CHAPTER 1

Introduction

1.1 WHY NEURAL NETWORKS AND WHY NOW?

As modern computers become ever more powerful, scientists continue to be challenged to use machines effectively for tasks that are relatively simple for humans. Based on examples, together with some feedback from a "teacher," we learn easily to recognize the letter *A* or distinguish a cat from a bird. More experience allows us to refine our responses and improve our performance. Although eventually, we may be able to describe rules by which we can make such decisions, these do not necessarily reflect the actual process we use. Even without a teacher, we can group similar patterns together. Yet another common human activity is trying to achieve a goal that involves maximizing a resource (time with one's family, for example) while satisfying certain constraints (such as the need to earn a living). Each of these types of problems illustrates tasks for which computer solutions may be sought.

Traditional, sequential, logic-based digital computing excels in many areas, but has been less successful for other types of problems. The development of artificial neural networks began approximately 50 years ago, motivated by a desire to try both to understand the brain and to emulate some of its strengths. Early

successes were overshadowed by rapid progress in digital computing. Also, claims made for capabilities of early models of neural networks proved to be exaggerated, casting doubts on the entire field.

Recent renewed interest in neural networks can be attributed to several factors. Training techniques have been developed for the more sophisticated network architectures that are able to overcome the shortcomings of the early, simple neural nets. High-speed digital computers make the simulation of neural processes more feasible. Technology is now available to produce specialized hardware for neural networks. However, at the same time that progress in traditional computing has made the study of neural networks easier, limitations encountered in the inherently sequential nature of traditional computing have motivated some new directions for neural network research. Fresh approaches to parallel computing may benefit from the study of biological neural systems, which are highly parallel. The level of success achieved by traditional computing approaches to many types of problems leaves room for a consideration of alternatives.

Neural nets are of interest to researchers in many areas for different reasons. Electrical engineers find numerous applications in signal processing and control theory. Computer engineers are intrigued by the potential for hardware to implement neural nets efficiently and by applications of neural nets to robotics. Computer scientists find that neural nets show promise for difficult problems in areas such as artificial intelligence and pattern recognition. For applied mathematicians, neural nets are a powerful tool for modeling problems for which the explicit form of the relationships among certain variables is not known.

There are various points of view as to the nature of a neural net. For example, is it a specialized piece of computer hardware (say, a VLSI chip) or a computer program? We shall take the view that neural nets are basically mathematical models of information processing. They provide a method of representing relationships that is quite different from Turing machines or computers with stored programs. As with other numerical methods, the availability of computer resources, either software or hardware, greatly enhances the usefulness of the approach, especially for large problems.

The next section presents a brief description of what we shall mean by a neural network. The characteristics of biological neural networks that serve as the inspiration for artificial neural networks, or *neurocomputing*, are also mentioned. Section 1.3 gives a few examples of where neural networks are currently being developed and applied. These examples come from a wide range of areas. Section 1.4 introduces the basics of how a neural network is defined. The key characteristics are the net's architecture and training algorithm. A summary of the notation we shall use and illustrations of some common activation functions are also presented. Section 1.5 provides a brief history of the development of neural networks. Finally, as a transition from the historical context to descriptions of the most fundamental and common neural networks that are the subject of the remaining chapters, we describe the McCulloch-Pitts neuron.

1.2 WHAT IS A NEURAL NET?

1.2.1 Artificial Neural Networks

An *artificial neural network* is an information-processing system that has certain performance characteristics in common with biological neural networks. Artificial neural networks have been developed as generalizations of mathematical models of human cognition or neural biology, based on the assumptions that:

1. Information processing occurs at many simple elements called neurons.
2. Signals are passed between neurons over connection links.
3. Each connection link has an associated weight, which, in a typical neural net, multiplies the signal transmitted.
4. Each neuron applies an activation function (usually nonlinear) to its net input (sum of weighted input signals) to determine its output signal.

A neural network is characterized by (1) its pattern of connections between the neurons (called its *architecture*), (2) its method of determining the weights on the connections (called its *training*, or *learning, algorithm*), and (3) its *activation function*.

