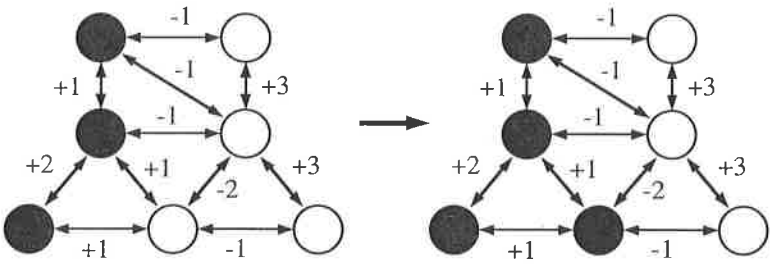


Figure 18.4: A Simplified View of What a Hopfield Net Computes

automatic learning.

In the next section, we look closely at learning in several neural network models, including perceptrons, backpropagation networks, and Boltzmann machines, a variation of Hopfield networks. After this, we investigate some applications of connectionism. Then we see how networks with feedback can deal with temporal processes and how distributed representations can be made efficient.

18.2 Learning in Neural Networks

18.2.1 Perceptrons

The *perceptron*, an invention of Rosenblatt [1962], was one of the earliest neural network models. A perceptron models a neuron by taking a weighted sum of its inputs and sending the output 1 if the sum is greater than some adjustable threshold value (otherwise it sends 0). Figure 18.5 shows the device. Notice that in a perceptron, unlike a Hopfield network, connections are unidirectional.

The inputs $(x_1, x_2, \ldots, x_n)$ and connection weights $(w_1, w_2, \ldots, w_n)$ in the figure are typically real values, both positive and negative. If the presence of some feature x_i tends to cause the perceptron to fire, the weight w_i will be positive; if the feature x_i inhibits the perceptron, the weight w_i will be negative. The perceptron itself consists of the weights, the summation processor, and the adjustable threshold processor. Learning is a process of modifying the values of the weights and the threshold. It is convenient to implement the threshold as just another weight w_0, as in Figure 18.6. This weight can be thought of as the propensity of the perceptron to fire irrespective of its inputs. The perceptron of Figure 18.6 fires if the weighted sum is greater than zero.

A perceptron computes a binary function of its input. Several perceptrons can be combined to compute more complex functions, as shown in Figure 18.7.

Such a group of perceptrons can be trained on sample input-output pairs until it learns to compute the correct function. The amazing property of perceptron learning

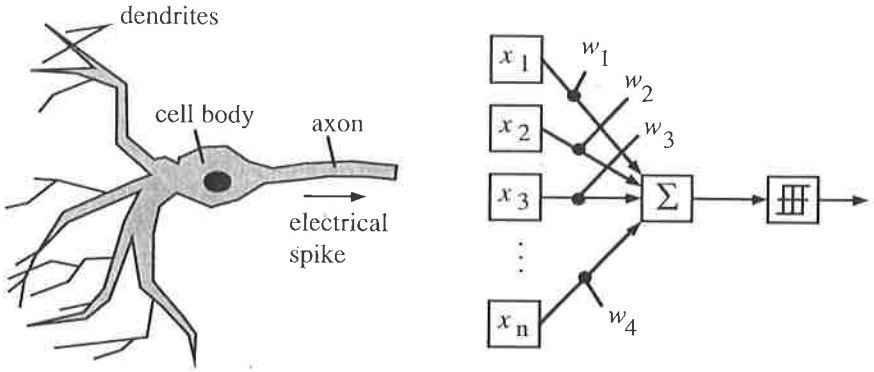


Figure 18.5: A Neuron and a Perceptron

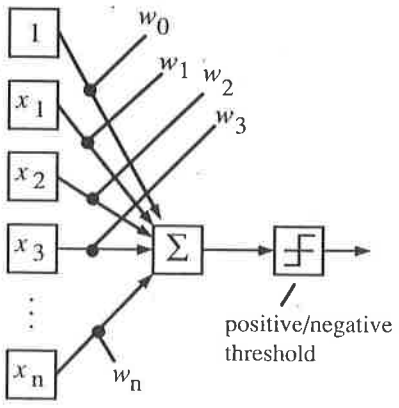


Figure 18.6: Perceptron with Adjustable Threshold Implemented as Additional Weight

