CHAPTER

18

CONNECTIONIST MODELS

The brain struggling to understand the brain is society trying to explain itself.

—Colin Blakemore
(1944-), British neurobiologist

In our quest to build intelligent machines, we have but one naturally occurring model: the human brain. One obvious idea for AI, then, is to simulate the functioning of the brain directly on a computer. Indeed, the idea of building an intelligent machine out of artificial neurons has been around for quite some time. Some early results on brainlike mechanisms were achieved by McCulloch and Pitts [1943], and other researchers pursued this notion through the next two decades, e.g., Ashby [1952], Minsky [1954], Minsky and Selfridge [1961], Block [1962], and Rosenblatt [1962]. Research in neural networks came to virtual halt in the 1970s, however, when the networks under study were shown to be very weak computationally. Recently, there has been a resurgence of interest in neural networks. There are several reasons for this, including the appearance of faster digital computers on which to simulate larger networks, the interest in building massively parallel computers, and, most important, the discovery of new neural network architectures and powerful learning algorithms.

The new neural network architectures have been dubbed "connectionist' architectures. For the most part, these architectures are not meant to duplicate the operation of the human brain, but rather to receive inspiration from known facts about how the brain works. They are characterized by having:

- A large number of very simple neuronlike processing elements.
- A large number or weighted connections between the elements. The weights on the connections encode the knowledge of a network.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

Connectionist researchers conjecture that thinking about computation in terms of the "brain metaphor" rather than the "digital computer metaphor" will lead to insights into the nature of intelligent behavior.

Computers are capable of amazing feats. They can effortlessly store vast quantities of information. Their circuits operate in nanoseconds. They can perform extensive arithmetic calculations without error. Humans cannot approach these capabilities. On the other hand, humans routinely perform "simple" tasks such as walking, talking, and commonsense reasoning. Current AI systems cannot do any of these things better than

humans can. Why not? Perhaps the structure of the brain is somehow suited to these tasks and not suited to tasks such as high-speed arithmetic calculation. Working under constraints similar to those of the brain may make traditional computation more difficult, but it may lead to solutions to AI problems that would otherwise be overlooked.

What constraints, then, does the brain offer us? First of all, individual neurons are extremely slow devices when compared to their counterparts in digital computers. Neurons operate in the millisecond range, an eternity to a VLSI designer. Yet, humans can perform extremely complex tasks, such as interpreting a visual scene or understanding a sentence, in just a tenth of a second. In other words, we do in about a hundred steps what current computers cannot do in 10 million steps. How can this be possible? Unlike a conventional computer, the brain contains a huge number of processing elements that act in parallel. This suggests that in our search for solutions, we should look for massively parallel algorithms that require no more than 100 time steps [Feldman and Ballard, 1985].

Also, neurons are failure-prone devices. They are constantly dying (you have certainly lost a few since you began reading this chapter), and their firing patterns are irregular. Components in digital computers, on the other hand, must operate perfectly. Why? Such components store bits of information that are available nowhere else in the computer: the failure of one component means a loss of information. Suppose that we built AI programs that were not sensitive to the failure of a few components, perhaps by using redundancy and distributing information across a wide range of components? This would open up the possibility of very large-scale implementations. With current technology, it is far easier to build a *billion-component* integrated circuit in which 95 percent of the components work correctly than it is to build a *million-component* machine that functions perfectly [Fahlman and Hinton, 1987].

Another thing people seem to be able to do better than computers is handle fuzzy situations. We have very large memories of visual, auditory, and problem-solving episodes, and one key operation in solving new problems is finding closest matches to old situations. Approximate matching is something brain-style models seem to be good at, because of the diffuse and fluid way in which knowledge is represented.

The idea behind connectionism, then, is that we may see significant advances in AI if we approach problems from the point of view of brain-style computation. Connectionist AI is quite different from the symbolic approach covered in the other chapters of this book. At the end of this chapter, we discuss the relationship between the two approaches.

18.1 INTRODUCTION: HOPFIELD NETWORKS

The history of AI is curious. The first problems attacked by AI researchers were problems such as chess and theorem proving, because they were thought to require the essence of intelligence. Vision and language understanding—processes easily mastered by five-year olds—were not thought to be difficult. These days, we have expert chess programs and expert medical diagnosis programs, but no programs that can match the basic perceptual skills of a child. Neural network researchers contend that there is a basic mismatch between standard computer information processing technology and the technology used by the brain.

In addition to these perceptual tasks, AI is just starting to grapple with the fundamental problems of memory and commonsense reasoning. Computers are notorious for their lack of common sense. Many people believe that common sense derives from our massive store of knowledge and, more important, our ability to access relevant knowledge quickly, effortlessly, and at the right time.

When we read the description "gray, large, mammal," we automatically think of elephants and their associated features. We access our memories *by content*. In traditional implementations, access by content involves expensive searching and matching procedures. Massively parallel networks suggest a more efficient method.

Hopfield [1982] introduced a neural network that he proposed as a theory of memory. A Hopfield network has the following interesting features:

- *Distributed Representation*—A memory is stored as a pattern of activation across a set of processing elements. Furthermore, memories can be superimposed on one another; different memories are represented by different patterns over the *same* set of processing elements.
- *Distributed, Asynchronous Control*—Each processing element makes decisions based only on its own local situation. All these local actions add up to a global solution.
- *Content-Addressable Memory*—A number of patterns can be stored in a net work. To retrieve a pattern, we need only specify a portion of it. The network automatically finds the closest match.
- *Fault Tolerance*—If a few processing elements misbehave or fail completely, the network will still function properly.

How are these features achieved? A simple Hopfield net is shown in Fig. 18.1. Processing elements, or *units,* are always in one of two states, active or inactive. In the figure, units colored black are active and units colored white are inactive. Units are connected to each other with weighted, symmetric connections. A positively weighted connection indicates that the two units tend to activate each other. A negative connection allows an active unit to deactivate a neighboring unit.

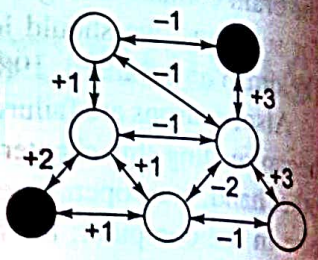


Fig. 18.1 *A Simple Hopfield Network*

The network operates as follows. A random unit is chosen. If any of its neighbors are active, the unit computes the sum of the weights on the connections to those active neighbors. If the sum is positive, the unit becomes active, otherwise it becomes inactive. Another random unit is chosen, and the process repeats until the network reaches a stable state, i.e., until no more units can change state. This process is called *parallel relaxation.* If the network starts in the state shown in Fig. 18.1, the unit in the lower left corner will tend to activate the unit above it. This unit, in turn, will attempt to activate the unit above *it,* but the inhibitory connection from the upper-right unit will foil this attempt, and so on.

This network has only four distinct stable states, which are shown in Fig. 18.2. Given any initial state, the network will necessarily settle into one of these four configurations.[1] The network can be thought of as "storing" the patterns in Fig. 18.2. Hopfield 's major contribution was to show that given any set of weights and any initial state, his parallel relaxation algorithm would eventually steer the network into a stable state. There can be no divergence or oscillation.

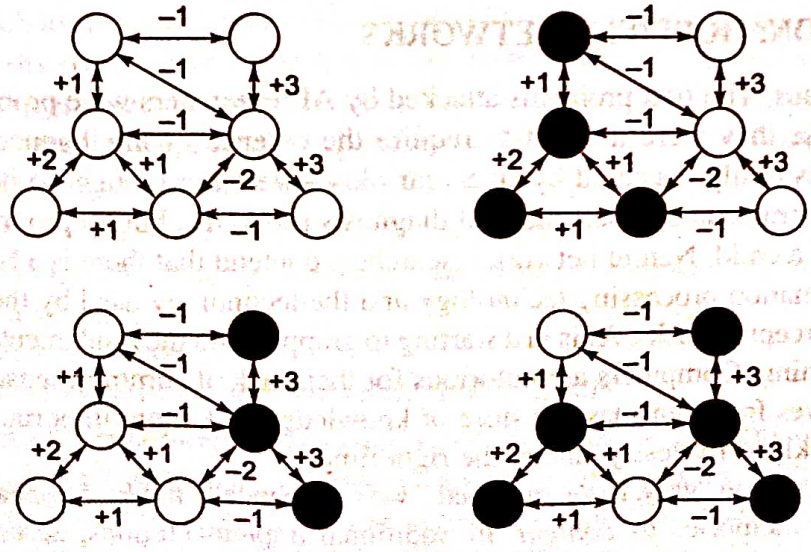


Fig. 18.2 *The Four Stable States of a Particular Hopfield Net*

[1]The stable state in which all units are inactive can only be reached if it is also the initial state.

The network can be used as a content-addressable memory by setting the activities of the units to correspond to a partial pattern. To retrieve a pattern, we need only supply a portion of it. The network will then settle into the stable state that best matches the partial pattern. An example is shown in Fig. 18.3.

