C H A P T E R

1

Introduction to ANS Technology

When the only tool you have is a hammer, every **problem** *you encounter tends to resemble a nail.*

—*Source unknown*

Why can't we build a computer that thinks? Why can't we expect machines that can perform 100 million floating-point calculations per second to be able to comprehend the meaning of shapes in visual images, or even to distinguish between different kinds of similar objects? Why can't that same machine *learn* from experience, rather than repeating forever an explicit set of instructions generated by a human programmer?

These are only a few of the many questions facing computer designers, engineers, and programmers, all of whom are striving to create more "intelligent" computer systems. The inability of the current generation of computer systems to interpret the world at large does not, however, indicate that these machines are completely inadequate. There are many tasks that are ideally suited to solution by conventional computers: scientific and mathematical problem solving; database creation, manipulation, and maintenance; electronic communication; word processing, graphics, and desktop publication; even the simple control functions that add intelligence to and simplify our household tools and appliances are handled quite effectively by today's computers.

In contrast, there are many applications that we would like to automate, but have not automated due to the complexities associated with programming a computer to perform the tasks. To a large extent, the problems are not unsolvable; rather, they are difficult to solve using sequential computer systems. This distinction is important. If the only tool we have is a sequential computer, then we will naturally try to cast every problem in terms of sequential algorithms. Many problems are not suited to this approach, however, causing us to expend a great deal of effort on the development of sophisticated algorithms, perhaps even failing to find an acceptable solution.

In the remainder of this text, we will examine many parallel-processing architectures that provide us with new tools that can be used in a variety of applications. Perhaps, with these tools, we will be able to solve more easily currently difficult-to-solve, or unsolved, problems. Of course, our proverbial hammer will still be extremely useful, but with a full toolbox we should be able to accomplish much more.

As an example of the difficulties we encounter when we try to make a sequential computer system perform an inherently parallel task, consider the problem of visual pattern recognition. Complex patterns consisting of numerous elements that, individually, reveal little of the total pattern, yet collectively represent easily recognizable (by humans) objects, are typical of the kinds of patterns that have proven most difficult for computers to recognize. For example, examine the illustration presented in Figure 1.1. If we focus strictly on the black splotches, the picture is devoid of meaning. Yet, if we allow our perspective to encompass all the components, we can see the image of a commonly recognizable object in the picture. Furthermore, once we see the image, it is difficult for us *not* to see it whenever we again see this picture.

Now, let's consider the techniques we would apply were we to program a conventional computer to recognize the object in that picture. The first thing our program would attempt to do is to locate the primary area or areas of interest in the picture. That is, we would try to segment or cluster the splotches into groups, such that each group could be uniquely associated with one object. We might then attempt to find edges in the image by completing line segments. We could continue by examining the resulting set of edges for consistency, trying to determine whether or not the edges found made sense in the context of the other line segments. Lines that did not abide by some predefined rules describing the way lines and edges appear in the real world would then be attributed to noise in the image and thus would be eliminated. Finally, we would attempt to isolate regions that indicated common textures, thus filling in the holes and completing the image.

The illustration of Figure 1.1 is one of a dalmatian seen in profile, facing left, with head lowered to sniff at the ground. The image indicates the complexity of the type of problem we have been discussing. Since the dog is illustrated as a series of black spots on a white background, how can we write a computer program to determine accurately which spots form the outline of the dog, which spots can be attributed to the spots on his coat, and which spots are simply distractions?

An even better question is this: How is it that we can see the dog in. the image quickly, yet a computer cannot perform this discrimination? This question is especially poignant when we consider that the switching time of the components in modern electronic computers are more than seven orders of magnitude faster than the cells that comprise our neurobiological systems. This

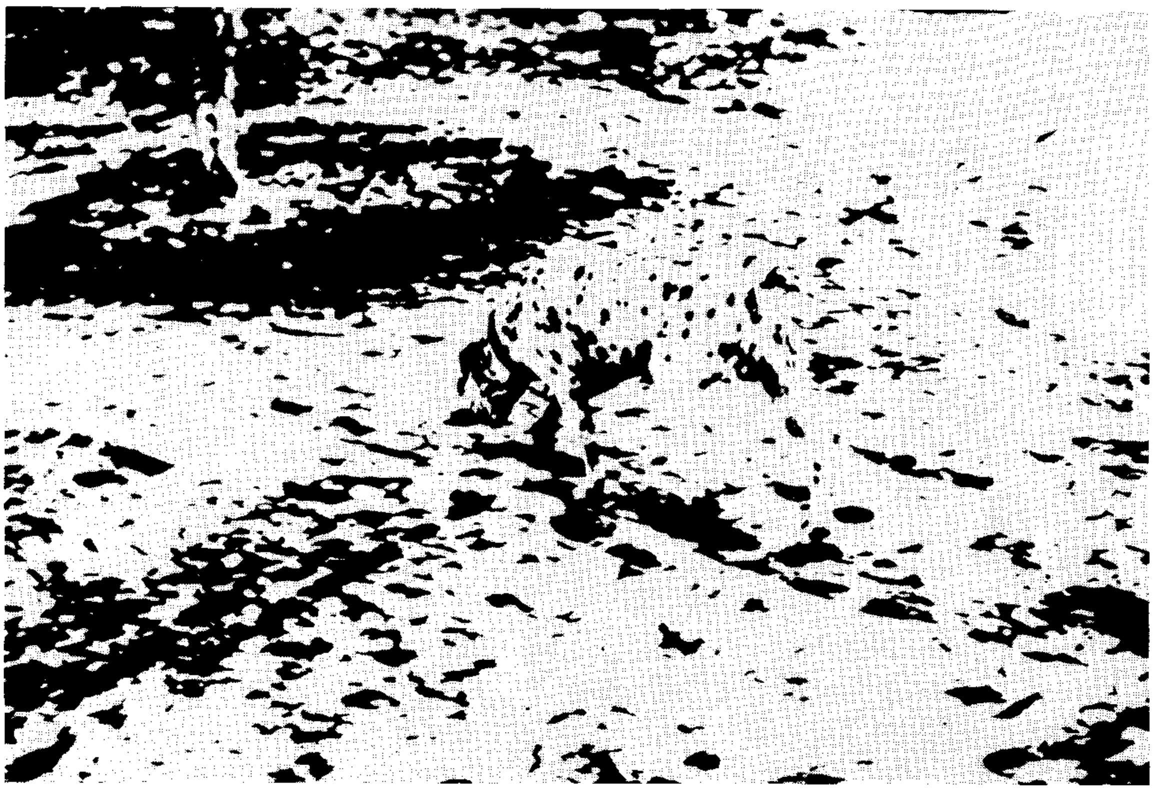


Figure 1.1 The picture is an example of a complex pattern. Notice how the image of the object in the foreground blends with the background clutter. Yet, there is enough information in this picture to enable us to perceive the image of a commonly recognizable object. *Source: Photo courtesy of Ron James.*

question is partially answered by the fact that the architecture of the human brain is significantly different from the architecture of a conventional computer. Whereas the response time of the individual neural cells is typically on the order of a few tens of milliseconds, the massive parallelism and **interconnectivity** observed in the biological systems evidently account for the ability of the brain to perform complex pattern recognition in a few hundred milliseconds.

In many real-world applications, we want our computers to perform complex pattern recognition problems, such as the one just described. Since our conventional computers are obviously not suited to this type of problem, we therefore borrow features from the physiology of the brain as the basis for our new processing models. Hence, the technology has come to be known as **artificial neural systems** (ANS) technology, or simply **neural networks.** Perhaps the models we discuss here will enable us eventually to produce machines that can interpret complex patterns such as the one in Figure 1.1.

In the next section, we will discuss aspects of neurophysiology that contribute to the ANS models we will examine. Before we do that, let's first consider how an ANS might be used to formulate a computer solution to a pattern-matching problem similar to, but much simpler than, the problem of recognizing the dalmation in Figure 1.1. Specifically, the problem we will address is recognition of hand-drawn alphanumeric characters. This example is particularly interesting for two reasons:

- Even though a character set can be defined rigorously, people tend to personalize the manner in which they write the characters. This subtle variation in style is difficult to deal with when an algorithmic pattern-matching approach is used, because it **combinatorially** increases the size of the legal input space to be examined.
- As we will see in later chapters, the neural-network approach to solving the problem not only can provide a feasible solution, but also can be used to gain insight into the nature of the problem.

We begin by defining a neural-network structure as a collection of *parallel processors* connected together in the form of a directed graph, organized such that the network structure lends itself to the problem being considered. Referring to Figure 1.2 as a typical network diagram, we can schematically represent each **processing element** (or **unit)** in the network as a **node,** with connections between units indicated by the arcs. We shall indicate the direction of information flow in the network through the use of the arrowheads on the connections.

To simplify our example, we will restrict the number of characters the neural network must recognize to the 10 decimal digits, $0, 1, \ldots, 9$, rather than using the full ASCII character set. We adopt this constraint only to clarify the example; there is no reason why an ANS could not be used to recognize all characters, regardless of case or style.

Since our objective is to have the neural network determine which of the 10 digits a particular hand-drawn character is, we can create a network structure that has 10 discrete output units (or processors), one for each character to be identified. This strategy simplifies the character-discrimination function of the network, as it allows us to use a network that contains binary units on the **output layer** (e.g., for any given input pattern, our network should activate one and only one of the 10 output units, representing which of the 10 digits that we are attempting to recognize the input most resembles). Furthermore, if we insist that the output units behave according to a simple **on–off** strategy, the process of converting an input signal to an output signal becomes a simple majority function.

Based on these considerations, we now know that our network should contain 10 binary units as its output structure. Similarly, we must determine how we will model the character input for the network. Keeping in mind that we have already indicated a preference for binary output units, we can again simplify our task if we model the input data as a vector containing binary elements, which will allow us to use a network with only one type of processing unit. To create this type of input, we borrow an idea from the video world and *pixelize* the character. We will arbitrarily size the pixel image as a 10 x 8 matrix, using a 1 to represent a pixel that is "on," and a 0 to represent a pixel that is "off."

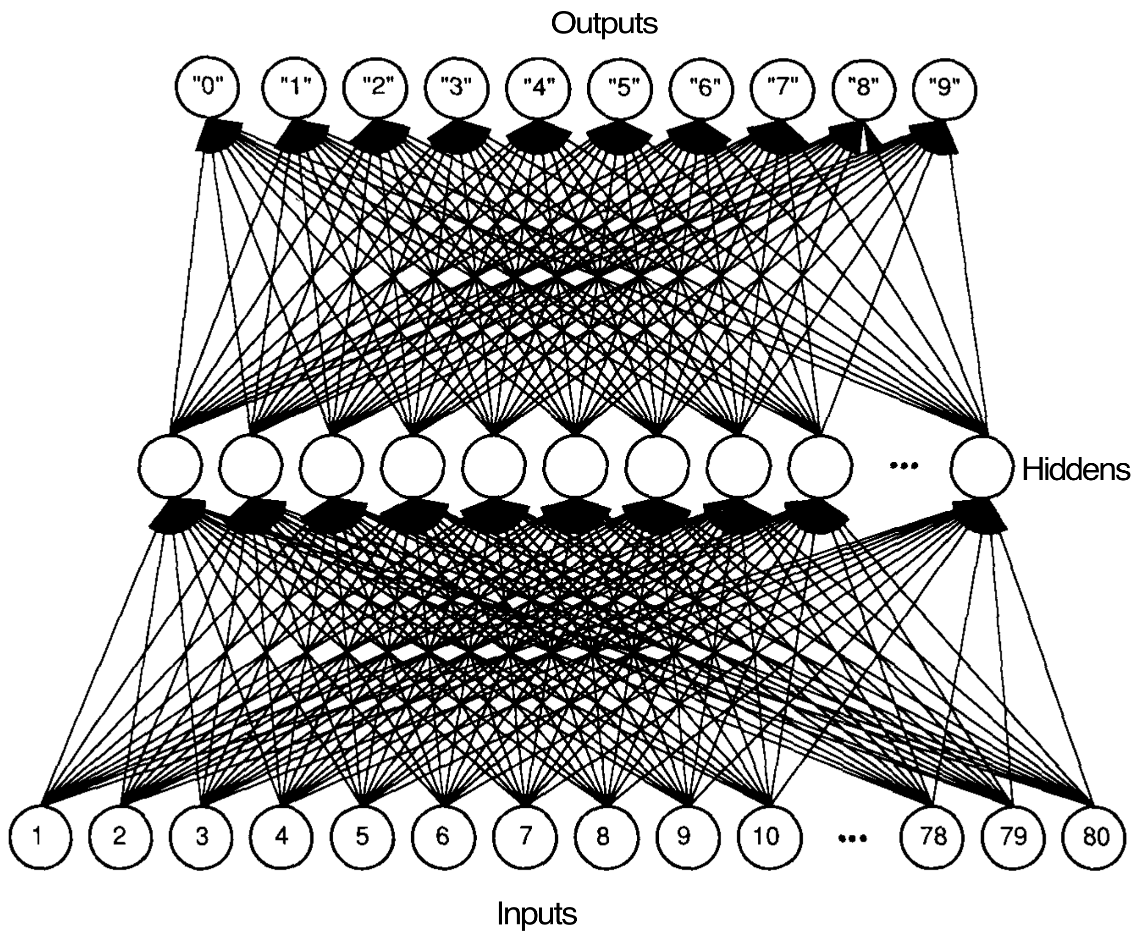


Figure 1.2 This schematic represents the character-recognition problem described in the text. In this example, application of an input pattern on the bottom layer of processors can cause many of the second-layer, or *hidden-layer,* units to activate. The activity on the hidden layer should then cause exactly one of the output-layer units to activate—the one associated with the pattern being identified. You should also note the large number of connections needed for this relatively small network.

Furthermore, we can dissect this matrix into a set of row vectors, which can then be concatenated into a single row vector of dimension 80. Thus, we have now defined the dimension and characteristics of the input pattern for our network.

At this point, all that remains is to size the number of processing units (called **hidden** units) that must be used internally, to connect them to the input and output units already defined using **weighted connections,** and to train the network with example data pairs.[1] This concept of learning by example is extremely important. As we shall see, a significant advantage of an ANS approach to solving a problem is that we need not have a well-defined process for algorithmically converting an input to an output. Rather, all that we need for most

[1] Details of how this training is accomplished will occupy much of the remainder of the text.

networks is a collection of representative examples of the desired translation. The ANS then adapts itself to reproduce the desired outputs when presented with the example inputs.

In addition, as our example network illustrates, an ANS is robust in the sense that it will respond with an output even when presented with inputs that it has never seen before, such as patterns containing noise. If the input noise has not obliterated the image of the character, the network will produce a good guess using those portions of the image that were not obscured and the information that it has stored about how the characters are supposed to look. The inherent ability to deal with noisy or obscured patterns is a significant advantage of an ANS approach over a traditional algorithmic solution. It also illustrates a neural-network maxim: The power of an ANS approach lies not necessarily in the elegance of the particular solution, but rather in the generality of the network to *find its own solution* to particular problems, given only examples of the desired behavior.

Once our network is trained adequately, we can show it images of numerals written by people whose writing was not used to train the network. If the training has been adequate, the information propagating through the network will result in a single element at the output having a binary 1 value, and that unit will be the one that corresponds to the numeral that was written. Figure 1.3 illustrates characters that the trained network can recognize, as well as several it cannot.

In the previous discussion, we alluded to two different types of network operation: *training* mode and *production* mode. The distinct nature of these two modes of operation is another useful feature of ANS technology. If we note that the process of training the network is simply a means of encoding information about the problem to be solved, and that the network spends most of its productive time being exercised after the training has completed, we will have uncovered a means of allowing automated systems to evolve *without explicit reprogramming*.

As an example of how we might benefit from this separation, consider a system that utilizes a software simulation of a neural network as part of its programming. In this case, the network would be modeled in the host computer system as a set of data structures that represents the current state of the network. The process of training the network is simply a matter of altering the connection weights systematically to encode the desired **input–output** relationships. If we code the network simulator such that the data structures used by the network are allocated dynamically, and are initialized by reading of connection-weight data from a disk file, we can also create a network simulator with a similar structure in another, off-line computer system. When the on-line system must change to satisfy new operational requirements, we can develop the new connection weights off-line by training the network simulator in the remote system. Later, we can update the operational system by simply changing the connection-weight initialization file from the previous version to the new version produced by the off-line system.