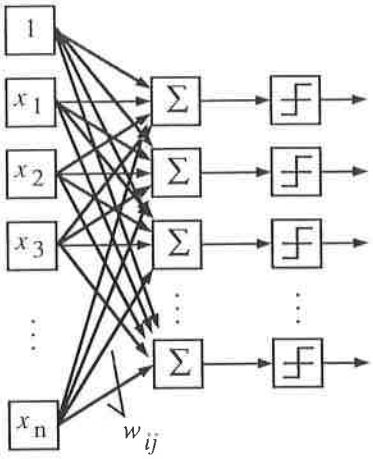


Figure 18.7: A Perceptron with Many Inputs and Many Outputs

is this: Whatever a perceptron can compute, it can *learn* to compute! We demonstrate this in a moment. At the time perceptrons were invented, many people speculated that intelligent systems could be constructed out of perceptrons (see Figure 18.8).

Since the perceptrons of Figure 18.7 are independent of one another, they can be separately trained. So let us concentrate on what a single perceptron can learn to do. Consider the pattern classification problem shown in Figure 18.9. This problem is *linearly separable*, because we can draw a line that separates one class from another. Given values for x_1 and x_2, we want to train a perceptron to output 1 if it thinks the input belongs to the class of white dots and 0 if it thinks the input belongs to the class of black dots. Pattern classification is very similar to *concept learning*, which was discussed in Chapter 17. We have no explicit rule to guide us; we must induce a rule from a set of training instances. We now see how perceptrons can learn to solve such problems.

First, it is necessary to take a close look at what the perceptron computes. Let $\vec{x}$ be an input vector $(x_1, x_2, \ldots, x_n)$. Notice that the weighted summation function $g(x)$ and the output function $o(x)$ can be defined as:

$$g(x) = \sum_{i=0}^{n} w_i x_i$$

$$o(x) = \begin{cases} 1 & \text{if } g(x) > 0 \\ 0 & \text{if } g(x) < 0 \end{cases}$$

Consider the case where we have only two inputs (as in Figure 18.9). Then:

$$g(x) = w_0 + w_1 x_1 + w_2 x_2$$

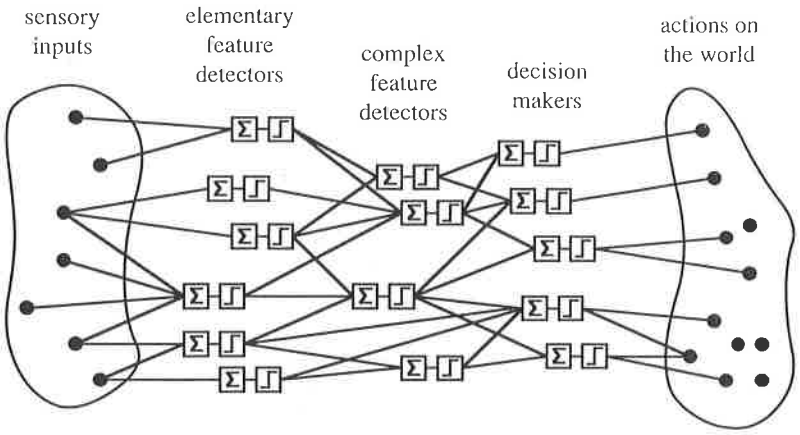


Figure 18.8: An Early Notion of an Intelligent System Built from Trainable Perceptrons

If $g(x)$ is exactly zero, the perceptron cannot decide whether to fire. A slight change in inputs could cause the device to go either way. If we solve the equation $g(x) = 0$, we get the equation for a line:

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{w_0}{w_2}$$

The location of the line is completely determined by the weights w_0, w_1, and w_2. If an input vector lies on one side of the line, the perceptron will output 1; if it lies on the other side, the perceptron will output 0. A line that correctly separates the training instances corresponds to a perfectly functioning perceptron. Such a line is called a *decision surface*. In perceptrons with many inputs, the decision surface will be a hyperplane through the multidimensional space of possible input vectors. The problem of *learning* is one of locating an appropriate decision surface.

We present a formal learning algorithm later. For now, consider the informal rule:

> If the perceptron fires when it should not fire, make each w_i smaller by an amount proportional to x_i. If the perceptron fails to fire when it should fire, make each w_i larger by a similar amount.