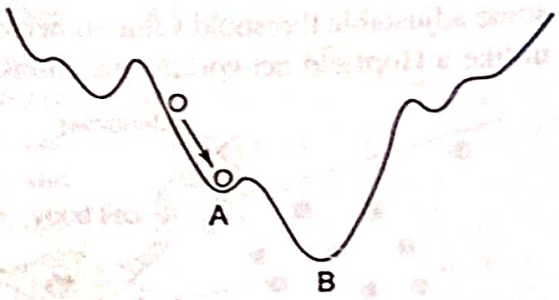


Fig. 18.3 *A Hopfield Net as a Model of Content-Addressable Memory*

Parallel relaxation is nothing more than search, albeit of a different style than the search described in the early chapters of this book. It is useful to think of the various states of a network as forming a search space, as in Fig. 18.4. A randomly chosen state will transform itself ultimately into one of the *local minima*, namely the nearest stable state. This is how we get the content-addressable behavior.[2] We also get error-correcting behavior. Suppose we read the description, "gray, large, fish, eats plankton." We imagine a whale, even though we know that a whale is a mammal, not a fish. Even if the initial state contains inconsistencies, a Hopfield network will settle into the solution that violates the fewest constraints offered by the inputs. Traditional match-and-retrieve procedures are less forgiving.

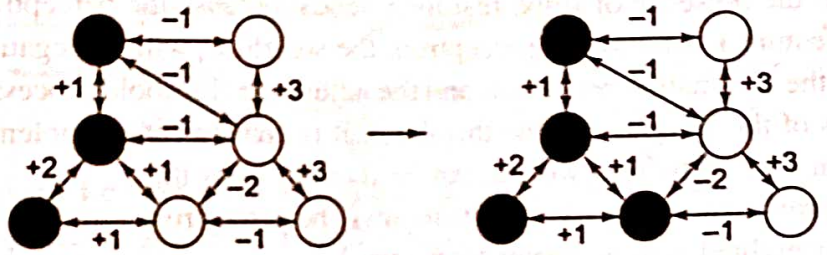


Fig. 18.4 *A Simplified View of What a Hopfield Net Computes*

Now, suppose a unit occasionally fails, say, by becoming active or inactive when it should not. This causes no major problem: surrounding units will quickly set it straight again. It would take the unlikely concerted effort of many errant units to push the network into the wrong stable state. In networks of thousands of more highly interconnected units, such fault tolerance is even more apparent—units and connections can disappear completely without adversely affecting the overall behavior of the network.

So parallel networks of simple elements can compute interesting things. The next important question is: What is the relationship between the weights on the network's connections and the local minima it settles into? In other words, if the weights encode the knowledge of a particular network, then how is that knowledge acquired? In Chapter 17 we saw several ways to acquire symbolic structures and descriptions. Such acquisition was quite difficult. One feature of connectionist architectures is that their method of representation (namely, real-valued connection weights) lends itself very nicely to 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

[2] In Fig. 18.4, state B is depicted as being lower than state A because fewer constraints are violated. A constraint is violated, for example, when two active units are connected by a negatively weighted connection.

some adjustable threshold value (otherwise it sends 0). Fig. 18.5 shows the device. Notice that in a perceptron, unlike a Hopfield network, connections are unidirectional.

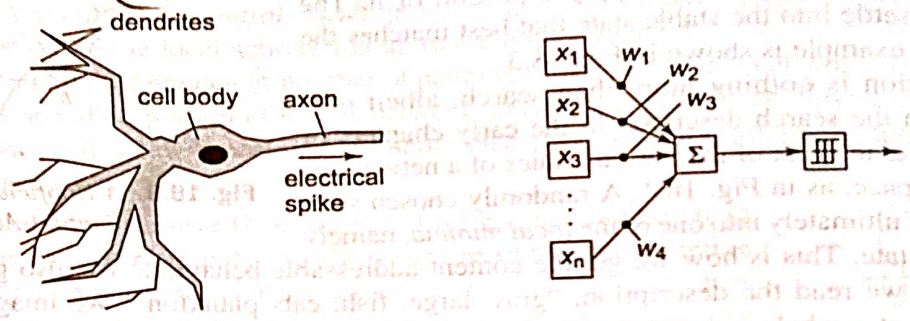


Fig. 18.5 *A Neuron and a Perceptron*

The inputs $(x_1, x_2, ..., x_n)$ and connection weights $(w_1, w_2, ..., w_n)$ 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 Fig. 18.6. This weight can be thought of as the propensity of the perceptron to fire irrespective of its inputs. The perceptron of Fig. 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 Fig. 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 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 Fig. 18.8).

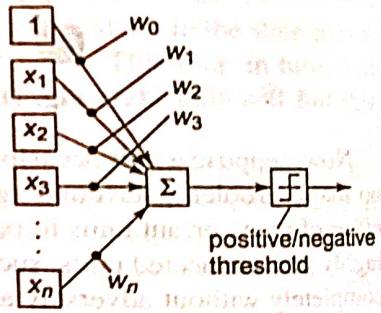


Fig. 18.6 *Perceptron with Adjustable Threshold Implemented as Additional Weight*

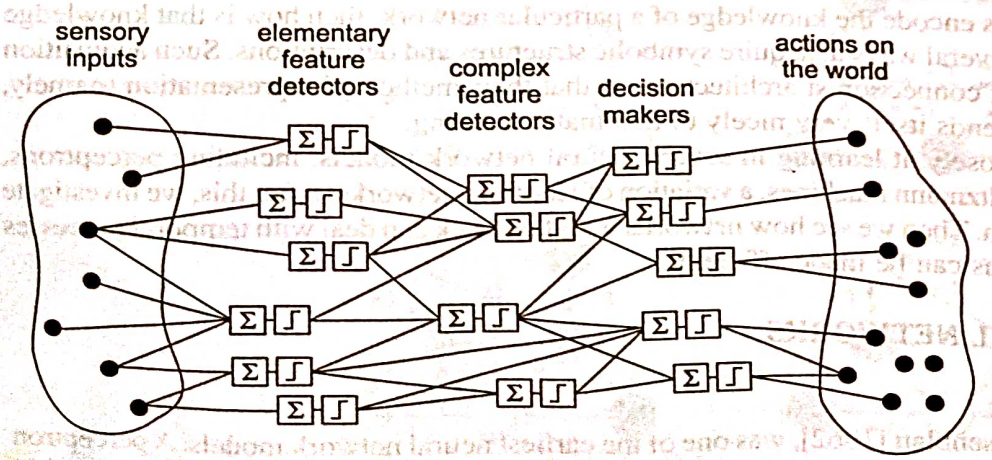


Fig. 18.8 *An Early Notion of an Intelligent System Built from Trainable Perceptrons*

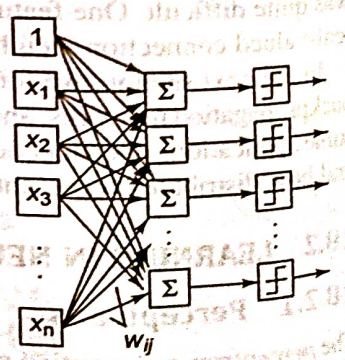


Fig. 18.7 *A Perceptron with Many Inputs and Many Outputs*

Since the perceptrons of Fig. 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 Fig. 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.

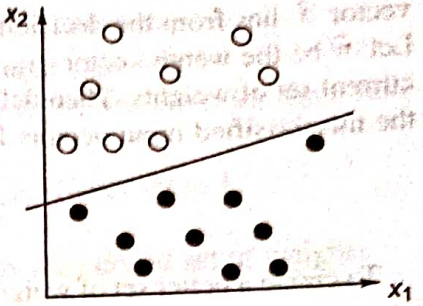


Fig. 18.9 *A Linearly Separable Pattern Classification Problem*

First, it is necessary to take a close look at what the perceptron computes. Let 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 Fig. 18.9). Then:

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

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.

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 weigh vector $(w_0, w_1, ..., w_n)$, 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:

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

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 car 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 Fig. 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 functioin to remove the absolute value sign:

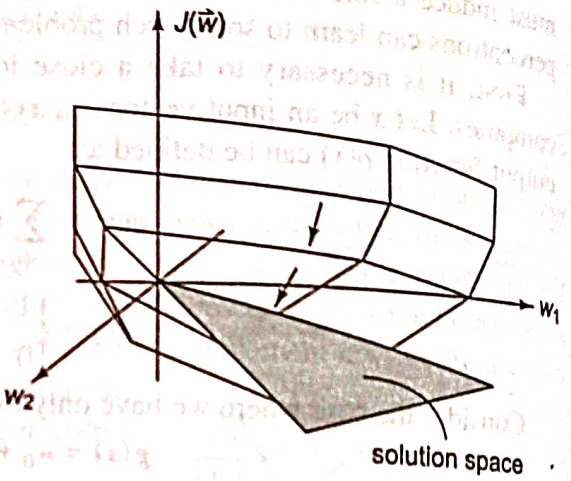


Fig. 18.10 *Adjusting the Weights by Gradient Descent, Minimizing* $J(\vec{w})$

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

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:

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

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

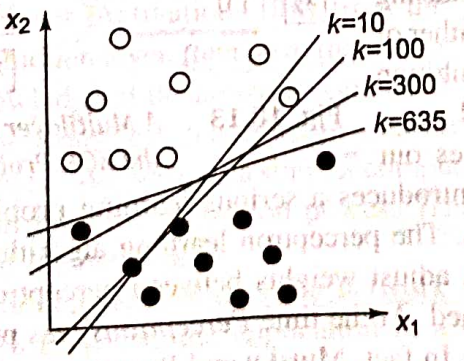
Given: A classification problem with n input features $(x_1, x_2, ..., x_n)$ and two output classes.
Compute: A set of weights $(w_0, w_1, w_2, ..., w_n)$ that will cause a perceptron to fire whenever the input falls into the first output class.