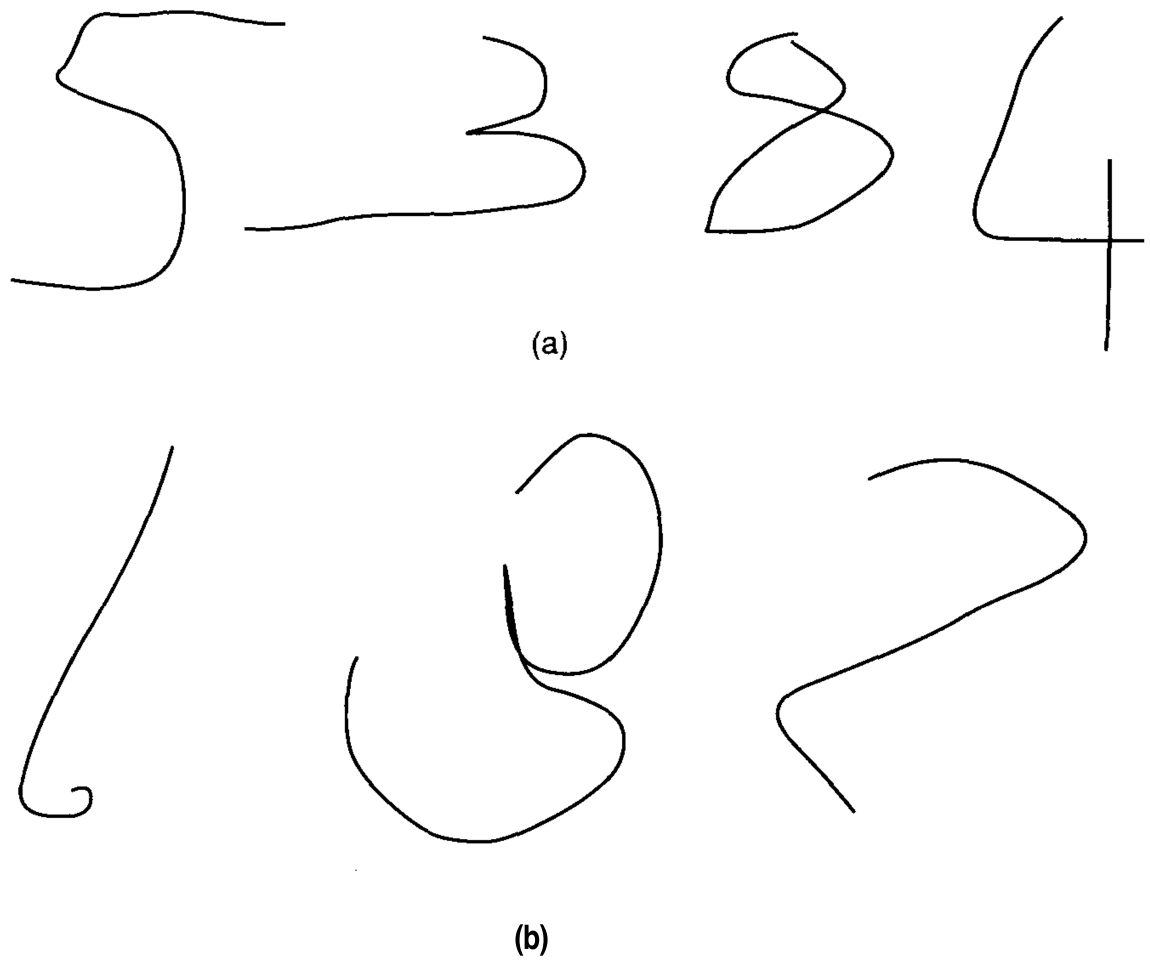


Figure 1.3 Handwritten characters vary greatly. (a) These characters were recognized by the network in Figure 1.2; (b) these characters were not recognized.

These examples hint at the ability of neural networks to deal with complex pattern-recognition problems, but they are by no means indicative of the limits of the technology. In later chapters, we will describe networks that can be used to diagnose problems from symptoms, networks that can adapt themselves to model a topological mapping accurately, and even networks that can learn to recognize and reproduce a temporal sequence of patterns. All these networks are based on the simple building blocks discussed previously, and derived from the topics we shall discuss in the next two sections.

Finally, the distinction made between the artificial and natural systems is intentional. We cannot overemphasize the fact that the ANS models we will examine bear only a perfunctory resemblance to their biological counterparts. What is important about these models is that they all exhibit the useful behaviors of learning, recognizing, and applying relationships between objects and patterns of objects in the real world. In this regard, they provide us with a whole new set of tools that we can use to solve "difficult" problems.

1.1 ELEMENTARY NEUROPHYSIOLOGY

From time to time throughout this text, we shall cite specific results from neurobiology that pertain to a particular ANS architecture. There are also basic concepts that have a more universal significance. In this regard, we look first at individual neurons, then at the synaptic junctions between neurons. We describe the McCulloch–Pitts model of neural computation, and examine its specific relationship to our neural-network models. We finish the section with a look at Hebb's theory of learning. Bear in mind that the following discussion is a simplified overview; the subject of neurophysiology is vastly more complicated than is the picture we paint here.

1.1.1 Single-Neuron Physiology

Figure 1.4 depicts the major components of a typical nerve cell in the central nervous system. The membrane of a neuron separates the intracellular plasma from the interstitial fluid external to the cell. The membrane is permeable to certain ionic species, and acts to maintain a potential difference between the

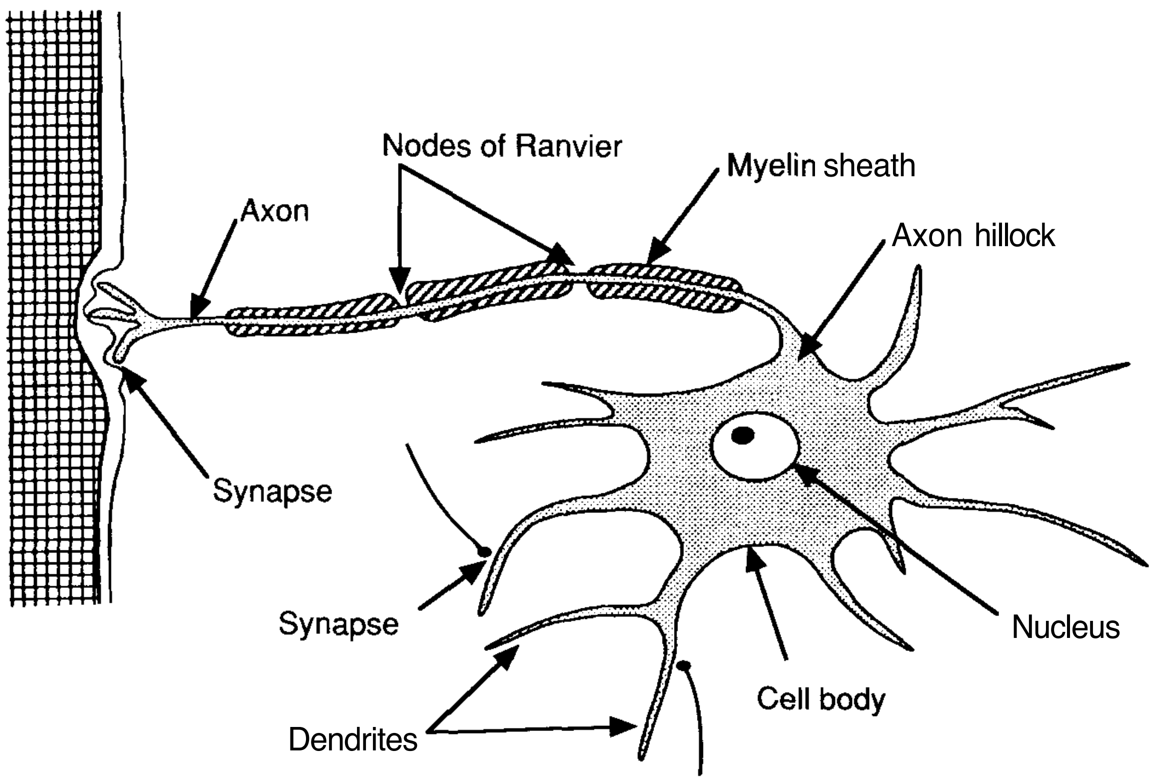


Figure 1.4 The major structures of a typical nerve cell include dendrites, the cell body, and a single axon. The axon of many neurons is surrounded by a membrane called the myelin sheath. Nodes of Ranvier interrupt the myelin sheath periodically along the length of the axon. Synapses connect the axons of one neuron to various parts of other neurons.

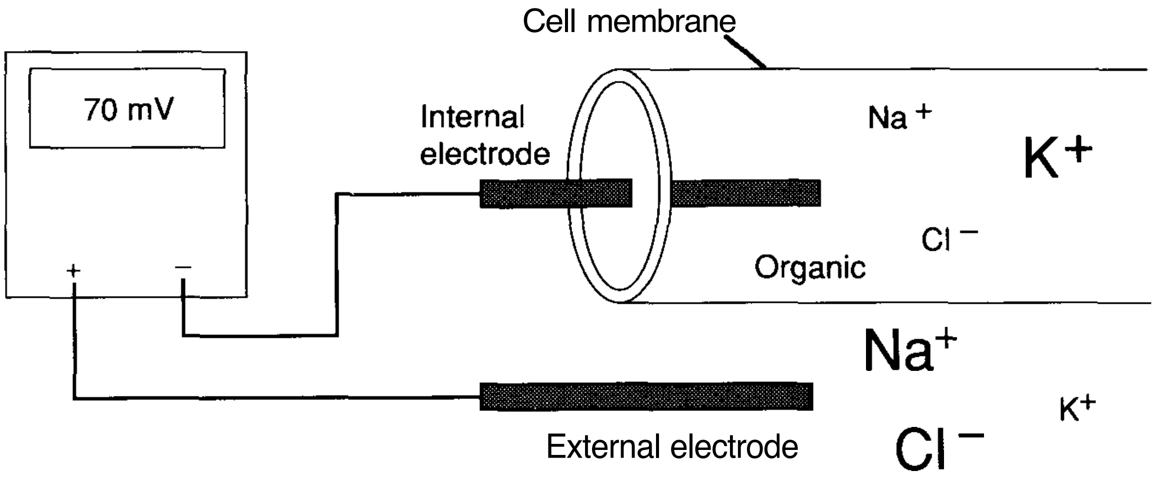


Figure 1.5 This figure illustrates the resting potential developed across the cell membrane of a neuron. The relative sizes of the labels for the ionic species indicate roughly the relative concentration of each species in the regions internal and external to the cell.

intracellular fluid and the extracellular fluid. It accomplishes this task primarily by the action of a sodium-potassium pump. This mechanism transports sodium ions out of the cell and potassium ions into the cell. Other ionic species present are chloride ions and negative organic ions.

All the ionic species can diffuse across the cell membrane, with the exception of the organic ions, which are too large. Since the organic ions cannot diffuse out of the cell, their net negative charge makes chloride diffusion into the cell unfavorable; thus, there will be a higher concentration of chloride ions outside of the cell. The sodium-potassium pump forces a higher concentration of potassium inside the cell and a higher concentration of sodium outside the cell.

The cell membrane is selectively more permeable to potassium ions than to sodium ions. The chemical gradient of potassium tends to cause potassium ions to diffuse out of the cell, but the strong attraction of the negative organic ions tends to keep the potassium inside. The result of these opposing forces is that an equilibrium is reached where there are significantly more sodium and chloride ions outside the cell, and more potassium and organic ions inside the cell. Moreover, the resulting equilibrium leaves a potential difference across the cell membrane of about 70 to 100 millivolts (mV), with the intracellular fluid being more negative. This potential, called the **resting potential** of the cell, is depicted schematically in Figure 1.5.

Figure 1.6 illustrates a neuron with several incoming connections, and the potentials that occur at various locations. The figure shows the axon with a covering called a **myelin sheath.** This insulating layer is interrupted at various points by the **nodes of Ranvier.**

Excitatory inputs to the cell reduce the potential difference across the cell membrane. The resulting depolarization at the **axon hillock** alters the permeability of the cell membrane to sodium ions. As a result, there is a large influx

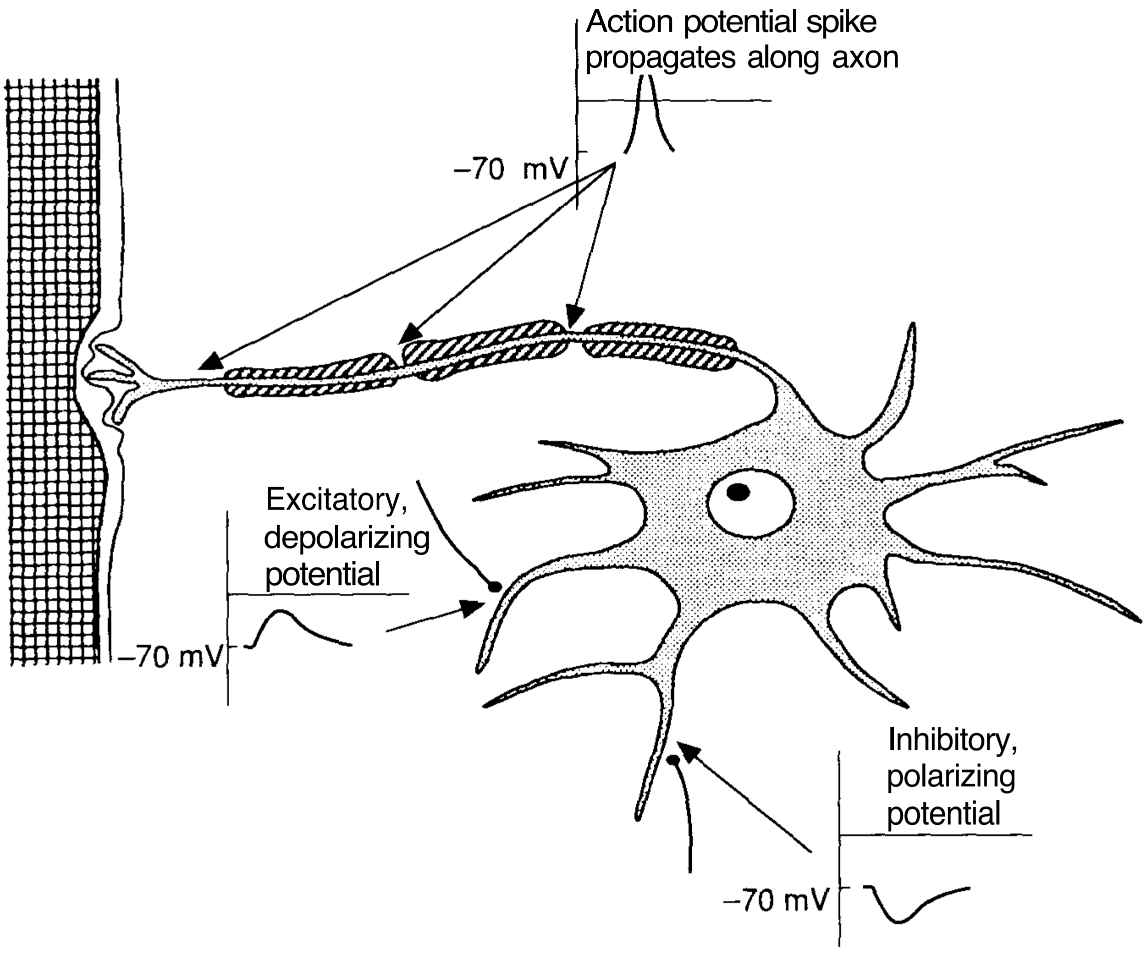

Action potential spike propagates along axon
-70 mV
Excitatory, depolarizing potential
-70 mV
Inhibitory, polarizing potential
-70 mV