Since *what* distinguishes (artificial) neural networks from other approaches to information processing provides an introduction to both *how* and *when* to use neural networks, let us consider the defining characteristics of neural networks further.

A neural net consists of a large number of simple processing elements called *neurons, units, cells,* or *nodes*. Each neuron is connected to other neurons by means of directed communication links, each with an associated weight. The weights represent information being used by the net to solve a problem. Neural nets can be applied to a wide variety of problems, such as storing and recalling data or patterns, classifying patterns, performing general mappings from input patterns to output patterns, grouping similar patterns, or finding solutions to constrained optimization problems.

Each neuron has an internal state, called its *activation* or *activity level,* which is a function of the inputs it has received. Typically, a neuron sends its activation as a signal to several other neurons. It is important to note that a neuron can send only one signal at a time, although that signal is broadcast to several other neurons.

For example, consider a neuron Y, illustrated in Figure 1.1, that receives inputs from neurons X_1, X_2, and X_3. The activations (output signals) of these neurons are x_1, x_2, and x_3, respectively. The weights on the connections from X_1, X_2, and X_3 to neuron Y are w_1, w_2, and w_3, respectively. The net input, y_in, to neuron Y is the sum of the weighted signals from neurons X_1, X_2, and X_3, i.e.,

$$y\_in = w_1x_1 + w_2x_2 + w_3x_3.$$

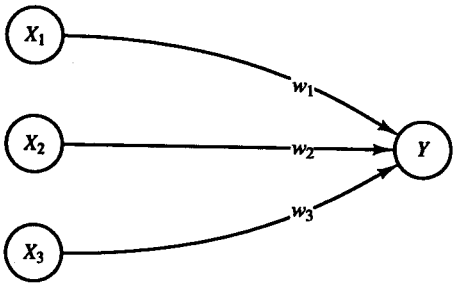

Figure 1.1 A simple (artificial) neuron.

The activation y of neuron Y is given by some function of its net input, $y = f(y\_in)$, e.g., the logistic sigmoid function (an S-shaped curve)

$$f(x) = \frac{1}{1 + \exp(-x)},$$

or any of a number of other activation functions. Several common activation functions are illustrated in Section 1.4.3.

Now suppose further that neuron Y is connected to neurons Z_1 and Z_2, with weights v_1 and v_2, respectively, as shown in Figure 1.2. Neuron Y sends its signal y to each of these units. However, in general, the values received by neurons Z_1 and Z_2 will be different, because each signal is scaled by the appropriate weight, v_1 or v_2. In a typical net, the activations z_1 and z_2 of neurons Z_1 and Z_2 would depend on inputs from several or even many neurons, not just one, as shown in this simple example.

Although the neural network in Figure 1.2 is very simple, the presence of a hidden unit, together with a nonlinear activation function, gives it the ability to solve many more problems than can be solved by a net with only input and output units. On the other hand, it is more difficult to train (i.e., find optimal values for the weights) a net with hidden units. The arrangement of the units (the architecture

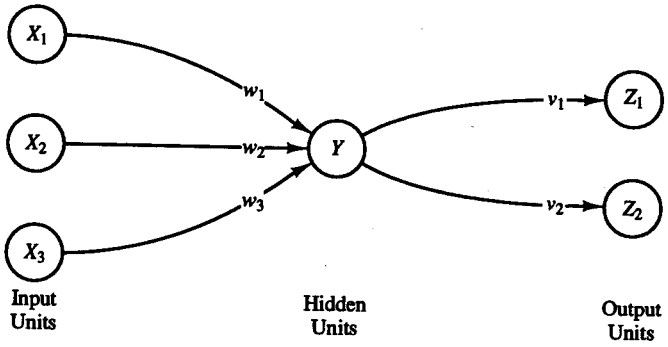

Figure 1.2 A very simple neural network.

of the net) and the method of training the net are discussed further in Section 1.4. A detailed consideration of these ideas for specific nets, together with simple examples of an application of each net, is the focus of the following chapters.