1. Create a perceptron with $n + 1$ inputs and $n + 1$ weights, where the x_0 is always set to 1.
2. Initialize the weights $(w_0, w_1, ..., w_n)$ 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 $(x_0, x_1, ..., x_n)$. 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 $-\vec{x}$ if $\vec{x}$ is an input for which the perceptron incorrectly *fires*. Multiply sum by a scale factor η.
6. Modify the weights $(w_0, w_1, ..., w_n)$ by adding the elements of the vectort 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 Fig. 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.

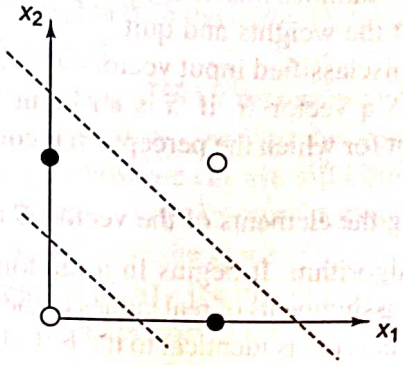


k	w_0	w_1	w_2
10	0.41	−0.17	0.14
100	0.22	−0.14	0.11
300	−0.10	−0.08	0.07
635	−0.49	−0.10	0.14

Fig. 18.11 *A Perceptron Learning to Solve a Classification Problem*

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 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 Fig. 18.12).



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

Fig. 18.12 *A Classification Problem, XOR, That Is Not Linearly Separable*

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 Fig. 18.13.

Note how the output of the first perceptron serves as one of the second perceptron, with a large, negatively weighted connection. If the first the input ($x_1 = 1$, $x_2 = 1$), it will send a massive inhibitory pulse to 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, 1 if either of the inputs is 1.

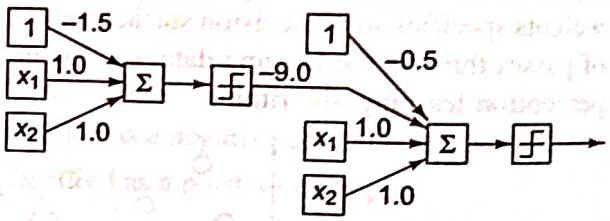


Fig. 18.13 *A Multilayer Perceptron That Solves the XOR Problem*

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 Fig. 18.13, the inhibitory weight "–9.0" was hand-coded, not learned. At the time *Perceptrons* was published, no one knew how multilayer perceptions could be made to learn. In fact, Minsky and Papert speculated:

> 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 Fig. 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 Fig. 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.

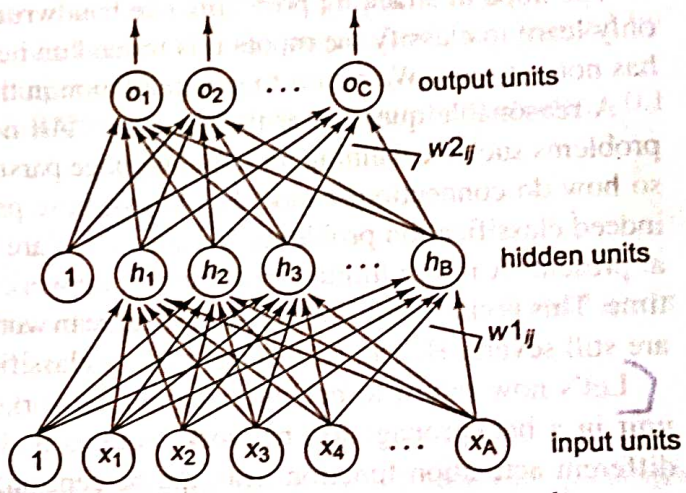


Fig. 18.14 *A Multilayer Network*

The existence of hidden units allows the network to develop complex feature detectors, or internal representations. Fig. 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 Fig. 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).

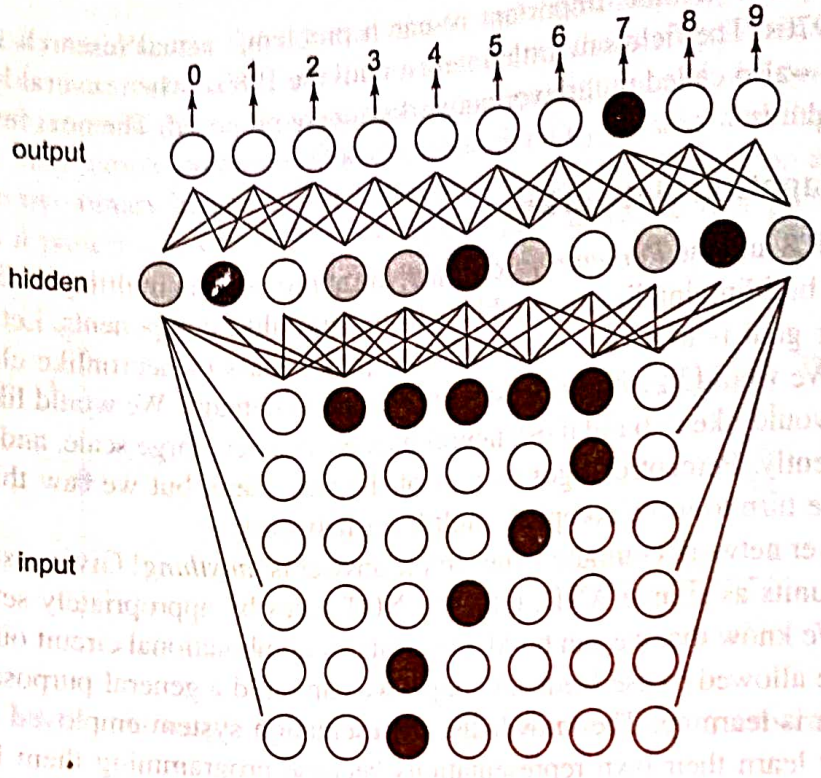


Fig. 18.15 *Using a Multilayer Network to Learn to Classify Handwritten Digits*

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 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 Fig. 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:

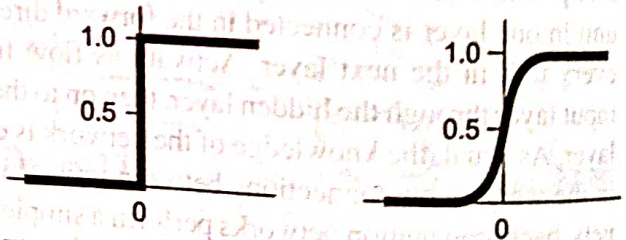


Fig. 18.16 *The Stepwise Activation Function of the Perceptron (left), and the Sigmoid Activation Function of the Backpropagation Unit (right)*

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

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.

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 Fig. 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 training input vectors. Let C be the number of units in the output. Now choose B, the number of units in the hidden layer.[3] As the input and hidden layers each have an extra unit used for thresholding therefore, the units in these layers will sometimes be indexed by ranges (0, ..., A) and (0, ..., 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 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.

$$w1_{ij} = random(-0.1, 0.1) \quad \text{for all} \quad i = 0, ..., A, j = 1, ..., B$$
$$w2_{ij} = random(-0.1, 0.1) \quad \text{for all} \quad i = 0, ..., B, j = 1, ..., C$$

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

$$x_0 = 1.0$$
$$h_0 = 1.0$$

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 hidden layer using the activation function of Fig. 18.16:

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

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.

[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).

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

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

Again, the thresholding weight $w2_{0j}$ for output unit j plays a role in the weighted summation, h_o 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).

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

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

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

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.

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

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

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

11. Go to step 4 and repeat. When all the input-output pairs have been 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:

$$\Delta w2_{ij}(t + 1) = \eta \cdot \delta 2_j \cdot h_i + \alpha \Delta w2_{ij}(t)$$
$$\Delta w1_{ij}(t + 1) = \eta \cdot \delta 1_j \cdot h_i + \alpha \Delta w1_{ij}(t)$$

where h_1, 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]

---

[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.

[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.

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_js 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 (Fig. 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 a, 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 will interpolate when given inputs it has never seen before. For example, after learning to distinguish a few different sized As from a few different sized Bs, a network will usually be able to distinguish *any* sized A from *any* sized B. Also, generalization will help to overcome any undesirable noise in the inputs.

There are some pitfalls, however. Figure 18.17 shows the common generalization effect during a long training period. During the first part of the training, performance on the training set improves as the network adjusts its weights through backpropagation. Performance on the test set (examples that the network is *not* allowed to learn on) also improves, although it is never quite as good as the training set. After a while, network performance reaches a plateau as the weights shift around, looking for a path to further improvement. Ultimately, such a path is found, and performance on the training set improves again. But performance on the test set gets worse. Why? The network has begun to memorize the individual input-output pairs rather than settling for weights that generally describe the mapping for all cases. With thousands of real-valued weights at its disposal, backpropagation is theoretically capable of storing entire training sets; with enough hidden units, the algorithm could learn to assign a hidden unit to every distinct input pattern in the training set. It is a testament to the power of backpropagation that this actually happens in practice.