Figure 1.6 Connections to the neuron from other neurons occur at various locations on the cell that are known as synapses. Nerve impulses through these connecting neurons can result in local changes in the potential in the cell body of the receiving neuron. These potentials, called graded potentials or input potentials, can spread through the main body of the cell. They can be either excitatory (decreasing the polarization of the cell) or inhibitory (increasing the polarization of the cell). The input potentials are summed at the axon hillock. If the amount of depolarization at the axon hillock is sufficient, an action potential is generated; it travels down the axon away from the main cell body.

of positive sodium ions into the cell, contributing further to the depolarization. This self-generating effect results in the **action potential.**

Nerve fibers themselves are poor conductors. The transmission of the action potential down the axon is a result of a sequence of depolarizations that occur at the nodes of Ranvier. As one node depolarizes, it triggers the depolarization of the next node. The action potential travels down the fiber in a discontinuous fashion, from node to node. Once an action potential has passed a given point,

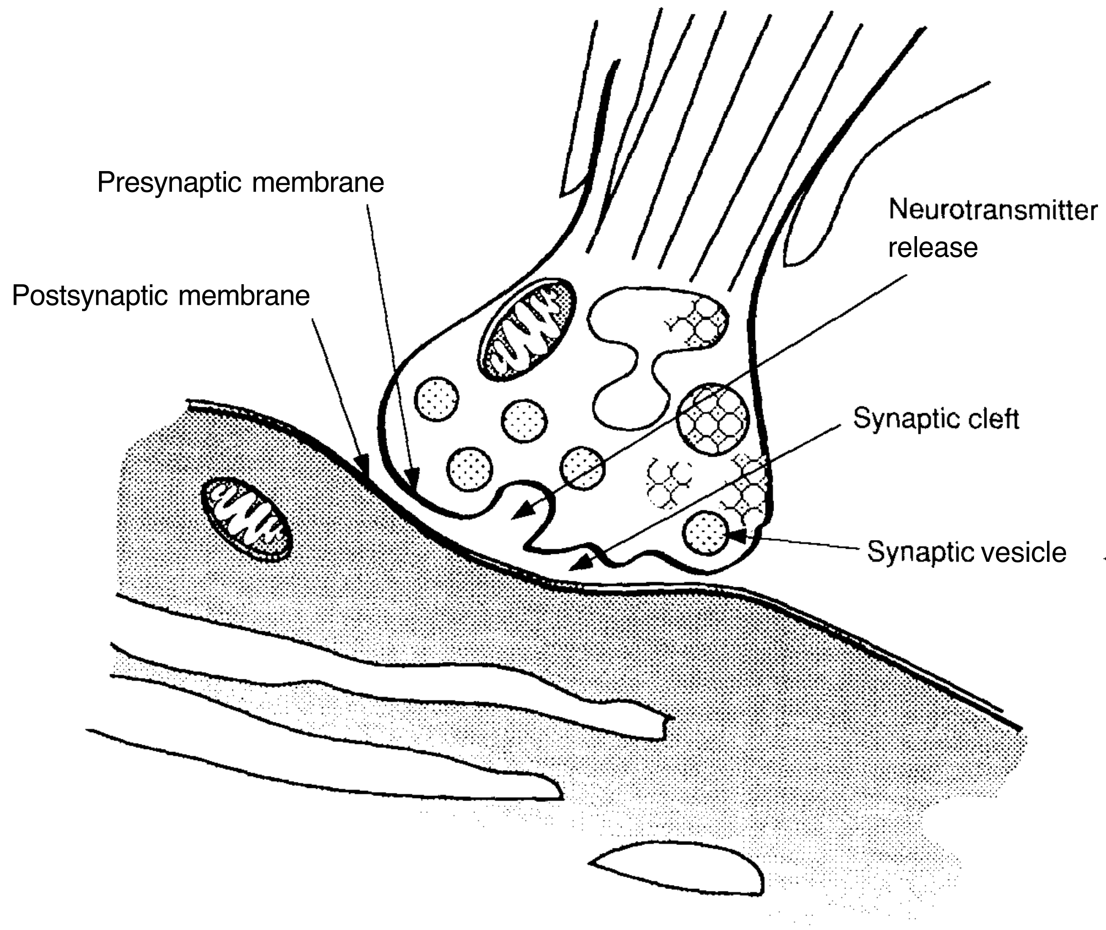


Figure 1.7 Neurotransmitters are held in vesicles near the presynaptic membrane. These chemicals are released into the synaptic cleft and diffuse to the postsynaptic membrane, where they are subsequently absorbed.

that point is incapable of being reexcited for about 1 millisecond, while it is restored to its resting potential. This **refractory period** limits the frequency of nerve-pulse transmission to about 1000 per second.

1.1.2 The Synaptic junction

Let's take a brief look at the activity that occurs at the connection between two neurons called the synaptic junction or **synapse.** Communication between neurons occurs as a result of the release by the presynaptic cell of substances called **neurotransmitters,** and of the subsequent absorption of these substances by the postsynaptic cell. Figure 1.7 shows this activity. When the action potential arrives as the presynaptic membrane, changes in the permeability of the membrane cause an influx of calcium ions. These ions cause the vesicles containing the neurotransmitters to fuse with the presynaptic membrane and to release their neurotransmitters into the synaptic cleft.

The neurotransmitters diffuse across the junction and join to the postsynaptic membrane at certain receptor sites. The chemical action at the receptor sites results in changes in the permeability of the postsynaptic membrane to certain ionic species. An influx of positive species into the cell will tend to depolarize the resting potential; this effect is excitatory. If negative ions enter, a hyperpolarization effect occurs; this effect is inhibitory. Both effects are local effects that spread a short distance into the cell body and are summed at the axon hillock. If the sum is greater than a certain threshold, an action potential is generated.

1.1.3 Neural Circuits and Computation

Figure 1.8 illustrates several basic neural circuits that are found in the central nervous system. Figures 1.8(a) and (b) illustrate the principles of divergence and convergence in neural circuitry. Each neuron sends impulses to many other neurons (divergence), and receives impulses from many neurons (convergence). This simple idea appears to be the foundation for all activity in the central nervous system, and forms the basis for most neural-network models that we shall discuss in later chapters.

Notice the feedback paths in the circuits of Figure 1.8(b), (c), and (d). Since synaptic connections can be either excitatory or inhibitory, these circuits facilitate control systems having either positive or negative feedback. Of course, these simple circuits do not adequately portray the vast complexity of neuroanatomy.

Now that we have an idea of how individual neurons operate and of how they are put together, we can pose a fundamental question: How do these relatively simple concepts combine to give the brain its enormous abilities? The first significant attempt to answer this question was made in 1943, through the seminal work by McCulloch and Pitts [24]. This work is important for many reasons, not the least of which is that the investigators were the first people to treat the brain as a *computational* organism.

The McCulloch–Pitts theory is founded on five assumptions:

1. The activity of a neuron is an all-or-none process.
2. A certain fixed number of synapses (> 1) must be excited within a period of latent addition for a neuron to be excited.
3. The only significant delay within the nervous system is synaptic delay.
4. The activity of any inhibitory synapse absolutely prevents excitation of the neuron at that time.
5. The structure of the interconnection network does not change with time.

Assumption 1 identifies the neurons as being binary: They are either on or off. We can therefore define a predicate, $N_i(t)$, which denotes the assertion that the ith neuron fires at time t. The notation, $\neg N_i(t)$, denotes the assertion that the ith neuron did not fire at time t. Using this notation, we can describe

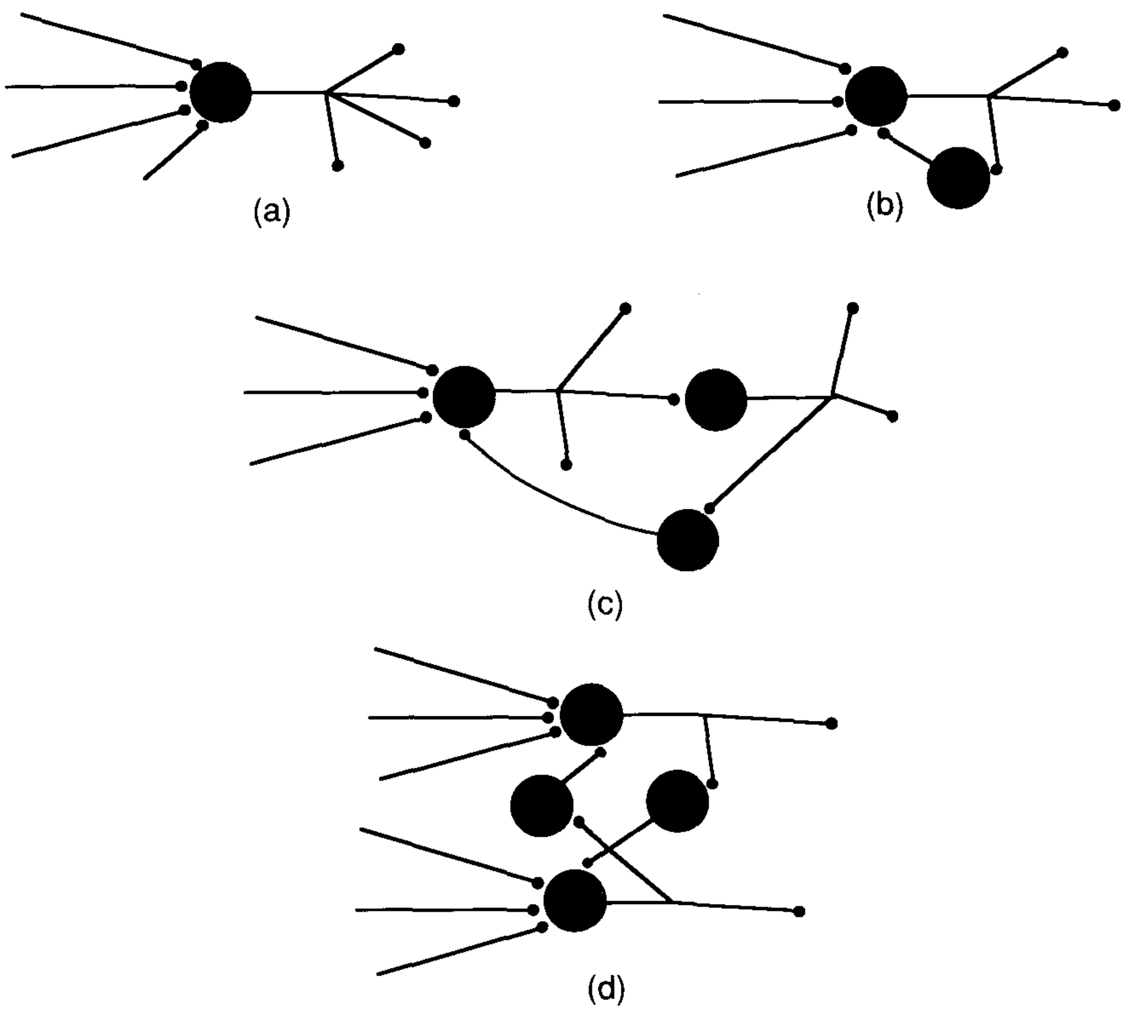


Figure 1.8 These schematics show examples of neural circuits in the central nervous system. The cell bodies (including the dendrites) are represented by the large circles. Small circles appear at the ends of the axons. Illustrated in (a) and (b) are the concepts of divergence and convergence. Shown in (b), (c), and (d) are examples of circuits with feedback paths.

the action of certain networks using propositional logic. Figure 1.9 shows five simple networks. We can write simple propositional expressions to describe the behavior of the first four (the fifth one appears in Exercise 1.1). Figure 1.9(a) describes precession: neuron 2 fires after neuron 1. The expression is $N_2(t) = N_1(t - 1)$. Similarly, the expressions for parts (b) through (d) of this figure are

- $N_3(t) = N_1(t - 1) \vee N_2(t - 1)$ (disjunction),
- $N_3(t) = N_1(t - 1) \& N_2(t - 1)$ (conjunction), and
- $N_3(t) = N_1(t - 1) \& \neg N_2(t - 1)$ (conjoined negation).

One of the powerful proofs in this theory was that *any* network that does not have feedback connections can be described in terms of combinations of these four

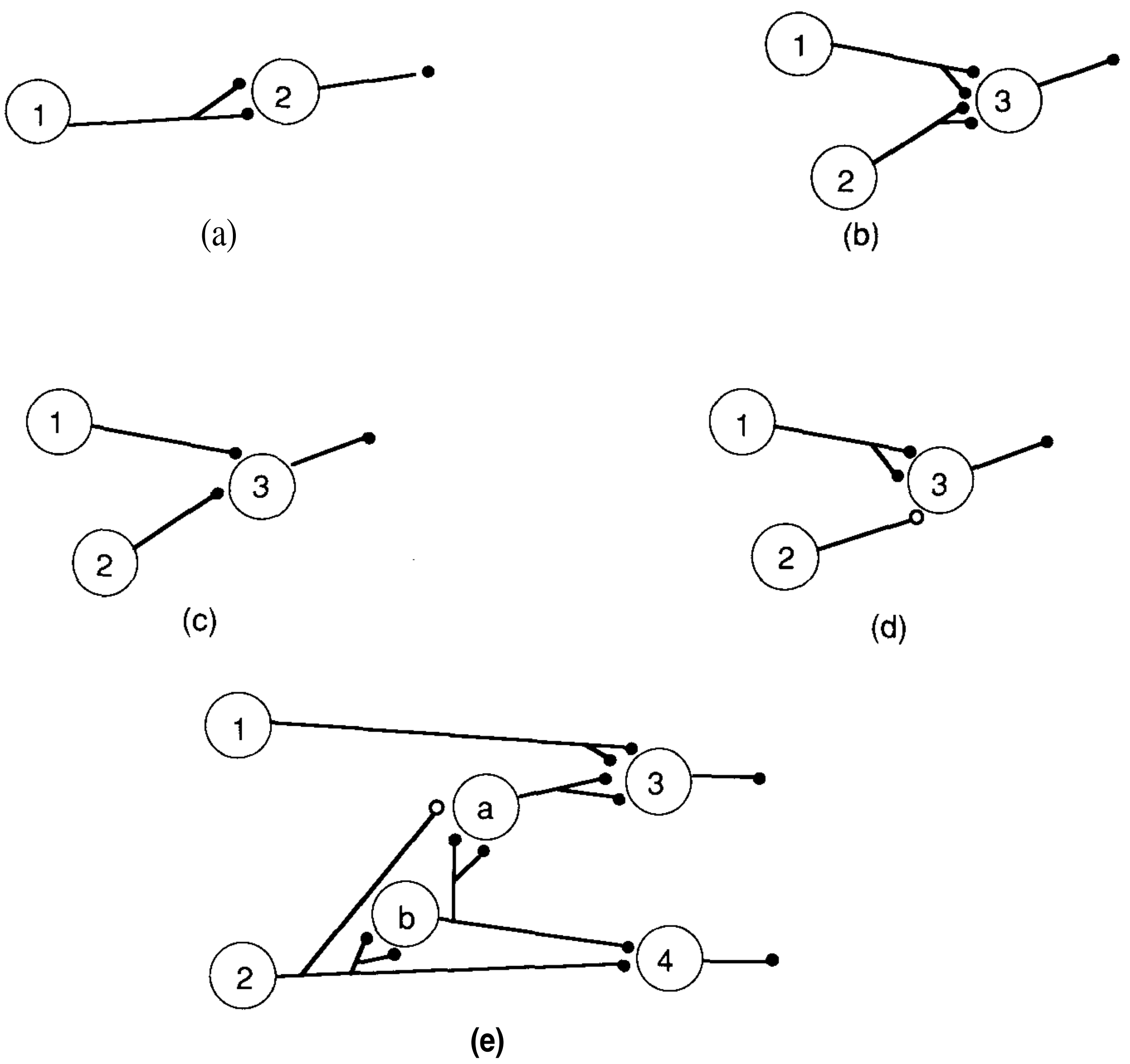


Figure 1.9 These drawings are examples of simple McCulloch-Pitts networks that can be defined in terms of the notation of propositional logic. Large circles with labels represent cell bodies. The small, filled circles represent excitatory connections; the small, open circles represent inhibitory connections. The networks illustrate (a) precession, (b) disjunction, (c) conjunction, and (d) conjoined negation. Shown in (e) is a combination of networks (a)-(d).

simple expressions, and vice versa. Figure 1.9(e) is an example of a network made from a combination of the networks in parts (a) through (d).