1.2.2 Biological Neural Networks

The extent to which a neural network models a particular biological neural system varies. For some researchers, this is a primary concern; for others, the ability of the net to perform useful tasks (such as approximate or represent a function) is more important than the biological plausibility of the net. Although our interest lies almost exclusively in the computational capabilities of neural networks, we shall present a brief discussion of some features of biological neurons that may help to clarify the most important characteristics of artificial neural networks. In addition to being the original inspiration for artificial nets, biological neural systems suggest features that have distinct computational advantages.

There is a close analogy between the structure of a biological neuron (i.e., a brain or nerve cell) and the processing element (or artificial neuron) presented in the rest of this book. In fact, the structure of an individual neuron varies much less from species to species than does the organization of the system of which the neuron is an element.

A biological neuron has three types of components that are of particular interest in understanding an artificial neuron: its *dendrites, soma,* and *axon.* The many dendrites receive signals from other neurons. The signals are electric impulses that are transmitted across a synaptic gap by means of a chemical process. The action of the chemical transmitter modifies the incoming signal (typically, by scaling the frequency of the signals that are received) in a manner similar to the action of the weights in an artificial neural network.

The soma, or cell body, sums the incoming signals. When sufficient input is received, the cell fires; that is, it transmits a signal over its axon to other cells. It is often supposed that a cell either fires or doesn't at any instant of time, so that transmitted signals can be treated as binary. However, the frequency of firing varies and can be viewed as a signal of either greater or lesser magnitude. This corresponds to looking at discrete time steps and summing all activity (signals received or signals sent) at a particular point in time.

The transmission of the signal from a particular neuron is accomplished by an action potential resulting from differential concentrations of ions on either side of the neuron's axon sheath (the brain's "white matter"). The ions most directly involved are potassium, sodium, and chloride.

A generic biological neuron is illustrated in Figure 1.3, together with axons from two other neurons (from which the illustrated neuron could receive signals) and dendrites for two other neurons (to which the original neuron would send signals). Several key features of the processing elements of artificial neural networks are suggested by the properties of biological neurons, viz., that:

1. The processing element receives many signals.
2. Signals may be modified by a weight at the receiving synapse.
3. The processing element sums the weighted inputs.
4. Under appropriate circumstances (sufficient input), the neuron transmits a single output.
5. The output from a particular neuron may go to many other neurons (the axon branches).

Other features of artificial neural networks that are suggested by biological neurons are:

6. Information processing is local (although other means of transmission, such as the action of hormones, may suggest means of overall process control).
7. Memory is distributed:
   a. Long-term memory resides in the neurons' synapses or weights.
   b. Short-term memory corresponds to the signals sent by the neurons.
8. A synapse's strength may be modified by experience.
9. Neurotransmitters for synapses may be excitatory or inhibitory.

Yet another important characteristic that artificial neural networks share with biological neural systems is *fault tolerance*. Biological neural systems are fault tolerant in two respects. First, we are able to recognize many input signals that are somewhat different from any signal we have seen before. An example of this is our ability to recognize a person in a picture we have not seen before or to recognize a person after a long period of time.

Second, we are able to tolerate damage to the neural system itself. Humans are born with as many as 100 billion neurons. Most of these are in the brain, and most are not replaced when they die [Johnson & Brown, 1988]. In spite of our continuous loss of neurons, we continue to learn. Even in cases of traumatic neural

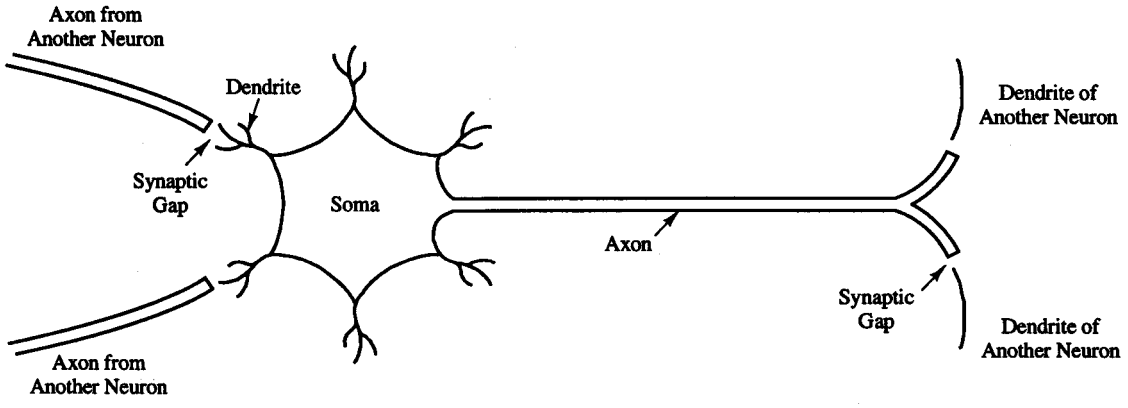

Figure 1.3 Biological neuron.