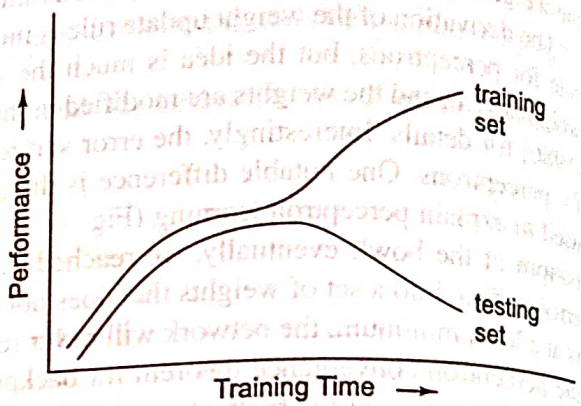


Fig. 18.17 *A Common Generalization Effect in Neural Network Learning*

Of course, that much power is undesirable. There are several ways to prevent backpropagation from resorting to a table-lookup scheme. One way is to stop training when a plateau has been reached, on the assumption that any other improvement will come through "cheating." Another way is to add deliberately small amounts of noise to the training inputs. The noise should be enough to prevent memorization, but it should not be so much that it confuses the classifier. A third way to help generalization is to reduce the number of hidden units in the network, creating a bottleneck between the input and output layers. Confronted with a bottleneck, the network will be forced to come up with compact internal representations of its inputs.

Finally, there is the issue of exceptions. In many domains, there are general rules, but there are also exceptions to the rules. For example, we can generally make the past tense of an English verb by adding "-ed" to it, but this is not true of verbs like "sing," "think," and "eat." When we show many present and past tense pairs to a network, we would like it to generalize inspite of the exceptions—but not to generalize so far that the exceptions are lost. Backpropagation performs fairly well in this regard, as do simple perceptrons, as reported in Rumelhart and McClelland [1986a].

18.2.4 Boltzmann Machines

A Boltzmann machine is a variation on the idea of a Hopfield network. Recall that pairs of units in a Hopfield net are connected by symmetric weights. Units update their states asynchronously by looking at their local connections to other units.

In addition to serving as content-addressable memories, Hopfield networks can solve a wide variety of constraint satisfaction problems. The idea is to view each unit as a "hypothesis," and to place positive weights on connections between units representing compatible or mutually supporting hypotheses, and negative weights on connections between units representing incompatible hypotheses. As the Hopfield net settles into a stable state, it attempts to assign truth and falsity to the various hypotheses while violating as few constraints as possible. We see examples of how neural networks attack real-world constraint satisfaction problems in Section 18.3.

The main problem with Hopfield networks is that they settle into local minima. Having many local minima is good for building content-addressable memories, but for constraint satisfaction tasks, we need to find the *globally* optimal state of the network. This state corresponds to an interpretation that satisfies as many interacting constraints as possible. Unfortunately, Hopfield networks cannot find global solutions because they settle into stable states via a completely distributed algorithm. If a network reaches a stable state like state *A* in Fig. 18.4, then no single unit is willing to change its state in order to move uphill, so the network will never reach globally optimal state *B*. If several units decided to change state simultaneously, the network might be able to scale the hill and slip into state *B*. We need a way to push networks into globally optimal states while maintaining our distributed approach.

At about the same time that Hopfield networks were developed, a new search technique, called *simulated annealing*, appeared in the literature. Simulated annealing, described in Chapter 3, is a technique for finding globally optimal solutions to combinatorial problems. Hinton and Sejnowski [1986] combined Hopfield networks and simulated annealing to produce networks called *Boltzmann machines*.

To understand how annealing applies, go back to Fig. 18.4 and imagine it as a black box. Imagine further a ball rolling around in the box. If we could not see into the black box, how could we coax the ball into the deepest valley? By shaking the box, of course. Now, if we shake too violently, the ball will bounce from valley to valley at random. That is, if the ball were in valley *A*, it might jump to valley *B;* but if the ball were in valley *B*, it might jump to valley *A*. If we shake too *softly*, however, the ball might find itself in valley *A*, unable to jump out. The answer suggested by annealing is to shake the box violently at first, then gradually slow down. At some point, the probability of the ball jumping from *A* to *B* will be larger than the probability of jumping from *B* to *A*. The ball will very likely find its way to valley *B*, and as the shaking becomes softer, it will be unable to escape. This is what we want.

How is this idea implemented in a neural network? Units in Boltzmann machines update their individual binary states by a *stochastic* rather than deterministic rule. The probability that any given unit will be active is given by *p:*

$$p = \frac{1}{1 + e^{\Delta E / T}}$$

where ΔE is the sum of the unit's active input lines and T is the "temperature" of the network. Stochastic updating of units is very similar to updating in Hopfield nets, except for the temperature factor. At high temperatures, units display random behavior, while at very low temperatures, units behave as in Hopfield nets. Annealing is the process of gradually moving from a high temperature down to a low temperature. The randomness added by the temperature helps the network escape from local minima.

There is a learning procedure for Boltzmann machines, i.e., a procedure that assigns weights to connections between units given a training set of initial states and final states. We do not go into the algorithm here; interested readers should see Hinton and Sejnowski [1986]. Boltzmann learning is more time-consuming than backpropagation,[8] because of the complex annealing process, but it has some advantages. For one thing, it is easier to use Boltzmann machines to solve constraint satisfaction problems. Unlike backpropagation networks, Boltzmann machines do not make a clear division between "input" and "output." For example, a Boltzmann machine might have three important sets of units, any two of which could have their values "clamped," or fixed, like the input layer of a backpropagation net—activations in the third set of units would be determined by parallel relaxation.

If the annealing is carried out properly, Boltzmann machines can avoid local minima and learn to compute any computable function of fixed-sized inputs and outputs.

[8] One deterministic variation of Boltzmann learning [Peterson and Anderson, 1987] promises to be more efficient.

18.2.5 Reinforcement Learning

What if we train our networks not with sample outputs but with punishment and reward instead? This process is certainly sufficient to train animals to perform relatively interesting tasks. Barto [1985] describes a network which learns as follows: (1) the network is presented with a sample input from the training set, (2) the network computes what it thinks should be the sample output, (3) the network is supplied with a real-valued judgment by the teacher, (4) the network adjusts its weights, and the process repeats. A positive value in step 3 indicates good performance, while a negative value indicates bad performance. The network seeks a set of weights that will prevent negative reinforcement in the future, much as an experimental rat seeks behaviors that will prevent electric shocks.

18.2.6 Unsupervised Learning

What if a neural network is given *no* feedback for its outputs, not even a real-valued reinforcement? Can the network learn anything useful? The unintuitive answer is yes.

	has-hair?	has-scales?	has-feathers?	flies?	lives in water?	lays eggs?
Dog	1	0	0	0	0	0
Cat	1	0	0	0	0	0
Bat	1	0	0	1	0	0
Whale	1	0	0	0	1	0
Canary	0	0	1	1	0	1
Robin	0	0	1	1	0	1
Ostrich	0	0	1	1	0	1
Snake	0	1	0	0	0	1
Lizard	0	1	0	0	0	1
Alligator	0	1	0	0	1	1

Fig. 18.18 *Data for Unsupervised Learning*

This form of learning is called *unsupervised learning* because no teacher is required.[9] Given a set of input data, the network is allowed to play with it to try to discover regularities and relationships between the different parts of the input.

Learning is often made possible through some notion of which features in the input set are important. But often we do not know in advance which features are important, and asking a learning system to deal with raw input data can be computationally expensive. Unsupervised learning can be used as a "feature discovery" module that precedes supervised learning.

Consider the data in Fig. 18.18. The group of ten animals, each described by its own set of features, breaks down naturally into *three* groups: mammals, reptiles and birds. We would like to build a network that can learn which group a particular animal belongs to, and to generalize so that it can identify animals it has not yet seen. We can easily accomplish this with a six-input, three-output backpropagation network. We simply present the network with an input, observe its output, and update its weights based on the errors it makes. Without a teacher, however, the error cannot be computed, so we must seek other methods.

Our first problem is to ensure that only one of the three output units becomes active for any given input. One solution to this problem is to let the network settle, find the output unit with the highest level of activation, and set that unit to 1 and all other output units to 0. In other words, the output unit with the highest activation is the only one we consider to be active. A more neural-like solution is to have the output units fight among

[9] One analogue of unsupervised learning in symbolic AI is *discovery* (Section 17.7).

themselves for control of an input vector. The scheme is shown in Fig. 18.19. The input units are directly connected to the output units, as in the perceptron, but the output units are also connected to each other via prewired negative, or inhibitory, connections. The output unit with the most activation along its input lines initially will most strongly dampen its competitors. As a result, the competitors will become weaker, losing their power of inhibition over the stronger output unit. The stronger unit then becomes even stronger, and its inhibiting effect on the other output units becomes overwhelming. Soon, the other output units are all completely inactive. This type of mutual inhibition is called *winner-take-all* behavior. One popular unsupervised learning scheme based on this behavior is known as *competitive learning*.