Although the McCulloch-Pitts theory has turned out not to be an accurate model of brain activity, the importance of the work cannot be overstated. The theory helped to shape the thinking of many people who were influential in the development of modern computer science. As Anderson and Rosenfeld point out, one critical idea was left unstated in the McCulloch-Pitts paper: Although neurons are simple devices, great computational power can be realized when these neurons are suitably connected and are embedded within the nervous system [2].

Exercise 1.1: Write the propositional expression for $N_3(t)$ and $N_4(t)$, of Figure 1.9(e).

Exercise 1.2: Construct McCulloch–Pitts networks for the following expressions:

1. $N_3(t) = N_2(t-2) \& \neg N_1(t-3)$
2. $N_4(t) = [N_2(t-1) \& \neg N_1(t-1)] \vee [N_3(t-1) \& \neg N_1(t-1)]$
   $\vee [N_2(t-1) \& N_3(t-1)]$

1.1.4 Hebbian Learning

Biological neural systems are not born preprogrammed with all the knowledge and abilities that they will eventually have. A learning process that takes place over a period of time somehow modifies the network to incorporate new information.

In the previous section, we began to see how a relatively simple neuron might result in a sophisticated computational device. In this section, we shall explore a relatively simple learning theory that suggests an elegant answer to this question: How do we learn?

The basic theory comes from a 1949 book by Hebb, *Organization of Behavior*. The main idea was stated in the form of an assumption, which we reproduce here for historical interest:

> When an axon of cell *A* is near enough to excite a cell *B* and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that *A's* efficiency, as one of the cells firing *B*, is increased. [10, p. 50]

As with the McCulloch-Pitts model, this learning *law* does not tell the whole story. Nevertheless, it appears in one form or another in many of the neural-network models that exist today.

To illustrate the basic idea, we consider the example of classical conditioning, using the familiar experiment of Pavlov. Figure 1.10 shows three idealized neurons that participate in the process.

Suppose that the excitation of C, caused by the sight of food, is sufficient to excite B, causing salivation. Furthermore, suppose that, in the absence of additional stimulation, the excitation of A, resulting from hearing a bell, is not sufficient to cause the firing of B.

Let's allow C to cause B to fire by showing food to the subject, and *while B is still firing,* stimulate A by ringing a bell. Because B is still firing, A is now participating in the excitation of B, even though by itself A would be insufficient to cause B to fire. In this situation, Hebb's assumption dictates that some change occur between A and B, so that A's influence on B is increased.

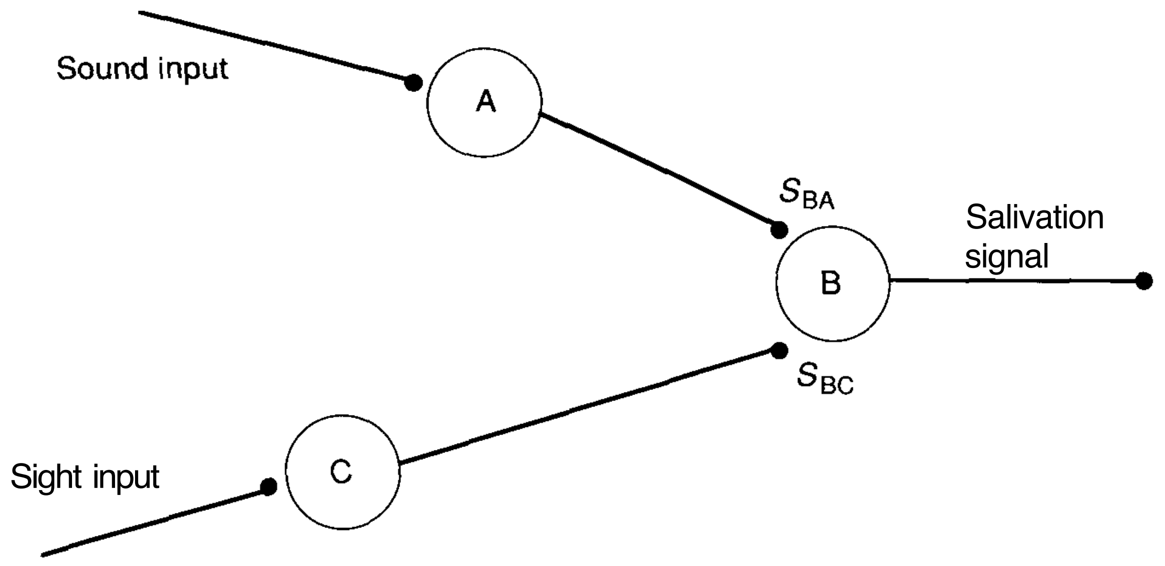


Figure 1.10 Two neurons, A and C, are stimulated by the sensory inputs of sound and sight, respectively. The third neuron, B, causes salivation. The two synaptic junctions are labeled S_{BA} and S_{BC}.

If the experiment is repeated often enough, A will eventually be able to cause B to fire *even in the absence of the visual stimulation from C*. Then, if the bell is rung, but no food is shown, salivation will still occur, because the excitation due to A alone is now sufficient to cause B to fire.

Because the connection between neurons is through the synapse, it is reasonable to guess that whatever changes occur during learning take place there. Hebb theorized that the area of the synaptic junction increased. More recent theories assert that an increase in the rate of neurotransmitter release by the presynaptic cell is responsible. In any event, changes certainly occur at the synapse. If either the pre- or postsynaptic cell were altered as a whole, other responses could be reinforced that are unrelated to the conditioning experiment.

Thus we conclude our brief look at neurophysiology. Before moving on, however, we reiterate a caution and issue a challenge to you. On the one hand, although there are many analogies between the basic concepts of neurophysiology and the neural-network models described in this book, we caution you not to portray these systems as actually modeling the brain. We prefer to say that these networks have been *inspired* by our current understanding of neurophysiology. On the other hand, it is often too easy for engineers, in their pursuit of solutions to specific problems, to ignore completely the neurophysiological foundations of the technology. We believe that this tendency is unfortunate. Therefore, we challenge ANS practitioners to keep abreast of the developments in neurobiology so as to be able to incorporate significant results into their systems. After all, what better model is there than the one example of a neural network with existing capabilities that far surpass any of our artificial systems?

Exercise 1.3: The analysis of high-dimensional data sets is often a complex task. One way to simplify the task is to use the **Karhunen–Loeve** (KL) matrix, which is defined as

$$F_{ij} = \frac{1}{N}\sum_{\mu=1}^{N} f_i^{\mu} f_j^{\mu}$$

where N is the number of vectors, and f_i^{μ} is the ith component of the μth vector. The KL matrix extracts the *principal components,* or directions of maximum information (correlation) from a data set. Determine the relationship between the KL formulation and the popular version of the Hebb rule known as the **Oja rule:**

$$\frac{d\phi_i(t)}{dt} = O(t)[I_i(t) - O(t)\phi_i(t)]$$

where $O(t)$ is the output of a simple, linear processing element; $I_i(t)$ are the inputs; and $\phi_i(t)$ are the synaptic strengths. (This exercise was suggested by Dr. Daniel Kammen, California Institute of Technology.)

1.2 FROM NEURONS TO ANS

In this section, we make a transition from some of the ideas gleaned from neurobiology to the idealized structures that form the basis of most ANS models. We first describe a general artificial neuron that incorporates most features we shall need for future discussions of specific models. Later in the section, we take a brief look at a particular example of an ANS called the perceptron. The perceptron was the result of an early attempt to simulate neural computation in order to perform complex tasks. We shall examine in particular what several limitations of this approach are and how they might be overcome.

1.2.1 The General Processing Element

The individual computational elements that make up most artificial neural-system models are rarely called *artificial neurons;* they are more often referred to as nodes, units, or processing elements (PEs). All these terms are used interchangeably throughout this book.

Another point to bear in mind is that it is not always appropriate to think of the processing elements in a neural network as being in a one-to-one relationship with actual biological neurons. It is sometimes better to imagine a single processing element as representative of the collective activity of a group of neurons. Not only will this interpretation help us to avoid the trap of speaking as though our systems were actual brain models, but also it will make the problem more tractable when we *are* attempting to model the behavior of some biological structure.

Figure 1.11 shows our general PE model. Each PE is numbered, the one in the figure being the ith. Having cautioned you not to make too many biological

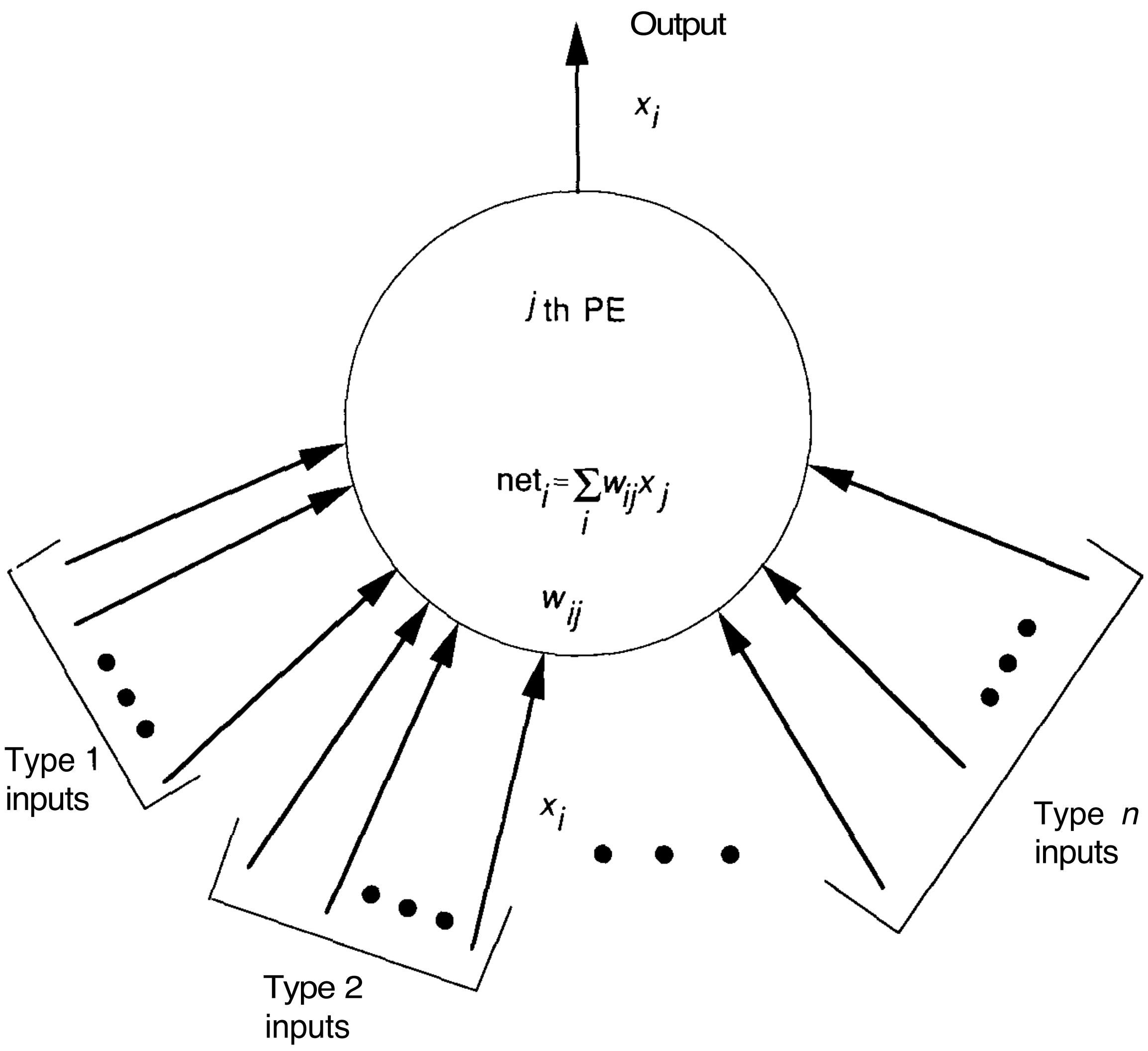


Figure 1.11 This structure represents a single PE in a network. The input connections are modeled as arrows from other processing elements. Each input connection has associated with it a quantity, w_{ij}, called a weight. There is a single output value, which can fan out to other units.

analogies, we shall now ignore our own advice and make a few ourselves. For example, like a real neuron, the PE has many inputs, but has only a single output, which can fan out to many other PEs in the network. The input the i**th** receives from the jth PE is indicated as *Xj* (note that this value is also the output of the jth node, just as the output generated by the i**th** node is labeled x_i). Each connection to the *i*th PE has associated with it a quantity called a **weight or connection strength.** The weight on the connection*from* the jth node *to* the ith node is denoted w_{ij}. All these quantities have analogues in the standard neuron model: The output of the PE corresponds to the firing frequency of the neuron, and the weight corresponds to the strength of the synaptic connection between neurons. In our models, these quantities will be represented as real numbers.

Notice that the inputs to the PE are segregated into various *types*. This segregation acknowledges that a particular input connection may have one of several effects. An input connection may be excitatory or inhibitory, for example. In our models, excitatory connections have positive weights, and inhibitory connections have negative weights. Other types are possible. The terms **gain, quenching,** and **nonspecific arousal** describe other, special-purpose connections; the characteristics of these other connections will be described later in the book. Excitatory and inhibitory connections are usually considered together, and constitute the most common forms of input to a PE.

Each PE determines a net-input value based on all its input connections. In the absence of special connections, we typically calculate the net input by summing the input values, gated (multiplied) by their corresponding weights. In other words, the net input to the *ith* unit can be written as

$$\text{net}_i = \sum_j x_j w_{ij} \tag{1.1}$$

where the index, j, runs over all connections to the PE. Note that excitation and inhibition are accounted for automatically by the sign of the weights. This sum-of-products calculation plays an important role in the network simulations that we will be describing later. Because there is often a very large number of interconnects in a network, the speed at which this calculation can be performed usually determines the performance of any given network simulation.

Once the net input is calculated, it is converted to an **activation value,** or simply **activation,** for the PE. We can write this activation value as

$$a_i(t) = F_i(a_i(t - 1), \text{net}_i(t)) \tag{1.2}$$

to denote that the activation is an explicit function of the net input. Notice that the current activation may depend on the previous value of the activation, $a(t - 1)$.[2] We include this dependence in the definition for generality. In the majority of cases, the activation and net input are identical, and the terms often are used interchangeably. Sometimes, activation and net input are not the same, and we must pay attention to the difference. For the most part, however, we will be able to use activation to mean net input, and vice versa.

Once the activation of the PE is calculated, we can determine the output value by applying an **output function:**

$$x_i = f_i(a_i) \tag{1.3}$$

Since, usually, $a_i = \text{net}_i$, this function is normally written as

$$x_i = f_i(\text{net}_i) \tag{1.4}$$

One reason for belaboring the issue of activation versus net input is that the term **activation function** is sometimes used to refer to the function, f_i, that

[2]Because of the emphasis on digital simulations in this text, we generally consider time to be measured in discrete steps. The notation $t - 1$ indicates one timestep prior to time t.

converts the net input value, net_i, to the node's output value, x_i. In this text, we shall consistently use the term *outputfunction* for $f_i()$ of Eqs. (1.3) and (1.4). Be aware, however, that the literature is not always consistent in this respect.

When we are describing the **mathematical** basis for network models, it will often be useful to think of the network as a **dynamical system**—that is, as a system that evolves over time. To describe such a network, we shall write differential equations that describe the time rate of change of the outputs of the various PEs. For example, $\dot{x}_i — g_i(x_i, net_i)$ represents a general differential equation for the output of the ith PE, where the dot above the x refers to differentiation with respect to time. Since net_i depends on the outputs of many other units, we actually have a system of coupled differential equations.

As an example, let's look at the equation

$$\pm i = -x_i + f_i(\text{net}_i)$$

for the output of the ith processing element. We apply some input values to the PE so that $\text{net}_i > 0$. If the inputs remain for a sufficiently long time, the output value will reach an equilibrium value, when $x_i = 0$, given by

$$x_i = f_i(\text{net}_i)$$

which is identical to Eq. (1.4). We can often assume that input values remain until equilibrium has been achieved.

