection weights. We know that we can build any arbitrary combinational circuit out of those basic logical units. In fact, if we are allowed to use feedback loops, we can build a general-purpose computer with them.

The major problem is learning. The knowledge representation system employed by neural nets is quite opaque: the nets *must* learn their own representations because programming them by hand is impossible. Perceptrons had the nice property that whatever they could compute, they could learn to compute. Does this property extend to multilayer networks? The answer is yes, sort of. Backpropagation is a step in that direction.

It will be useful to deal first with a subclass of multilayer networks, namely *fully connected, layered, feedforward* networks. A sample of such a network is shown in Figure 18.14. In this figure, x_i, h_i, and o_i represent unit activation levels of input, *hidden*, and output units. Weights on connections between the input and hidden layers are denoted here by $w1_{ij}$, while weights on connections between the hidden and output layers are denoted by $w2_{ij}$. This network has three layers, although it is possible and sometimes useful to have more. Each unit in one layer is connected in the forward direction to every unit in the next layer. Activations flow from the input layer through the hidden layer, then on to the output layer. As usual, the knowledge of the network is encoded in the weights on connections between units. In contrast to the parallel relaxation method used by Hopfield nets, backpropagation networks perform a simpler computation. Because activations flow in only one direction, there is no need for an iterative relaxation process. The activation levels of the units in the output layer determine the output of the network.

The existence of hidden units allows the network to develop complex feature detectors, or internal representations. Figure 18.15 shows the application of a three layer network to the problem of recognizing digits. The two-dimensional grid containing the numeral "7" forms the input layer. A single hidden unit might be strongly activated by a horizontal line in the input, or perhaps a diagonal. The important thing to note is that the behavior of these hidden units is automatically learned, not preprogrammed. In Figure 18.15, the input grid appears to be laid out in two dimensions, but the fully connected network is unaware of this 2-D structure. Because this structure can be important, many networks permit their hidden units to maintain only local connections to the input layer (e.g., a different 4 by 4 subgrid for each hidden unit).

The hope in attacking problems like handwritten character recognition is that the neural network will not only learn to classify the inputs it is trained on but that it will *generalize* and be able to classify inputs that it has not yet seen. We return to generalization in the next section.

A reasonable question at this point is: "All neural nets seem to be able to do

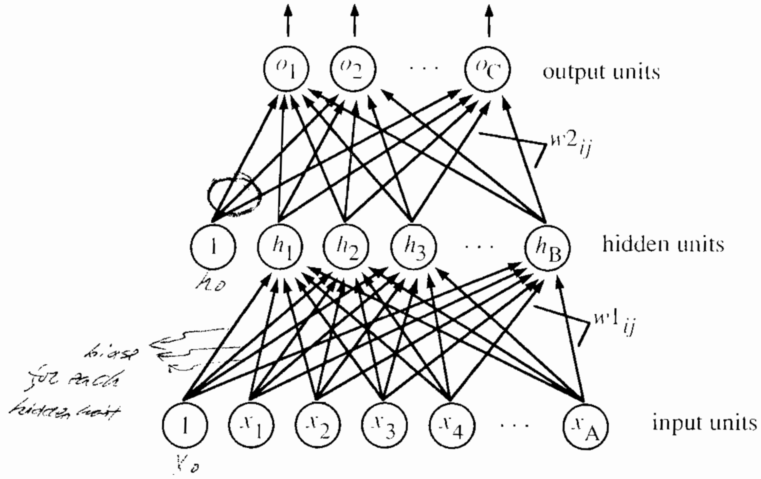

Figure 18.14: A Multilayer Network

is classification. Hard AI problems such as planning, natural language parsing, and theorem proving are not simply classification tasks, so how do connectionist models address these problems?" Most of the problems we see in this chapter are indeed classification problems, because these are the problems that neural networks are best suited to handle at present. A major limitation of current network formalisms is how they deal with phenomena that involve time. This limitation is lifted to some degree in work on recurrent networks (see Section 18.4), but the problems are still severe. Hence, we concentrate on classification problems for now.

Let's now return to backpropagation networks. The unit in a backpropagation network requires a slightly different activation function from the perceptron. Both functions are shown in Figure 18.16. A backpropagation unit still sums up its weighted inputs, but unlike the perceptron, it produces a real value between 0 and 1 as output, based on a sigmoid (or S-shaped) function, which is continuous and differentiable, as required by the backpropagation algorithm. Let *sum* be the weighted sum of the inputs to a unit. The equation for the unit's output is given by:

$$output = \frac{1}{1 + e^{-sum}}$$

Notice that if the sum is 0, the output is 0.5 (in contrast to the perceptron, where it must be either 0 or 1). As the sum gets larger, the output approaches 1. As the sum gets smaller, on the other hand, the output approaches 0.