loss, other neurons can sometimes be trained to take over the functions of the damaged cells. In a similar manner, artificial neural networks can be designed to be insensitive to small damage to the network, and the network can be retrained in cases of significant damage (e.g., loss of data and some connections).

Even for uses of artificial neural networks that are not intended primarily to model biological neural systems, attempts to achieve biological plausibility may lead to improved computational features. One example is the use of a planar array of neurons, as is found in the neurons of the visual cortex, for Kohonen's self-organizing maps (see Chapter 4). The topological nature of these maps has computational advantages, even in applications where the structure of the output units is not itself significant.

Other researchers have found that computationally optimal groupings of artificial neurons correspond to biological bundles of neurons [Rogers & Kabrisky, 1989]. Separating the action of a backpropagation net into smaller pieces to make it more local (and therefore, perhaps more biologically plausible) also allows improvement in computational power (cf. Section 6.2.3) [D. Fausett, 1990].

1.3 WHERE ARE NEURAL NETS BEING USED?

The study of neural networks is an extremely interdisciplinary field, both in its development and in its application. A brief sampling of some of the areas in which neural networks are currently being applied suggests the breadth of their applicability. The examples range from commercial successes to areas of active research that show promise for the future.

1.3.1 Signal Processing

There are many applications of neural networks in the general area of signal processing. One of the first commercial applications was (and still is) to suppress noise on a telephone line. The neural net used for this purpose is a form of ADALINE. (We discuss ADALINES in Chapter 2.) The need for adaptive echo cancelers has become more pressing with the development of transcontinental satellite links for long-distance telephone circuits. The two-way round-trip time delay for the radio transmission is on the order of half a second. The switching involved in conventional echo suppression is very disruptive with path delays of this length. Even in the case of wire-based telephone transmission, the repeater amplifiers introduce echoes in the signal.

The adaptive noise cancellation idea is quite simple. At the end of a long-distance line, the incoming signal is applied to both the telephone system component (called the hybrid) and the adaptive filter (the ADALINE type of neural net). The difference between the output of the hybrid and the output of the ADALINE is the error, which is used to adjust the weights on the ADALINE. The ADALINE is trained to remove the noise (echo) from the hybrid's output signal. (See Widrow and Stearns, 1985, for a more detailed discussion.)

1.3.2 Control

The difficulties involved in backing up a trailer are obvious to anyone who has either attempted or watched a novice attempt this maneuver. However, a driver with experience accomplishes the feat with remarkable ease. As an example of the application of neural networks to control problems, consider the task of training a neural "truck backer-upper" to provide steering directions to a trailer truck attempting to back up to a loading dock [Nguyen & Widrow, 1989; Miller, Sutton, & Werbos, 1990]. Information is available describing the position of the cab of the truck, the position of the rear of the trailer, the (fixed) position of the loading dock, and the angles that the truck and the trailer make with the loading dock. The neural net is able to learn how to steer the truck in order for the trailer to reach the dock, starting with the truck and trailer in any initial configuration that allows enough clearance for a solution to be possible. To make the problem more challenging, the truck is allowed only to back up.

The neural net solution to this problem uses two modules. The first (called the *emulator*) learns to compute the new position of the truck, given its current position and the steering angle. The truck moves a fixed distance at each time step. This module learns the "feel" of how a trailer truck responds to various steering signals, in much the same way as a driver learns the behavior of such a rig. The emulator has several hidden units and is trained using backpropagation (which is the subject of Chapter 6).