Once the unit has a nonzero output value, removal of the inputs will cause the output to return to zero. If $\text{net}_i = 0$, then

$$\dot{x}_i = -x$$

which means that $x \longrightarrow 0$.

It is also useful to view the collection of weight values as a dynamical system. Recall the discussion in the previous section, where we asserted that learning is a result of the modification of the strength of synaptic junctions between neurons. In an ANS, learning usually is accomplished by modification of the weight values. We can write a system of differential equations for the weight values, $w_{ij} = G_i(w_{ij}, x_i, x_j, \ldots)$, where G_i represents the **learning law.** The learning process consists of finding weights that encode the knowledge that we want the system to learn. For most realistic systems, it is not easy to determine a closed-form solution for this system of equations. Techniques exist, however, that result in an acceptable approximation to a solution. Proving the existence of stable solutions to such systems of equations is an active area of research in neural networks today, and probably will continue to be so for some time.

1.2.2 Vector Formulation

In many of the network models that we shall discuss, it is useful to describe certain quantities in terms of vectors. Think of a neural network composed of several **layers** of identical processing elements. If a particular layer contains n

units, the outputs of that layer can be thought of as an n-dimensional vector, $X = (x_1, x_2, \cdots, x_n)^t$, where the t superscript means *transpose*. In our notation, vectors written in boldface type, such as x, will be assumed to be column vectors. When they are written row form, the transpose symbol will be added to indicate that the vector is actually to be thought of as a column vector. Conversely, the notation $\mathbf{x}^t$ indicates a row vector.

Suppose the n-dimensional output vector of the previous paragraph provides the input values to each unit in an m-dimensional layer (a layer with m units). Each unit on the m-dimensional layer will have n weights associated with the connections from the previous layer. Thus, there are m n-dimensional weight vectors associated with this layer; there is one n-dimensional weight vector for each of the m units. The weight vector of the ith unit can be written as $\mathbf{w}_i = (w_{i1}, w_{i2}, \cdots, w_{in})^t$. A superscript can be added to the weight notation to distinguish between weights on different layers.

The net input to the ith unit can be written in terms of the inner product, or dot product, of the input vector and the weight vector. For vectors *of equal dimensions,* the inner product is defined as the sum of the products of the corresponding components of the two vectors. In the notation of the previous section,

$$\text{net}_i = \sum_{j=1}^{n} x_j w_{ij}$$

where n is the number of connections to the ith unit. This equation can be written succinctly in vector notation as

$$\text{net}_i = \mathrm{x} \bullet \mathbf{w}_i$$

or

$$\text{net}_i = \mathbf{x}^t \mathbf{w}_i$$

Also note that, because of the rules of multiplication of vectors,

$$\mathbf{x}^t \mathbf{w}_i = \mathbf{w}_i^t \mathbf{x}$$

We shall often speak of input *vectors* and output *vectors* and weight *vectors,* but we tend to reserve the vector notation for cases where it is particularly appropriate. Additional vector concepts will be introduced later as needed. In the next section, we shall use the notation presented here to describe a neural-network model that has an important place in history: the perceptron.

1.2.3 The Perceptron: Part 1

The device known as the perceptron was invented by psychologist Frank Rosenblatt m the late 1950s. It represented his attempt to "illustrate some of the rundamental properties of intelligent systems in general, without becoming too

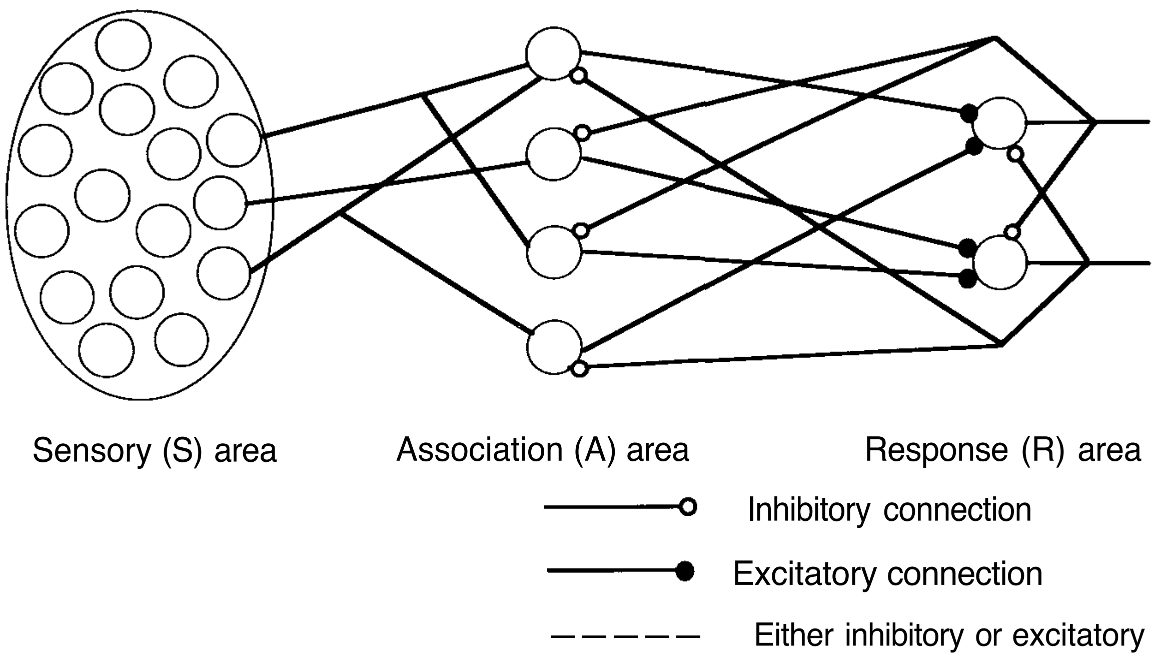


Figure 1.12 A simple photoperceptron has a sensory area, an association area, and a response area. The connections shown between units in the various areas are illustrative, and are not meant to be an exhaustive representation.

deeply enmeshed in the special, and frequently unknown, conditions which hold for particular biological organisms" [29, p. 387]. Rosenblatt believed that the connectivity that develops in biological networks contains a large random element. Thus, he took exception to previous analyses, such as the McCulloch–Pitts model, where symbolic logic was employed to analyze rather idealized structures. Rather, Rosenblatt believed that the most appropriate analysis tool was probability theory. He developed a theory of **statistical separability** that he used to characterize the gross properties of these somewhat randomly interconnected networks.

The **photoperceptron** is a device that responds to optical patterns. We show an example in Figure 1.12. In this device, light impinges on the **sensory** (S) **points** of the retina structure. Each S point responds in an all-or-nothing manner to the incoming light. Impulses generated by the S points are transmitted to the **associator** (A) **units** in the association layer. Each A unit is connected to a random set of S points, called the A unit's **source set,** and the connections may be either excitatory or inhibitory. The connections have the possible values, $+1$, -1, and 0. When a stimulus pattern appears on the retina, an A unit becomes active if the sum of its inputs exceeds some threshold value. If active, the A unit produces an output, which is sent to the next layer of units.

In a similar manner, A units are connected to **response** (R) **units** in the response layer. The pattern of connectivity is again random between the layers, but there is the addition of inhibitory feedback connections from the response

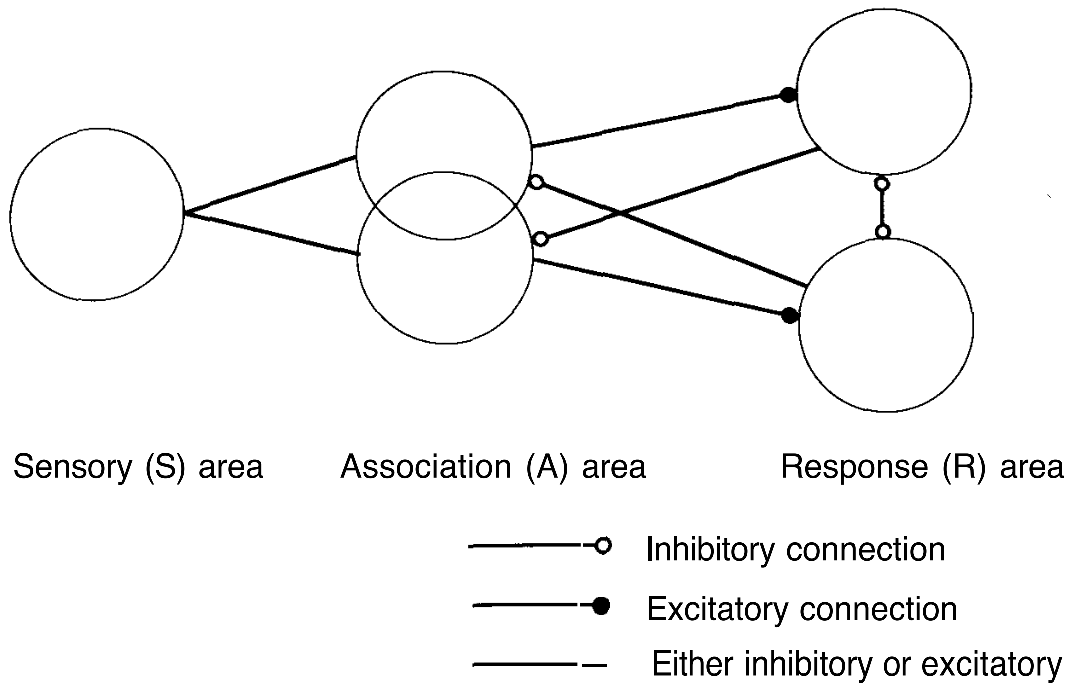


Figure 1.13 This Venn diagram shows the connectivity scheme for a simple perceptron. Each R unit receives excitatory connections from a group of units in the association area that is called the source set of the R unit. Notice that some A units are in the source set for both R units.

layer to the association layer, and of inhibitory connections between R units. The entire connectivity scheme is depicted in the form of a Venn diagram in Figure 1.13 for a simple perceptron with two R units.

This drawing shows that each R unit inhibits the A units in the complement to its own source set. Furthermore, each R unit inhibits the other. These factors aid in the establishment of a single, winning R unit for each stimulus pattern appearing on the retina. The R units respond in much the same way as do the A units. If the sum of their inputs exceeds a threshold, they give an output value of +1; otherwise, the output is —1. An alternative feedback mechanism would connect excitatory feedback connections from each R unit to that R unit's respective source set in the association layer.

A system such as the one just described can be used to classify patterns appearing on the retina into categories, according to the number of response units in the system. Patterns that are sufficiently similar should excite the same R unit. Thus, the problem is one of separability: Is it possible to construct a perceptron such that it can successfully distinguish between different pattern classes? The answer is "yes," but with certain conditions that we shall explore later.

The perceptron was a learning device. In its initial configuration, the perceptron was incapable of distinguishing the patterns of interest; through a *training* process, however, it could learn this capability. In essence, training involved a reinforcement process whereby the output of A units was either increased or decreased depending on whether or not the A units contributed to the correct response of the perceptron for a given pattern. A pattern was applied to the retina, and the stimulus was propagated through the layers until a response unit was activated. If the correct response unit was active, the output of the contributing A units was increased. If the incorrect R unit was active, the output of the contributing A units was decreased.

Using such a scheme, Rosenblatt was able to show that the perceptron could classify patterns successfully in what he termed a *differentiated environment,* where each class consisted of patterns that were in some sense similar to one another. The perceptron was also able to respond consistently to random patterns, but its accuracy diminished as the number of patterns that it attempted to learn increased.

Rosenblatt's work resulted in the proof of an important result known as the **perceptron convergence theorem.** The theorem is proved for a perceptron with one R unit that is learning to differentiate patterns of two distinct classes. It states, in essence, that, if the classification *can* be learned by the perceptron, then the procedure we have described guarantees that it *will* be learned in a finite number of training cycles.

Unfortunately, perceptrons caused a fair amount of controversy at the time they were described. Unrealistic expectations and exaggerated claims no doubt played a part in this controversy. The end result was that the field of artificial neural networks was almost entirely abandoned, except by a few die-hard researchers. We hinted at one of the major problems with perceptrons when we suggested that there were conditions attached to the successful operation of the perceptron. In the next section, we explore and evaluate these considerations.

Exercise 1.4: Consider a perceptron with one R unit and N_a association units, a_μ, which is attempting to learn to differentiate i patterns, S_i, each of which falls into one of two categories. For one category, the R unit gives an output of $+1$; for the other, it gives an output of -1. Let v_μ be the output of the μth A unit. Further, let ρ_i be ± 1, depending on the class of S_i, and let $e_{\mu i}$ be 1 if a_μ is in the source set for S_i, and 0 otherwise. Show that the successful classification of patterns Si requires that the following condition be satisfied:

$$\sum_{\mu=1}^{N_a} v_\mu e_{\mu i} \rho_i > \theta$$

where θ is the threshold value of the R unit.

1.2.4 The Perceptron: Part 2

In 1969, a book appeared that some people consider to have sounded the death knell for neural networks. The book was aptly entitled *Perceptrons: An Introduction to Computational Geometry* and was written by Marvin Minsky and

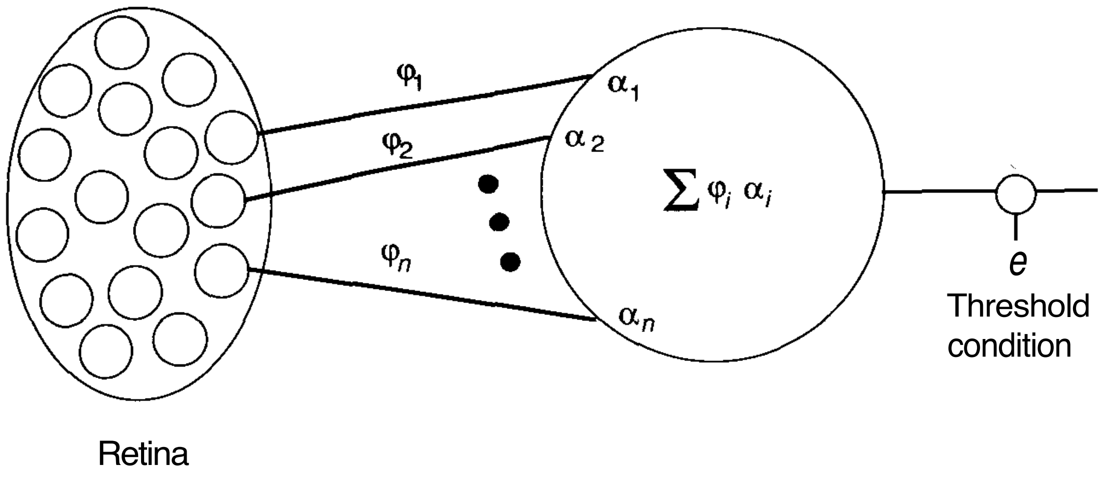


Figure 1.14 The simple perceptron structure is similar in structure to the general processing element shown in Figure 1.11. Note the addition of a threshold condition on the output. If the net input is greater than the threshold value, the output of the device is +1; otherwise, the output is 0.

Seymour Papert, both of MIT [26]. They presented an astute and detailed analysis of the perceptron in terms of its capabilities and limitations. Whether their intention was to defuse popular support for neural-network research remains a matter for debate. Nevertheless, the analysis is as timely today as it was in 1969, and many of the conclusions and concerns raised continue to be valid.

In particular, one of the points made in the previous section—a point treated in detail in Minsky and Papert's book—is the idea that there are certain restrictions on the class of problems for which the perceptron is suitable. Perceptrons can differentiate patterns only if the patterns are **linearly separable.** The meaning of the term *linearly separable* should become clear shortly. Because many classification problems do not possess linearly separable classes, this condition places a severe restriction on the applicability of the perceptron.