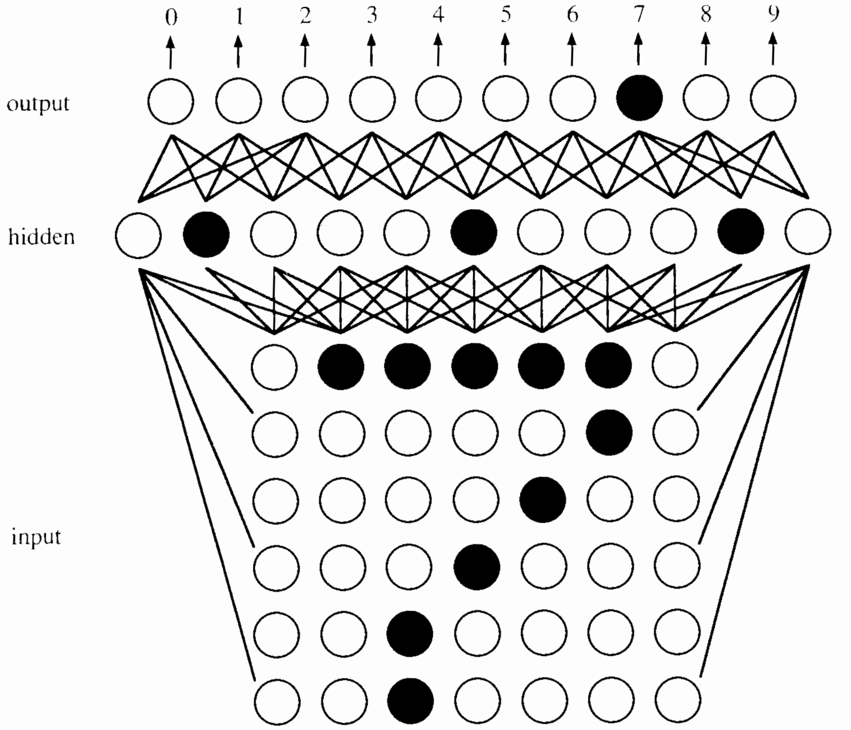

Figure 18.15: Using a Multilayer Network to Learn to Classify Handwritten Digits

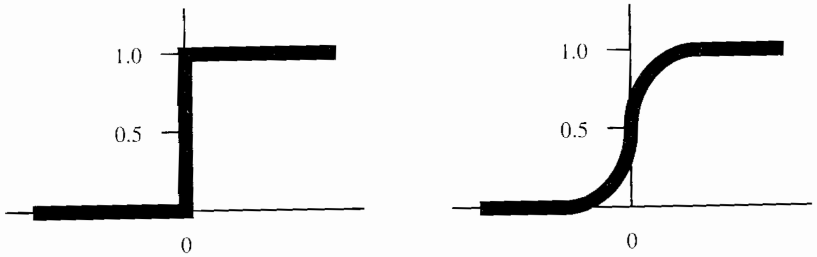

Figure 18.16: The Stepwise Activation Function of the Perceptron (*left*), and the Sigmoid Activation Function of the Backpropagation Unit (*right*)

Like a perceptron, a backpropagation network typically starts out with a random set of weights. The network adjusts its weights each time it sees an input-output pair. Each pair requires two stages: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. During the backward pass, the network's actual output (from the forward pass) is compared with the target output and error estimates are computed for the output units. The weights connected to the output units can be adjusted in order to reduce those errors. We can then use the error estimates of the output units to derive error estimates for the units in the hidden layers. Finally, errors are propagated back to the connections stemming from the input units.

Unlike the perceptron learning algorithm of the last section, the backpropagation algorithm usually updates its weights incrementally, after seeing each input-output pair. After it has seen all the input-output pairs (and adjusted its weights that many times), we say that one *epoch* has been completed. Training a backpropagation network usually requires many epochs.

Refer back to Figure 18.14 for the basic structure on which the following algorithm is based.

Algorithm: Backpropagation

Given: A set of input-output vector pairs.

Compute: A set of weights for a three-layer network that maps inputs onto corresponding outputs.