The second module is the *controller*. After the emulator is trained, the controller learns to give the correct series of steering signals to the truck so that the trailer arrives at the dock with its back parallel to the dock. At each time step, the controller gives a steering signal and the emulator determines the new position of the truck and trailer. This process continues until either the trailer reaches the dock or the rig jackknifes. The error is then determined and the weights on the controller are adjusted.

As with a driver, performance improves with practice, and the neural controller learns to provide a series of steering signals that direct the truck and trailer to the dock, regardless of the starting position (as long as a solution is possible). Initially, the truck may be facing toward the dock, may be facing away from the dock, or may be at any angle in between. Similarly, the angle between the truck and the trailer may have an initial value short of that in a jack-knife situation. The training process for the controller is similar to the recurrent backpropagation described in Chapter 7.

1.3.3 Pattern Recognition

Many interesting problems fall into the general area of pattern recognition. One specific area in which many neural network applications have been developed is the automatic recognition of handwritten characters (digits or letters). The large

variation in sizes, positions, and styles of writing make this a difficult problem for traditional techniques. It is a good example, however, of the type of information processing that humans can perform relatively easily.

General-purpose multilayer neural nets, such as the backpropagation net (a multilayer net trained by backpropagation) described in Chapter 6, have been used for recognizing handwritten zip codes [Le Cun et al., 1990]. Even when an application is based on a standard training algorithm, it is quite common to customize the architecture to improve the performance of the application. This backpropagation net has several hidden layers, but the pattern of connections from one layer to the next is quite localized.

An alternative approach to the problem of recognizing handwritten characters is the "neocognitron" described in Chapter 7. This net has several layers, each with a highly structured pattern of connections from the previous layer and to the subsequent layer. However, its training is a layer-by-layer process, specialized for just such an application.

1.3.4 Medicine

One of many examples of the application of neural networks to medicine was developed by Anderson et al. in the mid-1980s [Anderson, 1986; Anderson, Golden, and Murphy, 1986]. It has been called the "Instant Physician" [Hecht-Nielsen, 1990]. The idea behind this application is to train an autoassociative memory neural network (the "Brain-State-in-a-Box," described in Section 3.4.2) to store a large number of medical records, each of which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

The net performs surprisingly well, given its simple structure. When a particular set of symptoms occurs frequently in the training set, together with a unique diagnosis and treatment, the net will usually give the same diagnosis and treatment. In cases where there are ambiguities in the training data, the net will give the most common diagnosis and treatment. In novel situations, the net will prescribe a treatment corresponding to the symptom(s) it has seen before, regardless of the other symptoms that are present.

1.3.5 Speech Production

Learning to read English text aloud is a difficult task, because the correct phonetic pronunciation of a letter depends on the context in which the letter appears. A traditional approach to the problem would typically involve constructing a set of rules for the standard pronunciation of various groups of letters, together with a look-up table for the exceptions.

One of the most widely known examples of a neural network approach to

the problem of speech production is NETtalk [Sejnowski and Rosenberg, 1986], a multilayer neural net (i.e., a net with hidden units) similar to those described in Chapter 6. In contrast to the need to construct rules and look-up tables for the exceptions, NETtalk's only requirement is a set of examples of the written input, together with the correct pronunciation for it. The written input includes both the letter that is currently being spoken and three letters before and after it (to provide a context). Additional symbols are used to indicate the end of a word or punctuation. The net is trained using the 1,000 most common English words. After training, the net can read new words with very few errors; the errors that it does make are slight mispronunciations, and the intelligibility of the speech is quite good.

It is interesting that there are several fairly distinct stages to the response of the net as training progresses. The net learns quite quickly to distinguish vowels from consonants; however, it uses the same vowel for all vowels and the same consonant for all consonants at this first stage. The result is a babbling sound. The second stage of learning corresponds to the net recognizing the boundaries between words; this produces a pseudoword type of response. After as few as 10 passes through the training data, the text is intelligible. Thus, the response of the net as training progresses is similar to the development of speech in small children.

1.3.6 Speech Recognition

Progress is being made in the difficult area of speaker-independent recognition of speech. A number of useful systems now have a limited vocabulary or grammar or require retraining for different speakers. Several types of neural networks have been used for speech recognition, including multilayer nets (see Chapter 6) or multilayer nets with recurrent connections (see Section 7.2). Lippmann (1989) summarizes the characteristics of many of these nets.