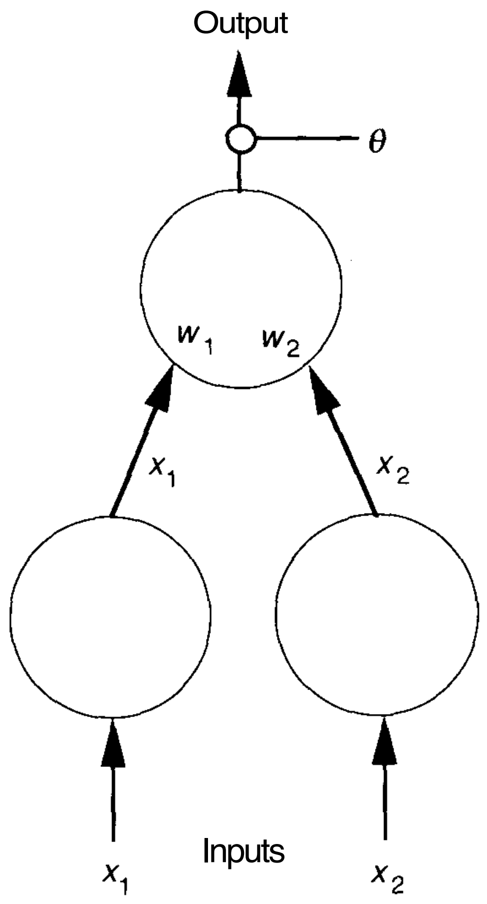
Minsky and Papert departed from the probabilistic approach championed by Rosenblatt, and returned to the ideas of predicate calculus in their analysis of the perceptron. Their idealized perceptron appears in Figure 1.14.

The set $\$ = \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ is a set of predicates. In the predicates' simplest form, $\varphi_i = 1$ if the ith point of the retina is on, and *(pi* — 0 otherwise. Each of the input predicates is weighted by a number from the set $\{\alpha_{\varphi_1}, \alpha_{\varphi_2}, \bullet\bullet., \alpha_{\varphi_n}\}\bullet$ The output, Ψ, is 1 if and only if $\sum_n \alpha_{\varphi_n} \varphi_n > \Theta$, where © is the threshold value.

One of the simplest examples of a problem that cannot be solved by a perceptron is the XOR problem. This problem is illustrated in Figure 1.15.

In the network of Figure 1.15, the output function of the output unit is a threshold function

$$f(\text{net}) = \begin{cases} 1 & \text{net} \geq 0 \\ 0 & \text{net} < 0 \end{cases}$$



x_1	x_2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Figure 1.15 This two-layer network has two nodes on the input layer with input values x\ and x_2 that can take on values of 0 or 1. We would like the network to be able to respond to the inputs such that the output *o* is the XOR function of the inputs, as indicated in the table.

where *0* is the threshold value. This type of node is called a **linear threshold unit.**

The output-node activation is

$$\text{net} = w_1x_1 + W2X2$$

and the output value *o* is

$$o = f(\text{net}) = \begin{cases} 1 & w_1x_1 + w_2x_2 > o \\ 0 & w_1x_1 + w_2x_2 < 0 \end{cases}$$

The problem is to select values of the weights such that each pair of input values results in the proper output value. This task cannot be done.

Let's look at the equation

$$6 = w_1x_1 + w_2x_2 \tag{1.5}$$

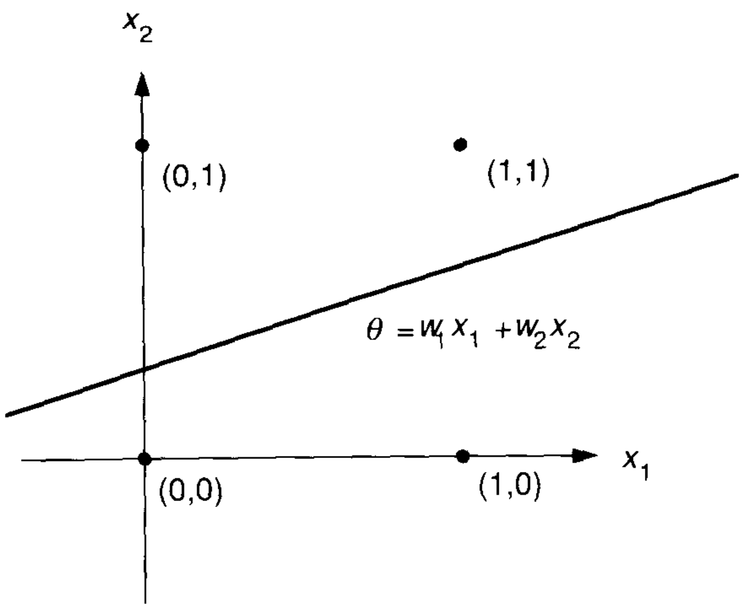


Figure 1.16 This figure shows the x_1, x_2 plane with the four points, $(0,0), (1,0), (0, 1)$, and (1. 1), which make up the four input vectors for the XOR problem. The line $9 — w_1x_1 + w_2x_2$ divides the plane into two regions but cannot successfully isolate the set of points $(0,0)$ and (1, 1) from the points $(0, 1)$ and $(1, 0)$.

This equation is the equation of a line in the x_1, x_2 plane. That plane is illustrated in Figure 1.16, along with the four points that are the possible inputs to the network. We can think of the problem as one of subdividing this space into regions and then attaching labels to the regions that correspond to the right answer for points in that region. We plot Eq. (1.5) for some values of 0, w_1, and w_2, as in Figure 1.16. The line can separate the plane into at most two distinct regions. We can then classify points in one region as belonging to the class having an output of 1, and those in the other region as belonging to the class having an output of 0; however, there is no way to arrange the position of the line so that the *correct* two points for each class both lie in the same region. (Try it.) The simple linear threshold unit cannot correctly perform the XOR function.

Exercise 1.5: A linear node is one whose output is equal to its activation. Show that a network such as the one in Figure 1.15, but with a linear output node, also is incapable of solving the XOR problem.

Before showing a way to overcome this difficulty, we digress for a moment to introduce the concept of **hyperplanes.** This idea shows up occasionally in the literature and can be useful in the evaluation of the performance of certain neural networks. We have already used the concept to analyze the XOR problem.

In familiar three-dimensional space, a plane is an object of two dimensions. A single plane can separate three-dimensional space into two distinct regions; two planes can result in three or four distinct regions, depending on their relative orientation, and so on. By extension, in an n-dimensional space, **hyperplanes** are objects of $n - 1$ dimensions. (An n-dimensional space is usually referred to as a **hyperspace.**) Suitable arrangement of hyperplanes allows an n-dimensional space to be partitioned into various distinct regions.

Many real problems involve the separation of regions of points in a hyperspace into individual categories, or classes, which must be distinguished from other classes. One way to make these distinctions is to select hyperplanes that separate the hyperspace into the proper regions. This task might appear difficult to perform in a high-dimensional space (higher than two, that is) — and it is. Fortunately, as we shall see later, certain neural networks can *learn* the proper partitioning, so we don't have to figure it out in advance.

In a general n-dimensional space, the equation of a hyperplane can be written as

$$\sum_{i=1}^{n} a_i x_i = C$$

where the a_is and C are constants, with at least one $a_i \neq 0$, and the x_is are the coordinates of the space.

Exercise 1.6: What are the general equations for the hyperplanes in two- and three-dimensional spaces? What geometric figures do these equations describe?

Let's return to the XOR problem to see how we might approach a solution. The graph in Figure 1.16 suggests that we could partition the space correctly if we had three regions. One region would belong to one output class, and the other two would belong to the second output class. There is no reason why disjoint regions cannot belong to the same class. Figure 1.17 shows a network of linear threshold units that performs the proper partitioning, along with the corresponding hyperspace diagram. You should verify that the network does indeed give the correct results.

The addition of the two hidden-layer, or middle-layer, units gave the network the needed flexibility to solve the problem. In fact, the existence of this hidden layer gives us the ability to construct networks that can solve complex problems.

This simple example is not intended to imply that all criticisms of the perceptron could be answered by the addition of hidden layers in the structure. It is intended to suggest that the technology continues to evolve toward systems with increasingly powerful computational abilities. Nevertheless, many concerns raised by Minsky and Papert should not be dismissed lightly. You can refer to the epilog of the 1988 reprinting of their book for a synopsis [27]. We briefly describe a second concern here.

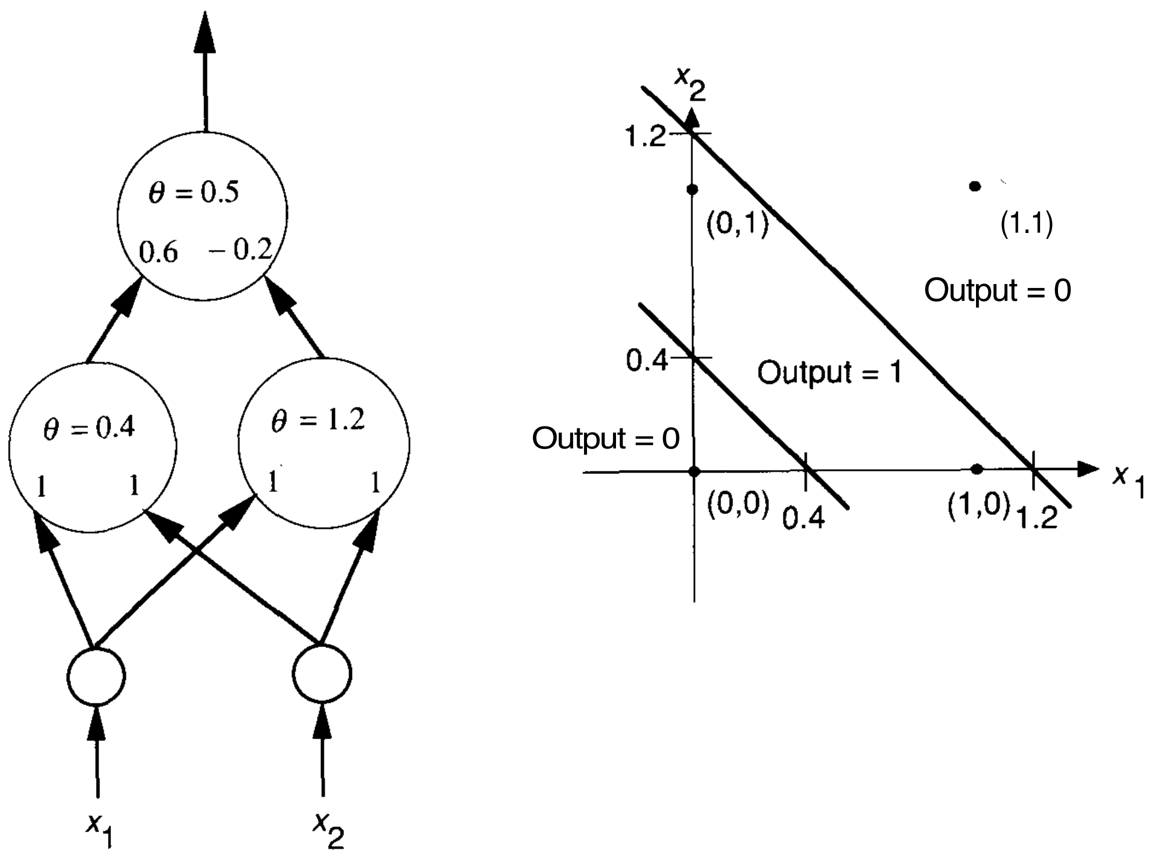


Figure 1.17 This network successfully solves the XOR problem. The hidden layer provides for two lines that can be used to separate the plane into three regions. The two regions containing the points $(0,0)$ and $(1,1)$ are associated with a network output of 0. The central region is associated with a network output of 1.

The subject of that concern is *scaling*. Many demonstrations of neural networks rely on the solution of what Minsky and Papert call *toy* problems—that is, problems that are only shadows of real-world items. Moving from these toy problems to real-world problems is often thought to be only a matter of time; we need only to wait until bigger, faster networks can be constructed. Several of the examples used in this text fall into the category of toy problems. Minsky and Papert claim that many networks suffer undesirable effects when scaled up to a large size. We raise this particular issue here, not because we necessarily believe that scale-up problems will defeat us, but because we wish to call attention to scaling as an issue that still must be resolved. We suspect that scaling problems do exist, but that there is a solution—perhaps one suggested by the architecture of the brain itself.

It seems plausible to us (and to Minsky and Papert) that the brain is composed of many different parallel, distributed systems, performing well-defined functions, but under the control of a serial-processing system at one or more levels. To address the issue of scaling, we may need to learn how to combine small networks and to place them under the control of other networks.

Of course, a "small" network in the brain challenges our current simulation capabilities, so we do not know exactly what the limitations are. The technology, although over 30 years old at this writing, is still emerging and deserves close scrutiny. We should always be aware of both the strengths and the limitations of our tools.

1.3 ANS SIMULATION

We will now consider several techniques for simulating ANS processing models using conventional programming methodologies. After presenting the design guidelines and goals that you should consider when implementing your own neural-network simulators, we will discuss the data structures that will be used throughout the remainder of this text as the basis for the network-simulation algorithms presented as a part of each chapter.

1.3.1 The Need for ANS Simulation

Most of the ANS models that we will examine in subsequent chapters share the basic concepts of distributed and highly interconnected PEs. Each network model will build on these simple concepts, implementing a unique learning law, an interconnection scheme (e.g., fully interconnected, sparsely interconnected, unidirectional, and bidirectional), and a structure, to provide systems that are tailored to specific kinds of problems.

If we are to explore the possibilities of ANS technology, and to determine what its *practical* benefits and limitations are, we must develop a means of testing as many as possible of these different network models. Only then will we be able to determine accurately whether or not an ANS can be used to solve a particular problem. Unfortunately, we do not have access to a computer system designed specifically to perform massively parallel processing, such as is found in all the ANS models we will study. However, we do have access to a tool that can be programmed rapidly to perform any type of algorithmic process, including simulation of a parallel-processing system. This tool is the familiar sequential computer.

Because we will study several different neural-network architectures, it is important for us to consider the aspects of code *portability* and *reusability* early in the implementation of our simulator. Let us therefore focus our attention on the characteristics common to most of the ANS models, and implement those characteristics as data structures that will allow our simulator to migrate to the widest variety of network models possible. The processing that is unique to the different neural-network models can then be implemented to use the data structures we will develop here. In this manner, we reduce to a minimum the amount of reprogramming needed to implement other network models.

1.3.2 Design Guidelines for Simulators

As we begin simulating neural networks, one of the first observations we will usually make is that it is necessary to design the simulation software such that the network can be sized dynamically. Even when we use only one of the network models described in this text, the ability to specify the number of PEs needed "on the fly," and in what organization, is paramount. The justification for this observation is based on the idea that it is not desirable to have to reprogram and recompile an ANS application simply because you want to change the network size. Since dynamic memory-allocation tools exist in most of the current generation of programming languages, we will use them to implement the network data structures.

The next observation you will probably make when designing your own simulator is that, at run time, the computer's central processing unit (CPU) will spend most of its time in the computation of the net_i, the input-activation term described earlier. To understand why this is so, consider how a uniprocessor computer will simulate a neural network. A program will have to be written to allow the CPU to *time multiplex* between units in the network; that is, each unit in the ANS model will share the CPU for some period. As the computer visits each node, it will perform the input computation and output translation function before moving on to the next unit. As we have already seen, the computation that produces the net_i value at each unit is normally a sum-of-products calculation—a very time-consuming operation if there is a large number of inputs at each node.

Compounding the problem, the sum-of-products calculation is done using floating-point numbers, since the network simulation is essentially a digital representation of analog signals. Thus, the CPU will have to perform two floating-point operations (a multiply and an add) for every input to each unit in the network. Given the large number of nodes in some networks, each with potentially hundreds or thousands of inputs, it is easy to see that the computer must be capable of performing several million floating-point operations per second (MFLOPS) to simulate an ANS of moderate size in a reasonable amount of time. Even assuming the computer has the floating-point hardware needed to improve the performance of the simulator, we, as programmers, must optimize the computer's ability to perform this computation by designing our data structures appropriately.