1. Let A be the number of units in the input layer, as determined by the length of the training input vectors. Let C be the number of units in the output layer. Now choose B, the number of units in the hidden layer.[3] As shown in Figure 18.14, the input and hidden layers each have an extra unit used for thresholding; therefore, the units in these layers will sometimes be indexed by the ranges $(0, \ldots, A)$ and $(0, \ldots, B)$. We denote the activation levels of the units in the input layer by x_j, in the hidden layer by h_j, and in the output layer by o_j. Weights connecting the input layer to the hidden layer are denoted by $w1_{ij}$, where the subscript i indexes the input units and j indexes the hidden units. Likewise, weights connecting the hidden layer to the output layer are denoted by $w2_{ij}$, with i indexing to hidden units and j indexing output units.

2. Initialize the weights in the network. Each weight should be set randomly to a number between -0.1 and 0.1.

$$w1_{ij} = random(-0.1, 0.1) \quad \text{for all} \quad i = 0, \ldots, A, \ j = 1, \ldots, B$$

$$w2_{ij} = random(-0.1, 0.1) \quad \text{for all} \quad i = 0, \ldots, B, \ j = 1, \ldots, C$$

3. Initialize the activations of the thresholding units. The values of these thresholding units should never change.

$$\left.\begin{aligned} x_0 &= 1.0 \\ h_0 &= 1.0 \end{aligned}\right\}$$

4. Choose an input-output pair. Suppose the input vector is x_i and the target output vector is y_i. Assign activation levels to the input units.

5. Propagate the activations from the units in the input layer to the units in the hidden layer using the activation function of Figure 18.16:

input → hidden

$$h_j = \frac{1}{1 + e^{-\sum_{i=0}^{A} w1_{ij}x_i}} \quad \text{for all} \quad j = 1, \ldots, B$$

Note that i ranges from 0 to A. $w1_{0j}$ is the thresholding weight for hidden unit j (its propensity to fire irrespective of its inputs). x_0 is always 1.0. → bias

6. Propagate the activations from the units in the hidden layer to the units in the output layer.

hidden → output

$$o_j = \frac{1}{1 + e^{-\sum_{i=0}^{B} w2_{ij}h_i}} \quad \text{for all} \quad j = 1, \ldots, C$$

Again, the thresholding weight $w2_{0j}$ for output unit j plays a role in the weighted summation. h_0 is always 1.0.

7. Compute the errors[4] of the units in the output layer, denoted $\delta 2_j$. Errors are based on the network's actual output (o_j) and the target output (y_j).

Error estimates for output

$$\delta 2_j = o_j(1 - o_j)(y_j - o_j) \quad \text{for all} \quad j = 1, \ldots, C$$

8. Compute the errors of the units in the hidden layer, denoted $\delta 1_j$.

Error estimates for hidden

$$\delta 1_j = h_j(1 - h_j) \sum_{i=1}^{C} \delta 2_i \cdot w2_{ji} \quad \text{for all} \quad j = \boxed{1, \ldots, B}$$

9. Adjust the weights between the hidden layer and output layer.[5] The learning rate is denoted η; its function is the same as in perceptron learning. A reasonable value of η is 0.35.

[3] Successful large-scale networks have used topologies like 203-80-26 [Sejnowski and Rosenberg, 1987], 960-9-45 [Pomerleau, 1989], and 459-24-24-1 [Tesauro and Sejnowski, 1989]. A larger hidden layer results in a more powerful network, but too much power may be undesirable (see Section 18.2.3).

[4] The error formula is related to the derivative of the activation function. The mathematical derivation behind the backpropagation learning algorithm is beyond the scope of this book.

[5] Again, we omit the details of the derivation. The basic idea is that each hidden unit tries to minimize the errors of output units to which it connects.

$$\Delta w2_{ij} = \eta \cdot \delta 2_j \cdot h_i \quad \text{for all} \quad i = 0, \ldots, B,\ j = 1, \ldots, C$$

10. Adjust the weights between the input layer and the hidden layer.

$$\Delta w1_{ij} = \eta \cdot \delta 1_j \cdot x_i \quad \text{for all} \quad i = 0, \ldots, A,\ j = 1, \ldots, B$$

11. Go to step 4 and repeat. When all the input-output pairs have been presented to the network, one epoch has been completed. Repeat steps 4 to 10 for as many epochs as desired.

The algorithm generalizes straightforwardly to networks of more than three layers.[6] For each extra hidden layer, insert a forward propagation step between steps 6 and 7, an error computation step between steps 8 and 9, and a weight adjustment step between steps 10 and 11. Error computation for hidden units should use the equation in step 8, but with i ranging over the units in the next layer, not necessarily the output layer.