Suppose we want to train a three-input perceptron to fire only when its first input is on. If the perceptron fails to fire in the presence of an active x_1, we will increase w_1 (and we may increase other weights). If the perceptron fires incorrectly, we will end up decreasing weights that are not w_1. (We will never decrease w_1 because undesired firings only occur when x_1 is 0, which forces the proportional change in w_1 also to be 0.) In addition, w_0 will find a value based on the total number of incorrect firings versus incorrect misfirings. Soon, w_1 will become large enough to overpower w_0, while w_2 and w_3 will not be powerful enough to fire the perceptron, even in the presence of both x_2 and x_3.

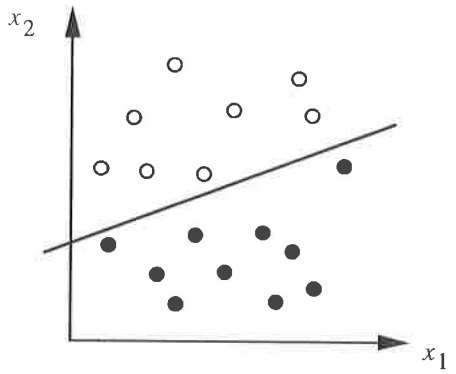


Figure 18.9: A Linearly Separable Pattern Classification Problem

Now let us return to the functions $g(x)$ and $o(x)$. While the sign of $g(x)$ is critical to determining whether the perceptron will fire, the magnitude is also important. The absolute value of $g(x)$ tells how *far* a given input vector $\vec{x}$ lies from the decision surface. This gives us a way of characterizing how good a set of weights is. Let $\vec{w}$ be the weight vector $(w_0, w_1, \ldots, w_n)$, and let X be the subset of training instances *misclassified* by the current set of weights. Then define the *perceptron criterion function*, $J(\vec{w})$, to be the sum of the distances of the misclassified input vectors from the decision surface:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \left| \sum_{i=0}^{n} w_i x_i \right| = \sum_{\vec{x} \in X} |\vec{w}\vec{x}|$$

To create a better set of weights than the current set, we would like to reduce $J(\vec{w})$. Ultimately, if all inputs are classified correctly, $J(\vec{w}) = 0$.

How do we go about minimizing $J(\vec{w})$? We can use a form of local-search hill climbing known as *gradient descent*. We have already seen in Chapter 3 how we can use hill-climbing strategies in symbolic AI systems. For our current purposes, think of $J(\vec{w})$ as defining a surface in the space of all possible weights. Such a surface might look like the one in Figure 18.10.

In the figure, weight w_0 should be part of the weight space but is omitted here because it is easier to visualize J in only three dimensions. Now, some of the weight vectors constitute solutions, in that a perceptron with such a weight vector will classify all its inputs correctly. Note that there are an infinite number of solution vectors. For any solution vector $\vec{w}_s$, we know that $J(\vec{w}_s) = 0$. Suppose we begin with a random weight vector $\vec{w}$ that is not a solution vector. We want to slide down the J surface. There is a mathematical method for doing this—we compute the gradient of the function $J(\vec{w})$. Before we derive the gradient function, we reformulate the perceptron criterion function to remove the absolute value sign:

$$J(\vec{w}) = \sum_{\vec{x} \in X} \vec{w} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

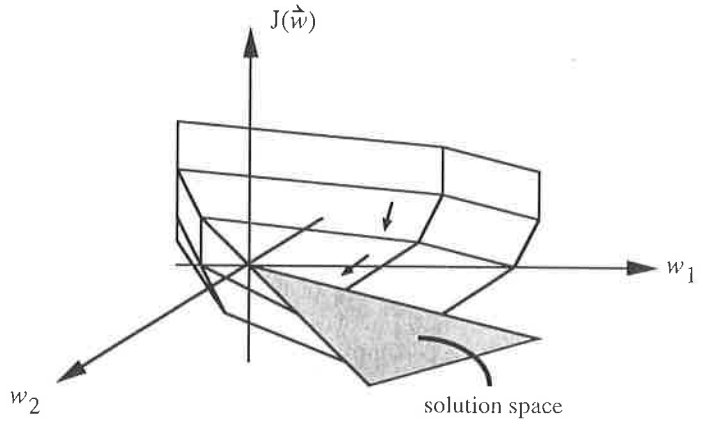


Figure 18.10: Adjusting the Weights by Gradient Descent, Minimizing $J(\vec{w})$

Recall that X is the set of misclassified input vectors.