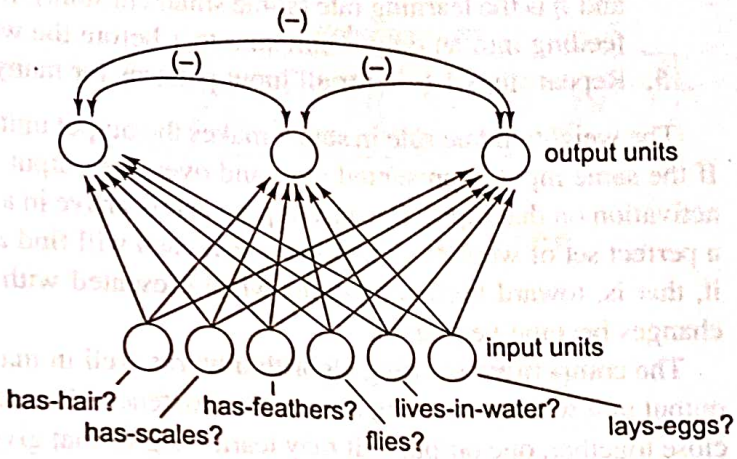


Fig. 18.19 *A Competitive Learning Network*

In competitive learning, output units fight for control over portions of the input space. A simple competitive learning algorithm is the following:

1. Present an input vector.
2. Calculate the initial activation for each output unit.
3. Let the output units fight until only one is active.
4. Increase the weights on connections between the active input units. This makes it more likely that the output unit will be time the pattern is repeated.

One problem with this algorithm is that one output unit may learn to be active all the time—it may claim all the space of inputs for itself. For example, if all the weights on a unit's input lines are large, it will tend to bully the other output units into submission. Learning will only further increase those weights.

The solution, originally due to Rosenblatt (and described in Rumelhart and Zipser [1986]), is to ration the weights. The sum of the weights on a unit's input lines is limited to 1. Increasing the weight of one connection requires that we decrease the weight of some other connection. Here is the learning algorithm.

Algorithm: Competitive Learning

Given: A network consisting of n binary-valued input units directly connected to any number of output units.

Produce: A set of weights such that the output units become active according to some natural division of the inputs.

1. Present an input vector, denoted $(x_1, x_2, ..., x_n)$.
2. Calculate the initial activation for each output unit by sum of its inputs.[10]
3. Let the output units fight until only one is active.[11]
4. Adjust the weights on the input lines that lead to the single active output unit.

$$\Delta w_j = \eta \frac{x_j}{m} - \eta w_j \quad \text{for all} \quad j = 1,..., n$$

[10] There is no reason to pass the weighted sum through a sigmoid function, as we did with backpropagation, because we only calculate activation levels for the purpose of singling out the most highly activated output unit.

[11] As mentioned earlier, any, method for determining the most highly activated output unit is sufficient. Simulators written in a serial programming language may dispense with the neural circuitry and simply compare activations levels to find the maximum.

hat this modification results in the winning neuron and its neighbours to move closer to (*learn*) nput. Naturally if the learning rate α were unity the neurons would converge on the input. For all er neurons the weights remain unaltered.

5. Update α by reducing it gradually over the iterations. This reduces the rate at which the neurons converge on the input.
6. Reduce the neighbourhood radius r gradually at specified iterations.

As can be inferred from the algorithm the modification of weight vectors results in their becoming dense in those portions where the inputs have something in common while the other areas remain rare. For instance, if we had presented the images of faces of a large number of human beings, the network would be able to tell us what is common to all of them viz. two eyes, two pinnas, a nasal elevation, lips, mouth and the like. Information stored in a Kohonen network thus could be used to find whether an alien is (or resembles) a human being. What is thus achieved is a classifier that tends to cluster the input patterns. Since this classification is done autonomously the Kohonen network is said to self organize and learn without supervision.

18.3 APPLICATIONS OF NEURAL NETWORKS

Connectionist models can be divided [Touretzky, 1989b] into the following categories based on the complexity of the problem and the network's behavior:

- Pattern recognizers and associative memories
- Pattern transformers
- Dynamic inferencers

Most of the examples we have seen so far fall into the first category. In this section, we also see networks that fall into the second category. General inferencing in connectionist networks is still at a primitive stage.

18.3.1 Connectionist Speech

Speech recognition is a difficult perceptual task (as we saw in Chapter 21). Connectionist networks have been applied to a number of problems in speech recognition; for a survey, see Lippmann [1989]. Fig. 18.21 shows how a three-layer backpropagation network can be trained to discriminate between different vowel sounds. The network is trained to output one of ten vowels, given a pair of frequencies taken from the speech waveform. Note the nonlinear decision surfaces created by backpropagation learning.

Speech production—the problem of translating text into speech rather than vice versa—has also been attacked with neural networks. Speech production is easier than speech recognition, and high performance programs are available. NETtalk [Sejnowski and Rosenberg, 1987], a network that learns to pronounce English text, was one of the first systems to demonstrate that connectionist methods could be applied to real-world tasks.

Linguists have long studied the rules governing the translation of text into speech units called *phonemes*. For example, the letter "x" is usually pronounced with a "ks" sound, as in "box" and "axe." A traditional approach to the problem would be to write all these rules down and use a production system to apply them. Unfortunately, most of the rules have exceptions—consider "xylophone"—and these exceptions must also be programmed in. Also, the rules may interact with one another in unpleasant, unforeseen ways. A connectionist approach is simply to present a network with words and their pronunciations, and hope that the network will discover the regularities and remember the exceptions. NETtalk succeeds fairly well at this task with a backpropagation network of the type described in Section 18.2.2.

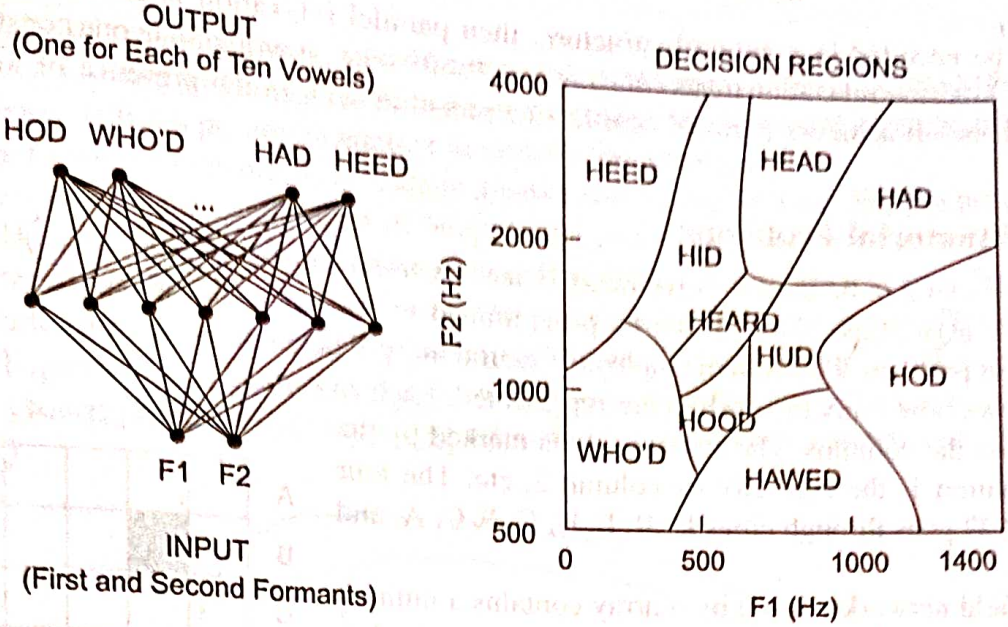


Fig. 18.21 *A Network That Learns to Distinguish Vowel Sounds*

We can think of NETtalk as an exercise in "extensional programming" [Cottrell *et al.*, 1987]. There exists some complex relationship between text and speech, and we program that relationship into the computer by showing it examples from the real world. Contrast this with traditional, "intensional programming," in which we write rules or specialized algorithms without reference to any particular examples. In the former case, we hope that the network generalizes to translate new words correctly; in the latter case, we hope that the algorithm is general enough to handle whatever words it receives. Extensional programming is a powerful technique because it drastically cuts down on knowledge acquisition time, a major bottleneck in the construction of AI systems. However, current learning methods are not adequate for the extensional programming of very complex tasks, such as the translation of English sentences into Japanese.

18.3.2 Connectionist Vision

Humans achieve significant visual progress with limited visual hardware. Only the center of the retina maintains good spatial resolution; as a result, we must constantly shift our attention among various points of interest. Each snapshot lasts only about two hundred milliseconds. Since individual neural firing rates usually lie in the millisecond range, each scene must be interpreted in about a hundred computational steps. To compound the problem, each interpretation must be rapidly integrated with previous interpretations to enable the construction of a stable three-dimensional model of the world. These severe timing constraints strongly suggest that human vision is highly parallel. Connectionism offers many methods for studying both the engineering and biological aspects of massively parallel vision.