We now offer a final guideline for those readers who will attempt to implement many different network models using the data structures and processing concepts presented here. To a large extent, our simulator design philosophy is based on networks that have a uniform interconnection strategy; that is, units in one layer have all been fully connected to units in another layer. However, many of the networks we present in this text will rely on different interconnection schemes. Units may be only sparsely interconnected, or may have connections to units outside of the next sequential layer. We must take these notions into account as we define our data structures, or we may well end up with a unique set of data structures for each network we implement.

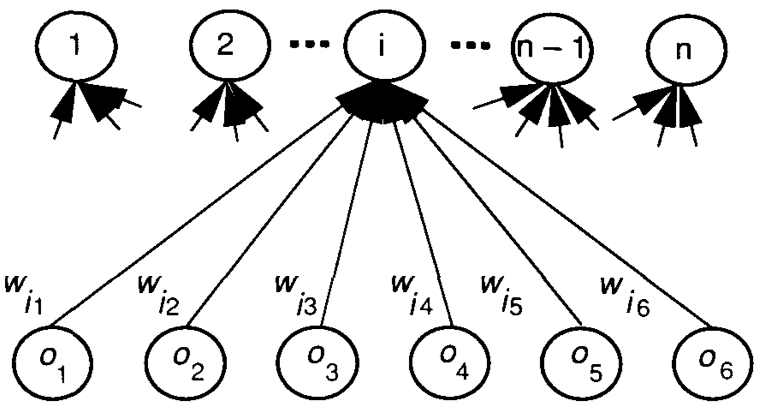


Figure 1.18 A two-layer network illustrating signal propagation is illustrated here. Each unit on the input layer generates a single output signal that is propagated through the connecting weights to each unit on the subsequent layer. Note that for each second-layer unit, the connection weights to the input layer can be modeled as a sequential array (or list) of values.

1.3.3 Array-Based ANS Data Structures

The observation made earlier that data will be processed as a sum of products (or as the *inner product* between two vectors) implies that the network data ought to be arranged in groups of linearly sequential arrays, each containing homogeneous data. The rationale behind this arrangement is that it is much faster to step through an array of data sequentially than it is to have to look up the address of every new value, as would be done if a linked-list approach were used. This grouping also is much more memory efficient than is a linked-list data structure, since there is no need to store pointers in the arrays. However, this efficiency is bought at the expense of algorithm generality, as we shall show later.

As an illustration of why arrays are more efficient than are linked records, consider the neural-network model shown in Figure 1.18. The input value present at the ith node in the upper layer is the sum of the modulated outputs received from every unit in the lower layer. To simulate this structure using data organized in arrays, we can model the connections and node outputs as values in two arrays, which we will call `weights` and `outputs` respectively.[3] The data in these arrays will be sequentially arranged so as to correspond one to one with the item being modeled, as shown in Figure 1.19. Specifically, the output from the first input unit will be stored in the first location in the `outputs` array, the second in the second, and so on. Similarly, the weight associated with the connection between the first input unit and the unit of interest, w_{i1}, will be

[3]Symbols that refer to variables, arrays, or code are identified in the text by the use of the typewriter typeface.

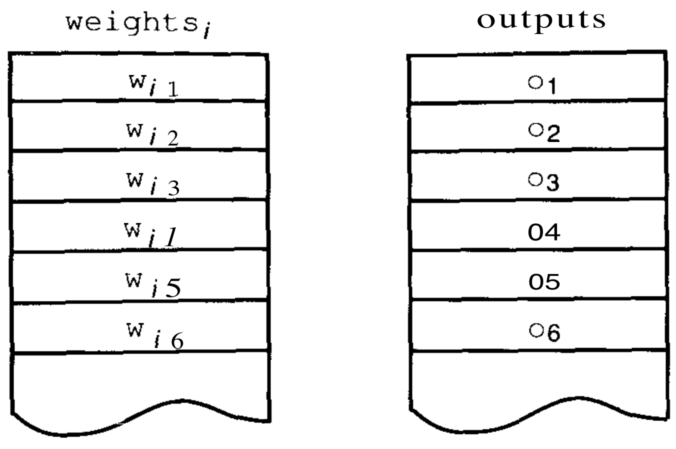


Figure 1.19 An array data structure is illustrated for the computation of the net, term. Here, we have organized the connection-weight values as a sequential array of values that map one to one to the array containing unit output values.

located as the first value in the ith weights array, weights [i]. Notice the index we now associate with the weights array. This index indicates that there will be many such arrays in the network, each containing a set of connection weights. The index here indicates that this array is one of these connection arrays—specifically, the one associated with the inputs to the ith network unit. We will expand on this notion later, as we extend the data structures to model a complete network.

The process needed to compute the aggregate input at the ith unit in the upper layer, net_i, is as follows. We begin by setting two pointers to the first location of the outputs and weights [i] arrays, and setting a local accumulator to zero. We then perform the computation by multiplying the values located in memory at each of the two array pointers, adding the resulting product to the local accumulator, incrementing both of the pointers, and repeating this sequence for all values in the arrays.

In most modern computer systems, this sequence of operations will compile into a two-instruction loop at the machine-code level (four instructions, if we count the compare and branch instructions needed to implement the loop), because the process of incrementing the array pointers can be done automatically as a part of the instruction-addressing mode. Notice that, if either of the arrays contains a structure of data as its elements, rather than a single value, the computation needed to access the next element in the array is no longer an *increment pointer* operation. Thus, the computer must execute additional instructions to compute the location of the next array value, as opposed to simply incrementing a register pointer as part of the instruction addressing mode. For small network applications, the overhead associated with these extra instructions is trivial. For applications that use very large neural networks, however, the overhead time needed for each connection, repeated hundreds of thousands (or perhaps millions) of times, can quickly overwhelm even a dedicated supercomputer. We therefore choose to emphasize efficiency in our simulator design; that is why we indicated earlier that the arrays ought to be constructed with homogeneous data.

This structure will do nicely for the general case of a fully interconnected network, but how can we adapt it to account for networks where the units are not fully interconnected? There are two strategies that can be employed to solve this dilemma:

- Implementation of a parallel index array to specify connectivity
- Use of a universal "zero" value to act as a placeholder connection

In the first case, an array with the same length as the weights [i] array is constructed and coexists with the weights [i] array. This array contains an integer index specifying the offset into the outputs array where the output from the transmitting unit is located. Such a structure, along with the network it describes, is illustrated in Figure 1.20. You should examine the diagram

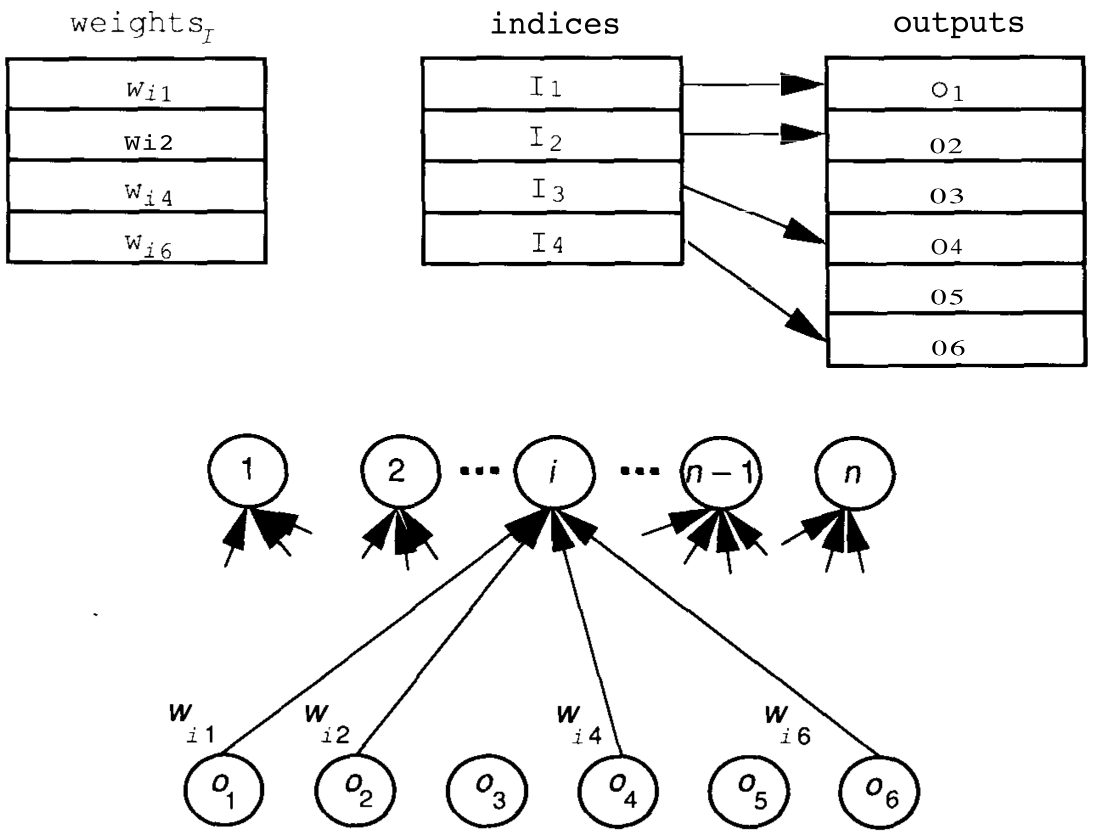


Figure 1.20 This sparse network is implemented using an index array. In this example, we calculate the input value at unit *i* by multiplying each value in the weights array with the value found in the output array at the offset indicated by the value in the indices *array.*

carefully to convince yourself that the data structure does implement the network structure shown.

In the second case, if we could specify to the network that a connection had a zero weight, the contribution to the total input of the node that it feeds would be zero. Therefore, the only reason for the existence of this connection would be that it acts as a placeholder, allowing the `weights` [i] array to maintain its one-to-one correspondence of location to connection. The cost of this implementation is the amount of time consumed performing a useless multiply-accumulate operation, and, in very sparsely connected networks, the large amount of wasted memory space. In addition, as we write the code needed to implement the learning law associated with the different network models, our algorithms must take a universal zero value into account and must not allow it to participate in the adaptation process; otherwise, the placeholder connection will be changed as the network adapts and eventually become an active participant in the signal-propagation process.

When is one approach preferable to the other? There is no absolute rule that will cover the wide variety of computers that will be the target machines for many ANS applications. In our experience, though, the break-even point is when the network is missing one-half of its interconnections. The desired approach therefore depends largely on how completely interconnected is the network that is being simulated. Whereas the "placeholder" approach consumes less memory and CPU time when only a relatively few connections are missing, the index array approach is much more efficient in very sparsely connected networks.

1.3.4 Linked-List ANS Data Structures

Many computer languages, such as Ada, LISP, and Modula-2, are designed to implement dynamic memory structures primarily as lists of records containing many different types of data. One type of data common to all records is the pointer type. Each record in the **linked list** will contain a pointer to the next record in the chain, thereby creating a threaded list of records. Each list is then completely described as a set of records that each contain pointers to other similar records, or contain null pointers. Linked lists offer a processing advantage in the algorithm generality they allow for neural-network simulation over the dynamic array structures described previously. Unfortunately, they also suffer from two disadvantages serious enough to limit their applicability to our simulator: excessive memory consumption and a significantly reduced processing rate for signal propagation.

To illustrate the disadvantages in the linked-list approach, we consider the two-layer network model and associated data structure depicted in Figure 1.21. In this example, each network unit is represented by an N_i record, and each connection is modeled as a *Cij* record. Since each connection is simultaneously part of two unique lists (the input list to a unit on the upper layer and the output list from a unit on the lower layer), each connection record must contain at least two separate pointers to maintain the lists. Obviously, just the memory needed

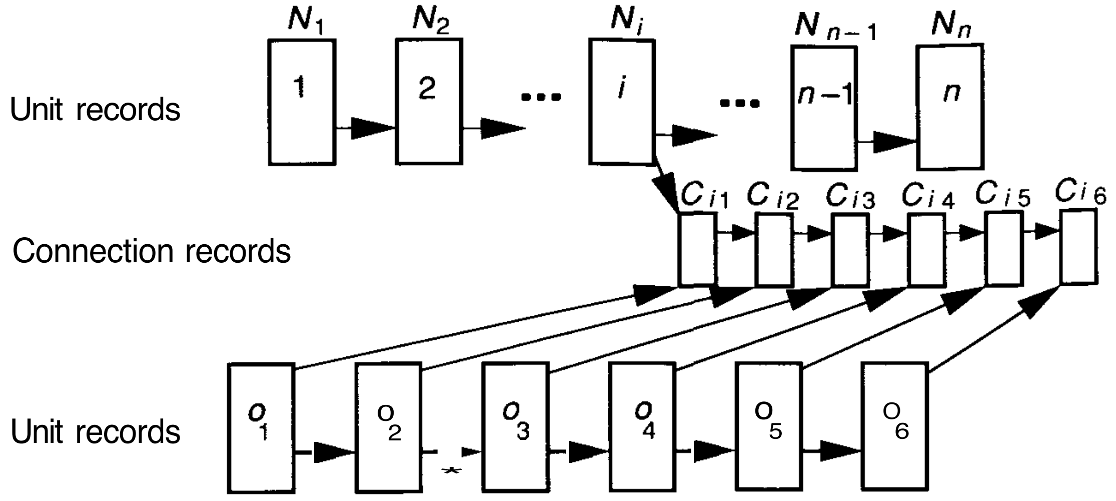


Figure 1.21 A linked-list implementation of a two-layer network is shown here. In this model, each network unit accesses its connections by following pointers from one connection record to the next. Here, the set of connection records modeling the input connections to unit N_i are shown, with links from all input-layer units.

to store those pointers will consume twice as much memory space as is needed to store the connection weight values. This need for extra memory results in roughly a three-fold reduction in the size of the network simulation that can be implemented when compared to the array-based model.[4] Similarly, the need to store pointers is not restricted to this particular structure; it is common to any linked-list data structure.

The linked-list approach is also less efficient than is the array model at run time. The CPU must perform many more data fetches in the linked-list approach (to fetch pointers), whereas in the array structure, the auto-postincrement addressing mode can be used to access the next connection implicitly. For very sparsely connected networks (or a very small network), this overhead is not significant. For a large network, however, the number of extra memory cycles required due to the large number of connections in the network will quickly overwhelm the host computer system for most ANS simulations.

On the bright side, the linked-list data structure is much more tolerant of "nonstandard" network connectivity schemes; that is, once the code has been written to enable the CPU to step through a standard list of input connections, no code modification is required to step through a nonstandard list. In this case, all the overhead is imposed on the software that constructs the original data structure for the network to be simulated. Once it is constructed, the CPU does

[4]This description is an obvious oversimplification, since it does not consider potential differences in the amount of memory used by pointers and floating-point numbers, virtual-memory systems, or other techniques for extending physical memory.

not know (or care) whether the connection list implements a fully interconnected network or a sparsely connected network. It simply follows the list to the end, then moves on to the next unit and repeats the process.

1.3.5 Extension of ANS Data Structures

Now that we have defined two possible structures for performing the input computations at each node in the network, we can extend these basic structures to implement an entire network. Since the array structure tends to be more efficient for computing input values at run time on most computers, we will implement the connection weights and node outputs as dynamically allocated arrays. Similarly, any additional parameters required by the different networks and associated with individual connections will also be modeled as arrays that coexist with the connection-weights arrays.

Now we must provide a higher-level structure to enable us to access the various instances of these arrays in a logical and efficient manner. We can easily create an adequate model for our integrated network structure if we adopt a few assumptions about how information is processed in a "standard" neural network:

- Units in the network can always be coerced into layers of units having similar characteristics, even if there is only one unit in some layers.
- All units in any layer must be processed completely before the CPU can begin simulating units in any other layer.
- The number of layers that our network simulator will support is indefinite, limited only by the amount of memory available.
- The processing done at each layer will usually involve the input connections to a node, and will only rarely involve output connections from a node (see Chapter 3 for an exception to this assumption).