One net that is of particular interest, both because of its level of development toward a practical system and because of its design, was developed by Kohonen using the self-organizing map (Chapter 4). He calls his net a "phonetic typewriter." The output units for a self-organizing map are arranged in a two-dimensional array (rectangular or hexagonal). The input to the net is based on short segments (a few milliseconds long) of the speech waveform. As the net groups similar inputs, the clusters that are formed are positioned so that different examples of the same phoneme occur on output units that are close together in the output array.

After the speech input signals are mapped to the phoneme regions (which has been done without telling the net what a phoneme is), the output units can be connected to the appropriate typewriter key to construct the phonetic typewriter. Because the correspondence between phonemes and written letters is very regular in Finnish (for which the net was developed), the spelling is often correct. See Kohonen (1988) for a more extensive description.

1.3.7 Business

Neural networks are being applied in a number of business settings [Harston, 1990]. We mention only one of many examples here, the mortgage assessment work by Nestor, Inc. [Collins, Ghosh, & Scofield, 1988a, 1988b].

Although it may be thought that the rules which form the basis for mortgage underwriting are well understood, it is difficult to specify completely the process by which experts make decisions in marginal cases. In addition, there is a large financial reward for even a small reduction in the number of mortgages that become delinquent. The basic idea behind the neural network approach to mortgage risk assessment is to use past experience to train the net to provide more consistent and reliable evaluation of mortgage applications.

Using data from several experienced mortgage evaluators, neural nets were trained to screen mortgage applicants for mortgage origination underwriting and mortgage insurance underwriting. The purpose in each of these is to determine whether the applicant should be given a loan. The decisions in the second kind of underwriting are more difficult, because only those applicants assessed as higher risks are processed for mortgage insurance. The training input includes information on the applicant's years of employment, number of dependents, current income, etc., as well as features related to the mortgage itself, such as the loan-to-value ratio, and characteristics of the property, such as its appraised value. The target output from the net is an "accept" or "reject" response.

In both kinds of underwriting, the neural networks achieved a high level of agreement with the human experts. When disagreement did occur, the case was often a marginal one where the experts would also disagree. Using an independent measure of the quality of the mortgages certified, the neural network consistently made better judgments than the experts. In effect, the net learned to form a consensus from the experience of all of the experts whose actions had formed the basis for its training.

A second neural net was trained to evaluate the risk of default on a loan, based on data from a data base consisting of 111,080 applications, 109,072 of which had no history of delinquency. A total of 4,000 training samples were selected from the data base. Although delinquency can result from many causes that are not reflected in the information available on a loan application, the predictions the net was able to make produced a 12% reduction in delinquencies.

1.4 HOW ARE NEURAL NETWORKS USED?

Let us now consider some of the fundamental features of how neural networks operate. Detailed discussions of these ideas for a number of specific nets are presented in the remaining chapters. The building blocks of our examination here are the network architectures and the methods of setting the weights (training).

We also illustrate several typical activation functions and conclude the section with a summary of the notation we shall use throughout the rest of the text.

1.4.1 Typical Architectures

Often, it is convenient to visualize neurons as arranged in layers. Typically, neurons in the same layer behave in the same manner. Key factors in determining the behavior of a neuron are its activation function and the pattern of weighted connections over which it sends and receives signals. Within each layer, neurons usually have the same activation function and the same pattern of connections to other neurons. To be more specific, in many neural networks, the neurons within a layer are either fully interconnected or not interconnected at all. If any neuron in a layer (for instance, the layer of hidden units) is connected to a neuron in another layer (say, the output layer), then each hidden unit is connected to every output neuron.

The arrangement of neurons into layers and the connection patterns within and between layers is called the *net architecture*. Many neural nets have an input layer in which the activation of each unit is equal to an external input signal. The net illustrated in Figure 1.2 consists of input units, output units, and one hidden unit (a unit that is neither an input unit nor an output unit).