Parallel relaxation plays an important role in connectionist vision systems [Ballard *et al.*, 1983; Ballard, 1984]. Recall our discussion of parallel relaxation search in Hopfield networks and Boltzmann machines. In a typical system, some neural units receive their initial activation levels from a video camera and then these activations are iteratively modified based on the influences of nearby units. One use for relaxation is detecting edges. If many units think they are located on an edge border, they can override any dissenters. The relaxation process settles on the most likely set of edges in the scene. While traditional vision programs running on serial computing engines must reason about which regions of a scene require edge detection processing, the connectionist approach simply assumes massively parallel machinery [Fahlman and Hinton, 1987].

Visual interpretation also requires the integration of many constraint sources. For example, if two adjacent areas in the scene have the same color and texture, then they are probably part of the same object. If these

constraints can be encoded in a network structure, then parallel relaxation is an attractive technique for combining them. Because relaxation treats constraints as "soft"—i.e., it will violate one constraint if necessary to satisfy the others—it achieves a global best-fit interpretation even in the presence of local ambiguity or noise.

18.3.3 Combinatorial Problems

Parallel relaxation can also be used to solve many other constraint satisfaction problems. Hopfield and Tank [1985] show how a Hopfield network can be programmed to come up with approximate solutions to the traveling salesman problem. The system employs n^2 neural units, where n is the number of cities to be toured. Figure 18.22 shows how tours themselves are represented. Each row stands for one city. The tour proceeds horizontally across the columns. The starting city is marked by the active unit in column 1, the next city by column 2, etc. The tour shown in Fig. 18.22 goes through cities D, B, E, H, G, F, C, A, and back to D.

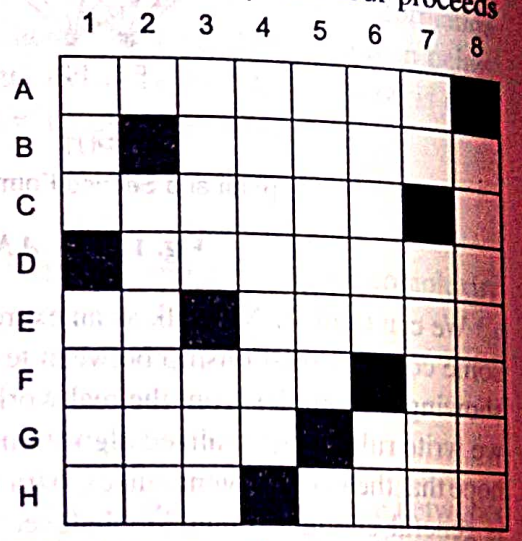


Fig. 18.22 *The Representation of a Traveling Salesman Tour in a Hopfield Network*

Like all Hopfield networks, this n by n array contains a number of weighted connections. The connection weights are initialized to reflect exactly the constraints of a particular problem instance.[12] First of all, every unit is connected with a negative weight to every other unit in its *column*, because only one city at a time can be visited. Second, every unit inhibits every other unit in its *row*, because each city can only be visited once. Third, units in adjacent columns inhibit each other in proportion to the distances between cities represented by their rows. For example, if city D is far from city G, then the fourth unit in column 3 will strongly inhibit the seventh units in columns 2 and 4. There is some global excitation, so in the absence of strong inhibition, individual units will prefer to be active.

Notice that each unit represents some hypothesis about the position of a particular city in a short tour. To find that tour, we start out by giving our units random activation values. Once all the weights are set, the units update themselves asynchronously according to the rule described in Section 18.1.[13] This updating continues until a stable state is reached. Stable states of the network correspond to short tours because conflicts between constraints are minimal. Hopfield and Tank [1985] have used these networks to come up with quick, approximate solutions to traveling salesman problems (but see Wilson and Pawley [1988] for a critique of their results). Many other combinatorial problems, such as graph-coloring, can be cast as constraint satisfaction problems and solved with parallel relaxation networks.

18.3.4 Other Applications

Other tasks successfully tackled by neural networks include learning to play backgammon [Tesauro and Sejnowski, 1989], to classify sonar signals [Gorman and Sejnowski, 1988], to compress images [Cottrell *et al.*, 1987], and to drive a vehicle along a road [Pomerleau, 1989]. While there are other techniques for attacking all these problems, learning-based connectionist systems can often be built more quickly and with less expertise than their traditional counterparts.

---

[12] Note that these connection weights are hand-coded, not learned.

[13] Actually, the units used by Hopfield and Tank [1985] take on real activation values (determined by a sigmoid curve) not binary values. By changing the shape of the sigmoid during processing, the network achieves some of the same results as does simulated annealing.

18.4 RECURRENT NETWORKS

One clear deficiency of neural network models compared to symbolic models is the difficulty in getting neural network models to deal with temporal AI tasks such as planning and natural language parsing. Recurrent networks, or networks with loops, are an attempt to remedy this situation.

Consider trying to teach a network how to shoot a basketball through a hoop. We can present the network with an input situation (distance and height of hoop, initial position of muscles), but we need more than a single output vector. We need a series of output vectors first move the muscles this way, then this way, then this way, etc. Jordan [1986] has invented a network that can do something like this. It is shown in Fig. 18.23. The network's *plan units* stay constant. They correspond to an instruction like "shoot a basket." The *state units* encode the current state of the network. The *output units* simultaneously give commands (e.g., move arm x to position y) and update the state units. The network never settles into a stable state; instead it changes at each time step.

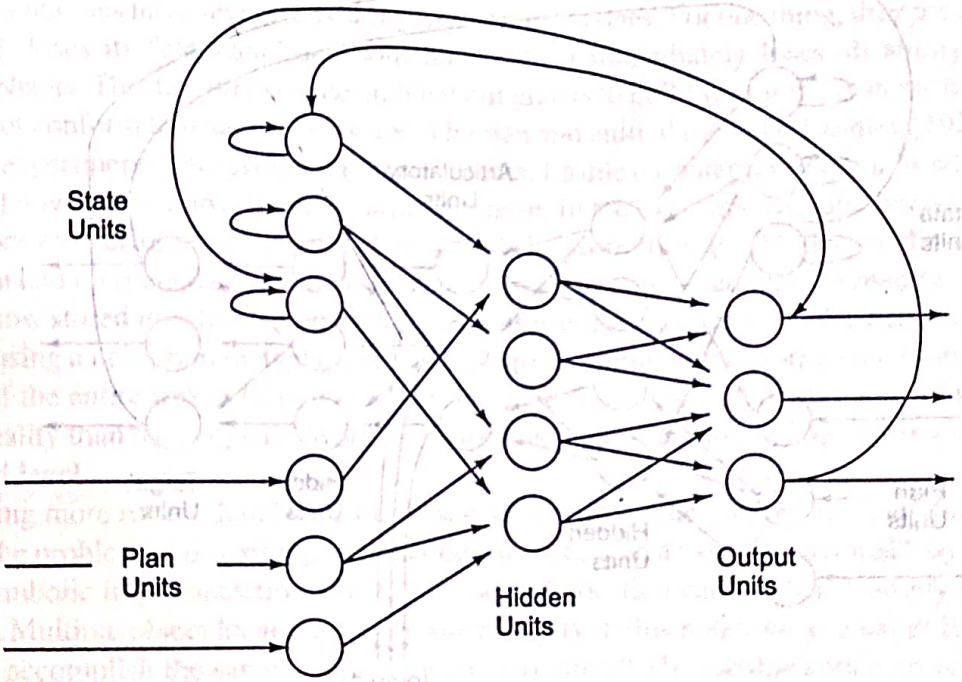


Fig. 18.23 *A Jordan Network*

Recurrent networks can be trained with the backpropagation algorithm. At each step, we compare the activations of the output units with the desired activations and propagate errors backward through the network. When training is completed, the network will be capable of performing a sequence of actions. Features of backpropagation, such as automatic generalization, also hold for recurrent networks. A few modifications are useful, however. First of all, we would like the state units to change smoothly. For example, we would not like to move from a crouched position to a jumping position instantaneously. Smoothness can be implemented as a change in the weight update rule; essentially, the "error" of an output becomes a combination of real error and the magnitude of the change in the state units. Enforcing the smoothness constraint turns out to be very important in fast learning, as it removes many of the weight-manipulation options available to backpropagation.

A major problem in supervised learning systems lies in correcting the network's behavior. If enough training data can be amassed, then target outputs can be provided for many input vectors. Recurrent networks have special training problems, however, because it is difficult to specify completely a series of target outputs. In shooting basketballs, for example, the feedback comes from the external world (i.e., where the basketball

lands), not from a teacher showing how to move each muscle. To get around this difficulty, we can learn a *mental model*, a mapping that relates the network's outputs to events in the world. Once such a model is known, the system can learn sequential tasks by backpropagating the errors it sees in the real world. So it is necessary to learn two different things: the relationship between the plan and the network's output, and the relationship between the network's output and the real world.

Networks of this type are described by Jordan [1988]. Fig. 18.24 shows such a network, which is essentially the same as a Jordan net except for the addition of two more layers: another hidden layer and a layer representing results as seen in the world. First, the latter portion of the network is trained (using backpropagation) on various pairs of outputs and targets until the network gets a good feel for how its outputs affect the real world. After these rough weights are established, the whole network is trained using real-world feedback until it is able to perform accurately.