The speed of learning can be increased by modifying the weight modification steps 9 and 10 to include a momentum term α. The weight update formulas become:

$$\Delta w2_{ij}(t+1) = \eta \cdot \delta 2_j \cdot h_i + \alpha \Delta w2_{ij}(t)$$

$$\Delta w1_{ij}(t+1) = \eta \cdot \delta 1_j \cdot x_i + \alpha \Delta w1_{ij}(t)$$

where h_i, x_i, $\delta 1_j$ and $\delta 2_j$ are measured at time $t+1$. $\Delta w_{ij}(t)$ is the change the weight experienced during the previous forward-backward pass. If α is set to 0.9 or so, learning speed is improved.[7]

Recall that the activation function has a sigmoid shape. Since infinite weights would be required for the actual outputs of the network to reach 0.0 and 1.0, binary target outputs (the y_j's of steps 4 and 7 above) are usually given as 0.1 and 0.9 instead. The sigmoid is required by backpropagation because the derivation of the weight update rule requires that the activation function be continuous and differentiable.

The derivation of the weight update rule is more complex than the derivation of the fixed-increment update rule for perceptrons, but the idea is much the same. There is an error function that defines a surface over weight space, and the weights are modified in the direction of the gradient of the surface. See Rumelhart *et al.* [1986] for details. Interestingly, the error surface for multilayer nets is more complex than the error surface for perceptrons. One notable difference is the existence of *local minima*. Recall the bowl-shaped space we used to explain perceptron learning (Figure 18.10). As we

modified weights, we moved in the direction of the bottom of the bowl; eventually, we reached it. A backpropagation network, however, may slide down the error surface into a set of weights that does not solve the problem it is being trained on. If that set of weights is at a local minimum, the network will never reach the optimal set of weights. Thus, we have no analogue of the perceptron convergence theorem for backpropagation networks.

There are several methods of overcoming the problem of local minima. The momentum factor α, which tends to keep the weight changes moving in the same direction, allows the algorithm to skip over small minima. *Simulated annealing*, discussed later in Section 18.2.4, is also useful. Finally, adjusting the shape of a unit's activation function can have an effect on the network's susceptibility to local minima.

Fortunately, backpropagation networks rarely slip into local minima. It turns out that, especially in larger networks, the high-dimensional weight space provides plenty of degrees of freedom for the algorithm. The lack of a convergence theorem is not a problem in practice. However, this pleasant feature of backpropagation was not discovered until recently, when digital computers became fast enough to support large-scale simulations of neural networks. The backpropagation algorithm was actually derived independently by a number of researchers in the past, but it was discarded as many times because of the potential problems with local minima. In the days before fast digital computers, researchers could only judge their ideas by proving theorems about them, and they had no idea that local minima would turn out to be rare in practice. The modern form of backpropagation is often credited to Werbos [1974], LeCun [1985], Parker [1985], and Rumelhart *et al.* [1986].

Backpropagation networks are not without real problems, however, with the most serious being the slow speed of learning. Even simple tasks require extensive training periods. The XOR problem, for example, involves only five units and nine weights, but it can require many, many passes through the four training cases before the weights converge, especially if the learning parameters are not carefully tuned. Also, simple backpropagation does not scale up very well. The number of training examples required is superlinear in the size of the network.

Since backpropagation is inherently a parallel, distributed algorithm, the idea of improving speed by building special-purpose backpropagation hardware is attractive. However, fast new variations of backpropagation and other learning algorithms appear frequently in the literature, e.g., Fahlman [1988]. By the time an algorithm is transformed into hardware and embedded in a computer system, the algorithm is likely to be obsolete.

18.2.3 Generalization

If all possible inputs and outputs are shown to a backpropagation network, the network will (probably, eventually) find a set of weights that maps the inputs onto the outputs. For many AI problems, however, it is impossible to give all possible inputs. Consider face recognition and character recognition. There are an infinite number of orientations and expressions to a face, and an infinite number of fonts and sizes for a character, yet humans learn to classify these objects easily from only a few examples. We would hope that our networks would do the same. And, in fact, backpropagation shows promise as a generalization mechanism. If we work in a domain (such as the classification domains just discussed) where similar inputs get mapped onto similar outputs, backpropagation

[6] A network with one hidden layer can compute any function that a network with many hidden layers can compute: with an exponential number of hidden units, one unit could be assigned to every possible input pattern. However, learning is sometimes faster with multiple hidden layers, especially if the input is highly nonlinear, i.e., hard to separate with a series of straight lines.

[7] Empirically, best results have come from letting α be *zero* for the first few training passes, then increasing it to 0.9 for the rest of training. This process first gives the algorithm some time to find a good general direction, and then moves it in that direction with some extra speed.