Now, here is ∇J, the gradient of $J(\vec{w})$ with respect to the weight space:

$$\nabla J(\vec{w}) = \sum_{\vec{x} \in X} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

The gradient is a vector that tells us the direction to move in the weight space in order to reduce $J(\vec{w})$. In order to find a solution weight vector, we simply change the weights in the direction of the gradient, recompute $J(\vec{w})$, recompute the new gradient, and iterate until $J(\vec{w}) = 0$. The rule for updating the weights at time $t + 1$ is:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \nabla J$$

Or in expanded form:

$$\vec{w}_{t+1} = \vec{w}_t + \eta \sum_{\vec{x} \in X} \begin{cases} \vec{x} \text{ if } \vec{x} \text{ is misclassified as a negative example} \\ -\vec{x} \text{ if } \vec{x} \text{ is misclassified as a positive example} \end{cases}$$

η is a scale factor that tells us how far to move in the direction of the gradient. A small η will lead to slower learning, but a large η may cause a move through weight space that "overshoots" the solution vector. Taking η to be a constant gives us what is usually called the "fixed-increment perceptron learning algorithm":

Algorithm: Fixed-Increment Perceptron Learning

Given: A classification problem with n input features $(x_1, x_2, \ldots, x_n)$ and two output classes.

Compute: A set of weights $(w_0, w_1, w_2, \ldots, w_n)$ that will cause a perceptron to fire whenever the input falls into the first output class.

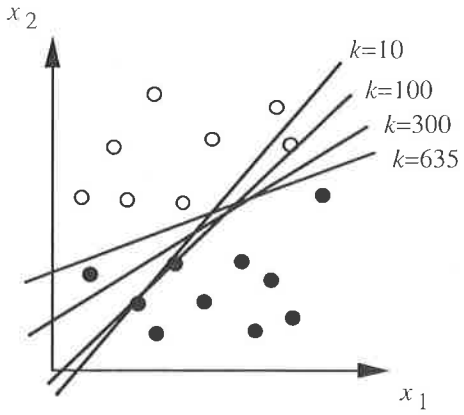
1. Create a perceptron with $n + 1$ inputs and $n + 1$ weights, where the extra input x_0 is always set to 1.
2. Initialize the weights $(w_0, w_1, \ldots, w_n)$ to random real values.
3. Iterate through the training set, collecting all examples *misclassified* by the current set of weights.
4. If all examples are classified correctly, output the weights and quit.
5. Otherwise, compute the vector sum S of the misclassified input vectors, where each vector has the form $(x_0, x_1, \ldots, x_n)$. In creating the sum, add to S a vector $\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly *fails to fire*, but add vector $-\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly *fires*. Multiply the sum by a scale factor η.
6. Modify the weights $(w_0, w_1, \ldots, w_n)$ by adding the elements of the vector S to them. Go to step 3.

The perceptron learning algorithm is a search algorithm. It begins in a random initial state and finds a solution state. The search space is simply all possible assignments of real values to the weights of the perceptron, and the search strategy is gradient descent. Gradient descent is identical to the hill-climbing strategy described in Chapter 3, except that we view good as "down" rather than "up."

So far, we have seen two search methods employed by neural networks, *gradient descent* in perceptrons and *parallel relaxation* in Hopfield networks. It is important to understand the relation between the two. Parallel relaxation is a problem-solving strategy, analogous to state space search in symbolic AI. Gradient descent is a learning strategy, analogous to techniques such as version spaces. In both symbolic and connectionist AI, learning is viewed as a type of problem solving, and this is why search is useful in learning. But the ultimate goal of learning is to get a system into a position where it can solve problems better. Do not confuse learning algorithms with others.

The *perceptron convergence theorem*, due to Rosenblatt [1962], guarantees that the perceptron will find a solution state, i.e., it will learn to classify any linearly separable set of inputs. In other words, the theorem shows that in the weight space, there are no local minima that do not correspond to the global minimum. Figure 18.11 shows a perceptron learning to classify the instances of Figure 18.9. Remember that every set of weights specifies some decision surface, in this case some two-dimensional line. In the figure, k is the number of passes through the training data, i.e., the number of iterations of steps 3 through 6 of the fixed-increment perceptron learning algorithm.