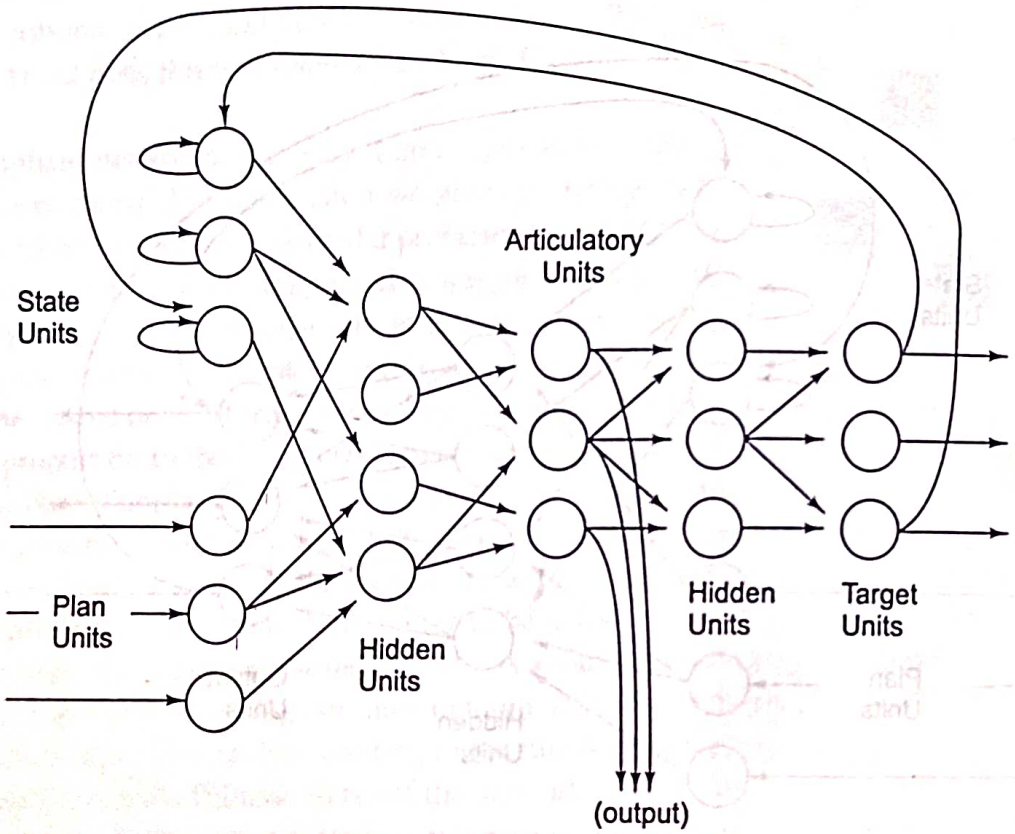


Fig. 18.24 *A Recurrent Network with a Mental Model*

Another type of recurrent network is described in Elman [1990]. In this model, activation levels are explicitly copied from hidden units to state units. Networks of this kind have been used in a number of applications, including natural language parsing.

18.5 DISTRIBUTED REPRESENTATIONS

As we have seen, the long-term knowledge of a connectionist network is stored as a set of weights on connections between units. This general scheme admits many kinds of representations, just as the basic slot-and-filler structure left room for all the representations discussed in Chapters 9 and 10. Connectionist networks can be divided roughly into two classes: those that use *localist* representations and those that use *distributed* representations.

NETL [Fahlman, 1979] is a highly parallel system that employs a localist representation. Each node in a NETL network stands for one concept in a semantic network. For example, there is a node for "elephant," a node for "gray," etc. When the network is considering an elephant, the elephant unit becomes active. This unit

then activates neighboring units, such as units for gray, large, and mammal. The reverse process works nicely as a content-addressable memory.

Distributed representations [Hinton *et al.*, 1986], on the other hand, do not use individual units to represent concepts; they use patterns of activations over many units. We have already seen one example of how this works: A Hopfield network provides a distributed representation for a content-addressable memory, in which each structure is stored as a collection of active units. One might be tempted to say that digital computers also use distributed representations. After all, a small integer is stored in a distributed fashion, as a pattern of activation over eight storage locations, each of which represents one bit of data. An extreme localist approach, on the other hand, would be to use 256 bits per integer, only one of which could be active at any given time. However, besides storing objects as patterns across many units, distributed representations have another important property, namely that stored objects may be superimposed on one another. One set of units can thus store many different objects. It is clearly impossible to store two 8-bit integers in one 8-bit place-holder, so we do not view such an encoding as a truly distributed representation.

Distributed representations have several advantages over localist ones. For one thing, they are more resistant to damage. If NETL loses its "elephant" unit somehow, then it immediately loses all ability to reason or remember about elephants. This fragility is undesirable if our goal is to build very large systems from unreliable parts. Also, it does not conform to what we know about human and animal memory. Lashley [1929] performed a number of classic experiments concerning memories in rats. Lashley wanted to find out in which part of its brain a rat stores its knowledge of how to run a particular maze. In the experiments, rats' brains were lesioned in many different places. Performance degraded in all rats in proportion to the size of the lesion, but the location of the lesion had no special effect on performance. Lashley concluded that the memory of how to run the maze was somehow stored in a distributed fashion across the entire rat cortex. Such a memory organization has been described using a hologram metaphor, in reference to the holographic storage medium, which allows the reconstruction of the entire image from just a portion of the recording (although the reconstructed image may be of poorer quality than the original). Work on distributed representations brings this metaphor down to an implementational level.

In addition to being more robust than localist representations, distributed representations can also be more efficient. Consider the problem of describing the locations of objects on a two-dimensional 8 by 8 grid [Hinton *et al.*, 1986]. In a symbolic implementation, this task is easy: A location can be stored simply as a list of two numbers, e.g., (2 3). Multiple object locations can be stored easily in this notation, as a list of lists: ((2 3) (6 5) (7 1)). How can we accomplish the same task with neuronlike units? The localist approach is to maintain an array of sixty-four units, one unit for every possible location (see Fig. 18.25). A more efficient approach would be to use a group of eight units for the x-axis and another group of eight units for the y-axis, as in Fig. 18.26. To represent the location (2 3), we activate two units: the second unit of the x-axis group and the third unit of the y-axis group. The other 14 units remain inactive. This method is not very damage resistant, however, and it will not support the representation of multiple object locations. To represent both (2 3) and (6 5) would require turning on two x-axis units and two y-axis units. But then we get the following *binding problem*: it is impossible to tell which of the four x-y pairs (2 3), (2 5), (6 3), and (6 5) correspond to actual object locations.

There is a distributed representation for solving this problem—it is called *coarse coding*. In coarse coding, we divide the space of possible object locations into a number of large, overlapping, circular zones. See, for example, Fig. 18.27, in which units are depicted as small dots and their receptive fields as large circles. A unit becomes active if any object is located within its receptive field. There is a unit associated with each zone—the zone is called the unit's *receptive field*.[14] Whenever an object is located in a unit's receptive field, the unit

[14] The term *receptive field* comes from the study of vision. A receptive field of a retinal cell is an area of the retina that the cell is responsible for. The cell is triggered by light in that area.

becomes active. By looking at a single active unit, we cannot tell with any accuracy where an object is located, but by looking at the pattern of activity across all the units, we can actually be quite precise. Consider that the intersection of several circular zones associated with a group of units may be a very small area—if only those units are active, we can be fairly precise about where the object is located. In fact, as the receptive fields become larger, i.e., as the individual units become *less* discriminating about object locations, the whole representation becomes *more* accurate, because the regions of intersection become smaller. In the end, we can represent multiple objects with some precision without paying the price of the localist representation scheme.