Neural nets are often classified as single layer or multilayer. In determining the number of layers, the input units are not counted as a layer, because they perform no computation. Equivalently, the number of layers in the net can be defined to be the number of layers of weighted interconnect links between the slabs of neurons. This view is motivated by the fact that the weights in a net contain extremely important information. The net shown in Figure 1.2 has two layers of weights.

The single-layer and multilayer nets illustrated in Figures 1.4 and 1.5 are examples of *feedforward* nets—nets in which the signals flow from the input units to the output units, in a forward direction. The fully interconnected competitive net in Figure 1.6 is an example of a *recurrent* net, in which there are closed-loop signal paths from a unit back to itself.

Single-Layer Net

A single-layer net has one layer of connection weights. Often, the units can be distinguished as input units, which receive signals from the outside world, and output units, from which the response of the net can be read. In the typical single-layer net shown in Figure 1.4, the input units are fully connected to output units but are not connected to other input units, and the output units are not connected to other output units. By contrast, the Hopfield net architecture, shown in Figure 3.7, is an example of a single-layer net in which all units function as both input and output units.

For pattern classification, each output unit corresponds to a particular cat-

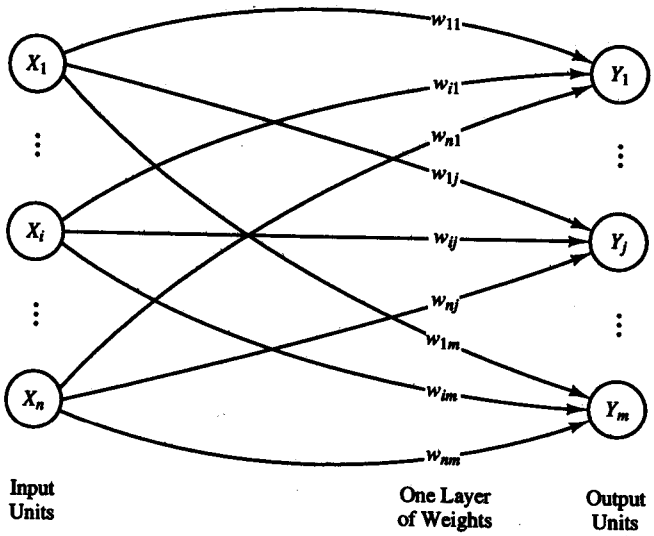

Figure 1.4 A single-layer neural net.

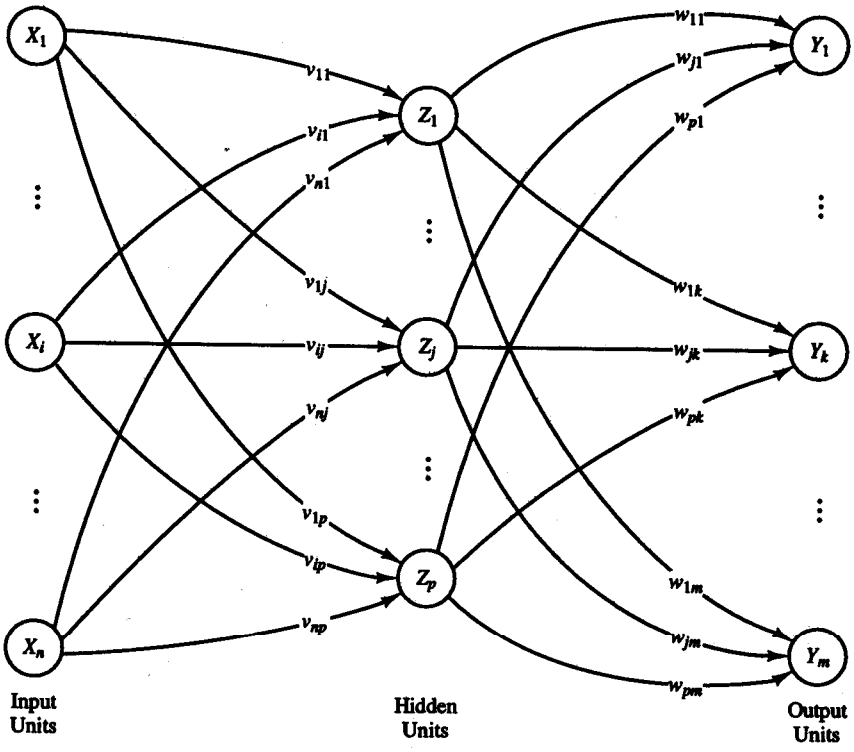

Figure 1.5 A multilayer neural net.