Based on these assumptions, let us presume that the *layer* will be the network structure that binds the units together. Then, a layer will consist of a record that contains pointers to the various arrays that store the information about the nodes on that layer. Such a layer model is presented in Figure 1.22. Notice that, whereas the layer record will locate the node output array directly, the connection arrays are accessed indirectly through an intermediate array of pointers. The reason for this intermediate structure is again related to our desire to optimize the data structures for efficient computation of the net_i value for each node. Since each node on the layer will produce exactly one output, the outputs for all the nodes on any layer can be stored in a single array. However, each node will also have many input connections, each with weights unique to that node. We must therefore construct our data structures to allow input-weight arrays to be identified uniquely with specific nodes on the layer. The intermediate `weight_pointer` array satisfies the need to associate input weights with the appropriate node (via the position of the pointer in the intermediate array),

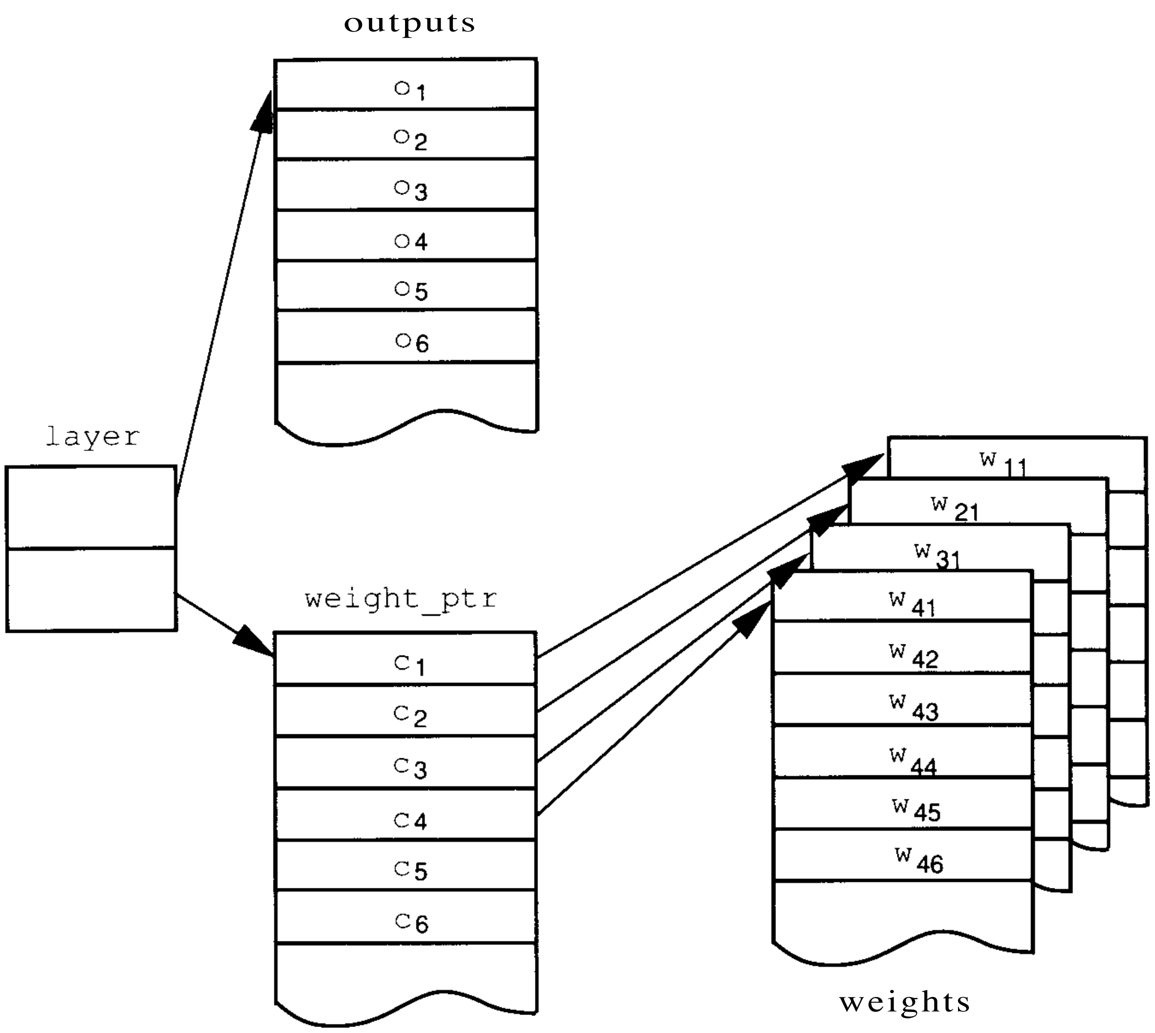


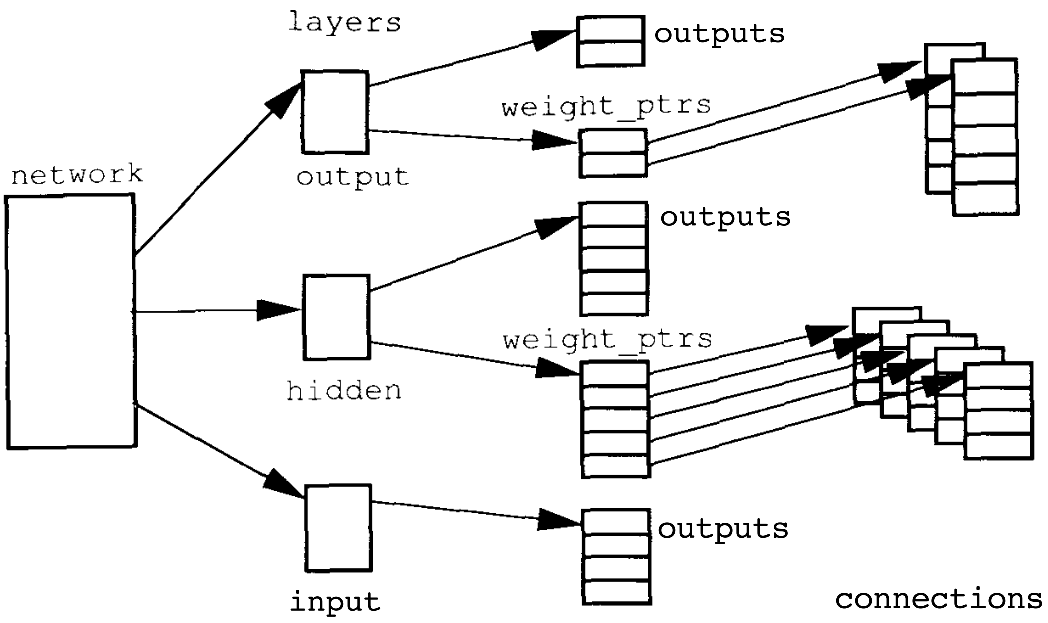
Figure 1.22 The layer structure is used to model a collection of nodes with similar function. In this example, the weight values of all input connections to the first processing unit (o_1) are stored sequentially in the w_{1j} *array,* connections to the second unit (o_2) in the w_{2j} *array,* and so on, enabling rapid sequential access to these values during the input computation operation.

while allowing the input weights for each node to be modeled as sequential arrays, thus maintaining the desired efficiency in the network data structures.

Finally, let us consider how we might model an entire network. Since we have decided that any network can be constructed from a set of layers, we will model the network as a record that contains both global data, and pointers to locate the first and last elements in a dynamically allocated array of layer records. This approach allows us to create a network of arbitrary depth while providing us with a means of immediate access to the two most commonly accessed layers in the network—the input and output layers.

Such a data structure, along with the network structure it represents, is depicted in Figure 1.23. By modeling the data in this way, we will allow for

(a)



(b)

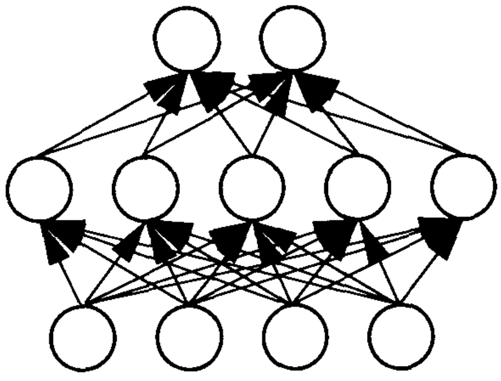


Figure 1.23 The figure shows a neural network (a) as implemented by our data structures, and (b) as represented schematically.

networks of arbitrary size and complexity, while optimizing the data structures for efficient run-time operation in the *inner loop*, the computation of the net_i term for each node in the network.

1.3.6 A Final Note on ANS Simulation

Before we move on to examine specific ANS models, we must mention that the earlier discussion of the ANS simulator data structures is meant to provide you with only an insight into how to go about simulating neural networks on conventional computers. We have specifically avoided any detailed discussion of the data structures needed to implement a simulator (such as might be found in a conventional computer-science textbook). Likewise, we have avoided any analysis of how much more efficient one technique may be over another. We have taken this approach because we believe that it is more important to convey

Syntax	Intended Meaning
`^type`	a pointer to an object type
`^type[]`	a pointer to the first object in an array
`record.slot`	access a field "slot" in a record
`pointer^.name`	access a field "name" through a pointer
`length(^type[])`	get number of items in the array
`{ text }`	curly braces enclose comments

Table 1.1 Pseudocode language definitions.

the ideas of *what* must be done to simulate an ANS model, than to advocate *how* to implement that model.

This philosophy carries through the remainder of the text as well, specifically in the sections in each chapter that describe how to implement the learning algorithms for the network being discussed. Rather than presenting algorithms that might indicate a preference for a specific computer language, we have opted to develop our own pseudocode descriptions for the algorithms.[5] We hope that you will have little difficulty translating our simulator algorithms to your own preferrred data structures and programming languages.

Part of the purpose of this text, however, is to illustrate the design of the algorithms needed to construct simulators for the various neural-network models we shall present. For that reason, the algorithms developed in this text will be rather detailed, perhaps more so than some people prefer. We have elected to use detailed algorithms so that we can illustrate, wherever possible, algorithmic enhancements that should be made to improve the performance of the simulator. However, with this detail comes a responsibility. Since we have elected to present *pseudocode* algorithms, we are obligated to develop a syntax that precisely describes the actions we intend the computer to perform. For that reason, you should become comfortable with the notations described in Table 1.1, as we will adhere to this syntax in the description of the various algorithms and data structures throughout the remainder of this text.

One final note for programmers who prefer the C language. You should be aware of the fundamental differences between the mathematical summations that will be described in the ANS-specific chapters that follow, and the C **for(;;)** construct. Although it is mathematically correct to describe a summation with an index, i, starting at 1 and running through n, in C it is preferable to use arrays that start at index 0 and run through $n - 1$. Those readers that will be writing simulators in C must account for this difference in their code, and should be alert to this difference as they read the theoretical sections of this text.

[5]We have thus ensured that we have offended everyone equally.

Suggested Readings

There are many books appearing that cover various aspects of ANS technology from widely differing perspectives. We shall not attempt to give an exhaustive bibliography here; rather, we shall indicate sources that we have found to be useful, for one reason or another.

Neurocomputing: Foundations ofResearch is an excellent source of papers having an historical interest, as well as of more modern works [2]. Perhaps the most widely read books are the PDP series, edited by Rumelhart and McClelland [23, 22]. Volume III of that series contains a disk with software that can be used for experiments with the technology. An earlier work, *Parallel Models ofAssociative Memory,* contains papers of the same type as those found in the PDP series [13]. We have also included in the bibliography other books, some that are collections and some that are monographs, on the general topic of ANS [1, 7, 6, 8, 12, 16, 25, 27, 28, 31]. On a less technical level is the recent work by Caudill and Butler, which provides an excellent review of the subject [5]. An excellent introduction to neurophysiology is given by Lavine [19]. This book presents the basic terminology and technical details of neurophysiology without the excruciating detail of a medical text. A more comprehensive review of neural modeling is given in the book by McGregor [21]. For a cognitive-psychology viewpoint, the works by Anderson and Baron are both well written and thought provoking [3, 4].

An excellent review article is that by Richard Lippmann [20]. This article gives an overview of several of the algorithms presented in later chapters of this text. You might also want to read the *Scientific American* article by David Tank and John Hopfield [30]. Two other well-written and informative review articles appear in the first edition of the journal *Neural Networks* [9, 18]. For a comparison between traditional classification techniques and neural-network classifiers, see the papers by Huang and Lippmann [15, 14]. You can get an idea of the types of applications to which neural-network technology may apply from the paper by Hecht-Nielsen [11].

Bibliography

[1] Igor Aleksander, editor. *Neural Computing Architectures.* MIT Press, Cambridge, MA, 1989.

[2] James A. Anderson and Edward Rosenfeld, editors. *Neurocomputing: Foundations ofResearch.* MIT Press, Cambridge, MA, 1988.

[3] John R. Anderson. *The Architecture ofCognition.* Cognitive Science Series. Harvard University Press, Cambridge, MA, 1983.

[4] Robert J. Baron. *The Cerebral Computer.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[5] Maureen Caudill and Charles Butler. *Naturally Intelligent Systems.* MIT Press, Cambridge, MA, 1990.

[6] John S. Denker, editor. *Neural Networks for Computing: AIP Conference Proceedings 151.* American Institute of Physics, New York, 1986.

[7] Rolf Eckmiller and Christoph v. d. Malsburg, editors. *Neural Computers.* NATO ASI Series F: Computer and Systems Sciences. Springer-Verlag, Berlin, 1988.

[8] Stephen Grossberg, editor. *Neural Networks and Natural Intelligence.* MIT Press, Cambridge, MA, 1988.

[9] Stephen Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural Networks,* 1(1):17-62, 1988.

[10] Donald O. Hebb. *The Organization ofBehavior.* Wiley, New York, 1949.

[11] Robert Hecht-Nielsen. Neurocomputer applications. In Rolf Eckmiller and Christoph v.d. Malsburg, editors, *Neural Computers,* pages 445–454. Springer-Verlag, 1988. NATO ASI Series F: Computers and System Sciences Vol. 41.

[12] Robert Hecht-Nielsen. *Neurocomputing.* Addison-Wesley, Reading, MA, 1990.

[13] Geoffrey E. Hinton and James A. Anderson, editors. *Parallel Models of Associative Memory.* Lawrence Erlbaum Associates, Hillsdale, NJ, 1981.

[14] Willian Y. Huang and Richard P. Lippmann. Comparison between neural net and conventional classifiers. In *Proceedings ofthe IEEE First International Conference on Neural Networks,* San Diego, CA, pp. iv.485-iv.494 June 1987.

[15] Willian Y. Huang and Richard P. Lippmann. Neural net and traditional classifiers. In *Proceedings of the Conference on Neural Information Processing Systems,* Denver, CO, November 1987.

[16] Tarun Khanna. *Foundations ofNeural Networks.* Addison-Wesley, Reading, MA, 1990.

[17] C. Klimasauskas. *The 1989 Neuro-Computing Bibliography,* MIT Press, Cambridge, MA, 1989.

[18] Teuvo Kohonen. An introduction to neural computing. *Neural Networks,* 1(1):3-16, 1988.

[19] Robert A. Lavine. *Neurophysiology: The Fundamentals.* The Collamore Press, Lexington, MA, 1983.

[20] Richard P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine,* pp. 4–22, April 1987.

[21] Ronald J. MacGregor. *Neural and Brain Modeling.* Academic Press, San Diego, CA, 1987.

[22] James McClelland and David Rumelhart. *Explorations in Parallel Distributed Processing*. MIT Press, Cambridge, MA, 1986.

[23] James McClelland and David Rumelhart. *Parallel Distributed Processing*, volumes 1 and 2. MIT Press, Cambridge, MA, 1986.

[24] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin ofMathematical Biophysics,* 5:115–133, 1943.

[25] Carver A. Mead and M. A. Mahowald. A silicon model of early visual processing. *Neural Networks,* 1(1):91-98, 1988.

[26] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, MA, 1969.

[27] Marvin Minsky and Seymour Papert. *Perceptrons: Expanded Edition*. MIT Press, Cambridge, MA, 1988.

[28] Yoh-Han Pao. *Adaptive Pattern Recognition and Neural Networks*. Addison-Wesley, Reading, MA, 1989.

[29] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review,* 65:386–408, 1958.

[30] David W. Tank and John J. Hopfield. Collective computation in neuronlike circuits. *Scientific American,* 257(6): 104–114, 1987.

[31] Philip D. Wasserman. *Neural Computing: Theory and Practice*. Van Nostrand Reinhold, New York, 1989.