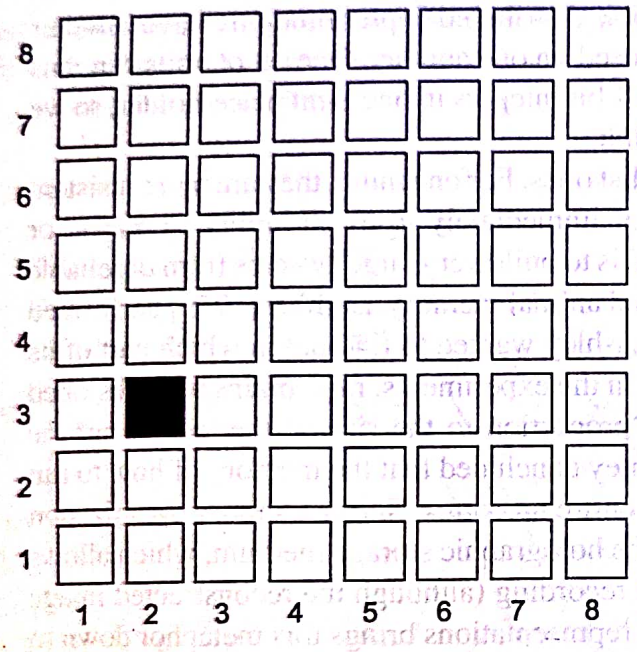


Fig. 18.25 *A Localist Representation of Location (2 3) on an 8 by 8 Grid*

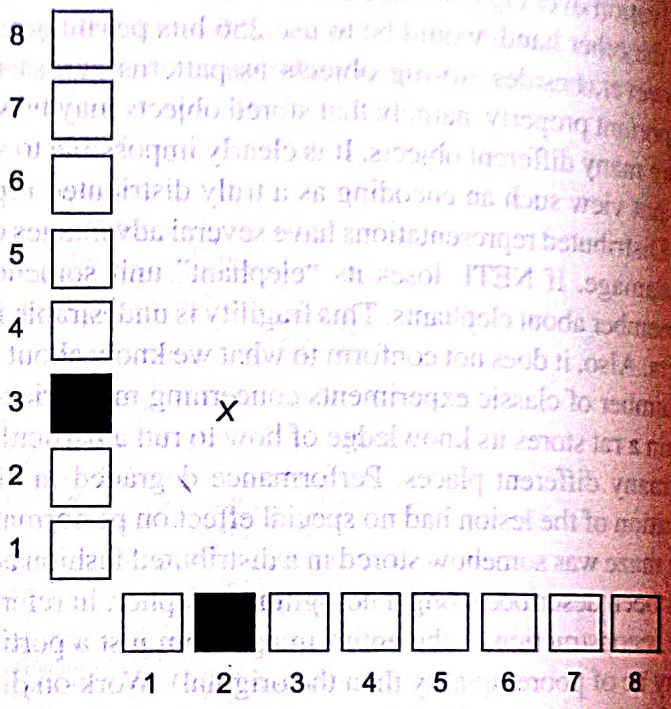


Fig. 18.26 *A More Efficient Representation, Requiring Only 16 Units, but Unable to Store Multiple Locations*

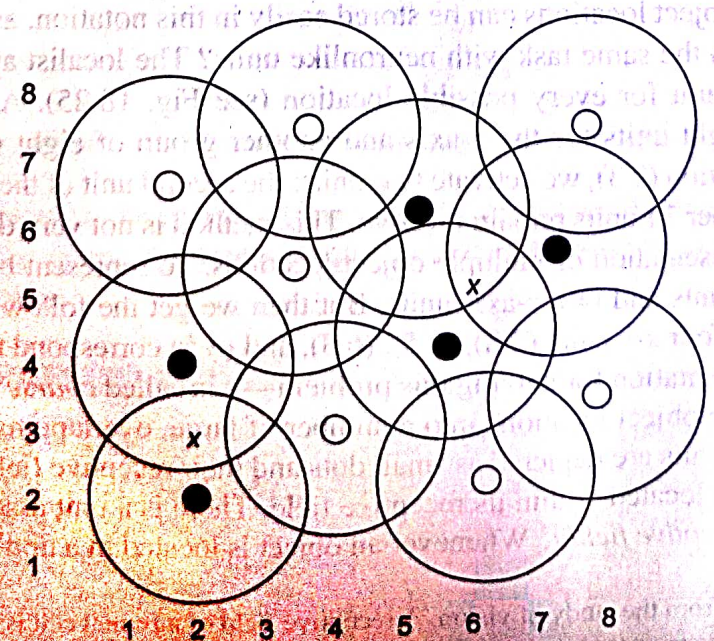


Fig. 18.27 *Distributed Representation Using Coarse Coding*

One drawback to distributed representations is that they cannot store many densely packed objects. A localist or symbolic system could easily represent the three distinct objects at (4 4), (4 5), and (5 4), but a distributed scheme would be confounded by the loss of information caused by the effect of many objects on a single unit's receptive field. On the other hand, psychological experiments have shown that a similar *interference effect* is very likely a cause of forgetting in human memory [Gleitman, 1981]. A more serious deficiency concerning distributed representations lies in the difficulty of interpreting, acquiring, and modifying them by hand. Thus, they are usually used in conjunction with automatic learning mechanisms of the type discussed in Section 18.2.

18.6 CONNECTIONIST AI AND SYMBOLIC AI

The connectionist approach to AI is quite different from the traditional symbolic approach. Both approaches are certainly joined at the problem; both try to address difficult issues in search, knowledge representation, and learning. Let's list some of the methods used by both:

1. Connectionist
   - Search—Parallel relaxation.
   - Knowledge Representation—Very large number of real valued connection strengths. Structures often stored as distributed patterns of activation.
   - Learning—Backpropagation, Boltzmann machines, reinforcement learning, unsupervised learning.
2. Symbolic
   - Search—State space traversal.
   - Knowledge Representation—Predicate logic, semantic frames, scripts.
   - Learning—Macro-operators, version spaces, explantion-learning, discovery.

The approaches have different strengths and weaknesses. One major allure of connectionist systems is that they employ knowledge representations that seem to be more learnable than their symbolic counterparts. Nearly all connectionist systems have a strong learning component. However, neural network learning algorithms usually involve a large number of training examples and long training periods compared to their symbolic cousins. Also, after a network has learned to perform a difficult task, its knowledge is usually quite opaque, an impenetrable mass of connection weights. Getting the network to explain its reasoning, then, is difficult. Of course, this may not be a bad thing. Humans, for example, appear to have little access to the procedures they use for many tasks such as speech recognition and vision. It is no accident that the most promising uses for neural networks are in these areas of low-level perception.

Connectionist knowledge representation offers other advantages besides learnability. Touretzky and Geva [1987] discuss the fluidity and richness of connectionist representations. In connectionist models, concepts are represented as feature vectors, sets of activation values over groups of units. Similar concepts are given similar feature vector representations. In symbolic models, on the other hand, concepts are usually given atomic labels that bear no surface relation to each other, such as *Car* and *Porsche*. Links (like *isa*) are used to describe relationships between concepts. When the relationships become more fuzzy than *isa*, however, symbolic systems have difficulty doing matching. For example, consider the phrases "mouth of a bird" and "nose of a bird." People have no trouble mapping these phrases onto the concept *Beak*. A connectionist system could perform this fuzzy match by considering that *Nose*, *Mouth*, and *Beak* have similar feature value representations. Moreover, symbolic systems do not handle multiple, related shades of meaning very well. Consider the sentence, "The newspaper changed its format." Usually, the word "newspaper" is interpreted either as (1) something made of black and white paper or (2) a group of people in charge of producing a daily

periodical. In the sentence above, however, it is impossible to choose between the two readings. In symbolic systems, different word senses are represented as independent atomic objects. Connectionist models offer several ways of maintaining multiple meanings: the simultaneous activations of different units (localist), the superposition of activity patterns (distributed), and the choice of intermediate feature vectors. The third method involves choosing a representation that shares some features of one meaning and some feature of another, but the intermediate representation itself has no single, corresponding symbolic concept.

A major part of this book has been devoted to the study of search in symbolic systems. It is difficult to see how connectionist systems will tackle difficult problems that state-space search addresses (e.g., chess, theorem-proving, and planning). Parallel relaxation search, however, does have some advantages over symbolic search. First of all, it maps naturally onto highly parallel hardware. When such hardware becomes widely available, parallel relaxation methods will be extremely efficient. More importantly, parallel relaxation search may prove even more efficient because it makes use of states that have no analogues in symbolic search. We saw this phenomenon briefly in Section 18.3.3 when we considered a Hopfield network that comes up with short traveling salesman tours. In the process of settling into a solution state, the network enters and exits many "impossible" states, such as ones in which a city is visited twice, or ones in which the traveler is in two places at the same time. Eventually, a valid solution state falls out of the relaxation process. In contrast, a symbolic system can only expand new search nodes that correspond to valid, possible states of the world.

A good deal of connectionist research concerns itself with modeling human mental processes. Neural networks seem to display many psychologically and biologically plausible features such as content-addressable memory, fault tolerance, distributed representations, and automatic generalization. Can we integrate these desirable properties into symbolic AI systems? Certainly, high-level theories of cognition can incorporate such features as new psychological primitives. Practically speaking, we may want to use connectionist architectures for low-level tasks such as vision, speech recognition, and memory, feeding results from these modules into symbolic AI programs. Another idea is to take a symbolic notion and implement it in a connectionist framework. Touretzky and Hinton [1988] describe a connectionist production system, and Derthick [1988] describes a connectionist semantic network.

A third idea is to program a symbolic system with the basic principles that are necessary to perform a task and then use the symbolic system to guide the performance of a neural network, which refines its behavior as it acquires experience. An example of this approach is described by Handelman *et al.* [1989], who describe a robot arm that can throw a ball at a target. Initially, a symbolic system guides the behavior of the arm. Each throw produces a training case, which is fed to a neural network. The symbolic system monitors the progress of the network, which is acquiring the fine motor control that the symbolic system lacks. When the network's behavior exceeds a set criterion, control of the arm is turned over to it.

Ultimately, connectionists would like to see symbolic structures "emerge" naturally from complex interactions among simple units, in the same way that "wetness" emerges from the combination of hydrogen and oxygen, although it is an intrinsic property of neither.

Most of the promising advantages of connectionist systems described in this section are just that: promising. A great deal of work remains to be done to turn these promises into results. Only time will tell how influential connectionist models will be in the evolution of AI research. In any case, connectionists can at least point to the brain as an existence proof that neural networks, in some form, are capable of exhibiting intelligent behavior.