The introduction of perceptrons in the late 1950s created a great deal of excitement. Here was a device that strongly resembled a neuron and for which well-defined learning algorithms were available. There was much speculation about how intelligent systems could be constructed from perceptron building blocks. In their book *Perceptrons*, Minsky and Papert [1969] put an end to such speculation by analyzing the computational capabilities of the devices. They noticed that while the convergence theorem guaranteed correct classification of linearly separable data, most problems do not supply such nice data. Indeed, the perceptron is incapable of learning to solve some very simple problems. One example given by Minsky and Papert is the exclusive-or (XOR) problem: Given



k	w_0	w_1	w_2
10	0.41	-0.17	0.14
100	0.22	-0.14	0.11
300	-0.10	-0.08	0.07
635	-0.49	-0.10	0.14

Figure 18.11: A Perceptron Learning to Solve a Classification Problem

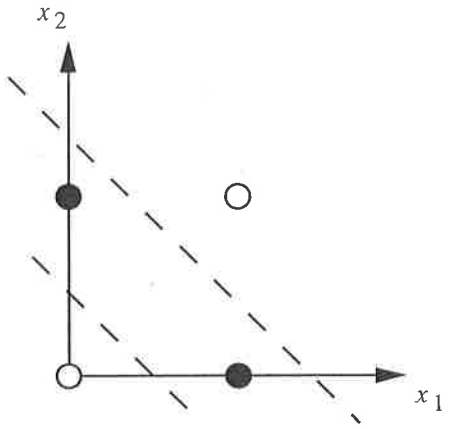
two binary inputs, output 1 if *exactly* one of the inputs is on and output 0 otherwise. We can view XOR as a pattern classification problem in which there are four patterns and two possible outputs (see Figure 18.12).

The perceptron cannot learn a linear decision surface to separate these different outputs, *because no such decision surface exists.* No single line can separate the 1 outputs from the 0 outputs. Minsky and Papert gave a number of problems with this property including telling whether a line drawing is connected, and separating figure from ground in a picture. Notice that the deficiency here is not in the perceptron learning algorithm, but in the way the perceptron represents knowledge.

If we could draw an elliptical decision surface, we could encircle the two "1" outputs in the XOR space. However, perceptrons are incapable of modeling such surfaces. Another idea is to employ two separate line-drawing stages. We could draw one line to isolate the point ($x_1 = 1, x_2 = 1$) and then another line to divide the remaining three points into two categories. Using this idea, we can construct a "multilayer" perceptron (a series of perceptrons) to solve the problem. Such a device is shown in Figure 18.13.

Note how the output of the first perceptron serves as one of the inputs to the second perceptron, with a large, negatively weighted connection. If the first perceptron sees the input ($x_1 = 1, x_2 = 1$), it will send a massive inhibitory pulse to the second perceptron, causing that unit to output 0 regardless of its other inputs. If either of the inputs is 0, the second perceptron gets no inhibition from the first perceptron, and it outputs 1 if either of the inputs is 1.

The use of multilayer perceptrons, then, solves our knowledge representation problem. However, it introduces a serious learning problem: The convergence theorem does not extend to multilayer perceptrons. The perceptron learning algorithm can correctly adjust weights between inputs and outputs, but it cannot adjust weights between perceptrons. In Figure 18.13, the inhibitory weight "−9.0" was hand-coded, not learned. At the time *Perceptrons* was published, no one knew how multilayer perceptrons could be made to learn. In fact, Minsky and Papert speculated:



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

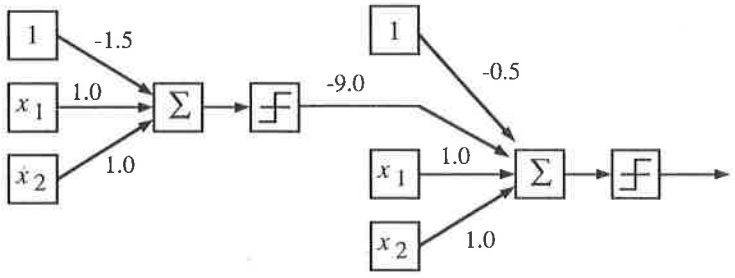


Figure 18.13: A Multilayer Perceptron That Solves the XOR Problem

> 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

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