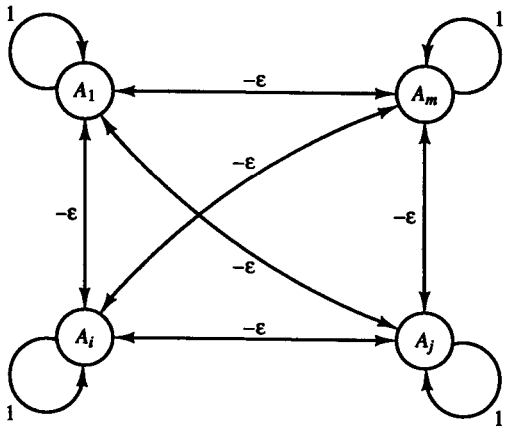

Figure 1.6 Competitive layer.

egory to which an input vector may or may not belong. Note that for a single-layer net, the weights for one output unit do not influence the weights for other output units. For pattern association, the same architecture can be used, but now the overall pattern of output signals gives the response pattern associated with the input signal that caused it to be produced. These two examples illustrate the fact that the same type of net can be used for different problems, depending on the interpretation of the response of the net.

On the other hand, more complicated mapping problems may require a multilayer network. The characteristics of the problems for which a single-layer net is satisfactory are considered in Chapters 2 and 3. The problems that require multilayer nets may still represent a classification or association of patterns; the type of problem influences the choice of architecture, but does not uniquely determine it.

Multilayer net

A multilayer net is a net with one or more layers (or levels) of nodes (the so-called hidden units) between the input units and the output units. Typically, there is a layer of weights between two adjacent levels of units (input, hidden, or output). Multilayer nets can solve more complicated problems than can single-layer nets, but training may be more difficult. However, in some cases, training may be more successful, because it is possible to solve a problem that a single-layer net cannot be trained to perform correctly at all.

Competitive layer

A competitive layer forms a part of a large number of neural networks. Several examples of these nets are discussed in Chapters 4 and 5. Typically, the interconnections between neurons in the competitive layer are not shown in the architecture diagrams for such nets. An example of the architecture for a competitive

layer is given in Figure 1.6; the competitive interconnections have weights of $-\epsilon$. The operation of a winner-take-all competition, MAXNET [Lippman, 1987], is described in Section 4.1.1.

1.4.2 Setting the Weights

In addition to the architecture, the method of setting the values of the weights (training) is an important distinguishing characteristic of different neural nets. For convenience, we shall distinguish two types of training—supervised and unsupervised—for a neural network; in addition, there are nets whose weights are fixed without an iterative training process.

Many of the tasks that neural nets can be trained to perform fall into the areas of mapping, clustering, and constrained optimization. Pattern classification and pattern association may be considered special forms of the more general problem of mapping input vectors or patterns to the specified output vectors or patterns.

There is some ambiguity in the labeling of training methods as supervised or unsupervised, and some authors find a third category, self-supervised training, useful. However, in general, there is a useful correspondence between the type of training that is appropriate and the type of problem we wish to solve. We summarize here the basic characteristics of supervised and unsupervised training and the types of problems for which each, as well as the fixed-weight nets, is typically used.

Supervised training

In perhaps the most typical neural net setting, training is accomplished by presenting a sequence of training vectors, or patterns, each with an associated target output vector. The weights are then adjusted according to a learning algorithm. This process is known as *supervised training*.

Some of the simplest (and historically earliest) neural nets are designed to perform pattern classification, i.e., to classify an input vector as either belonging or not belonging to a given category. In this type of neural net, the output is a bivalent element, say, either 1 (if the input vector belongs to the category) or -1 (if it does not belong). In the next chapter, we consider several simple single-layer nets that were designed or typically used for pattern classification. These nets are trained using a supervised algorithm. The characteristics of a classification problem that determines whether a single-layer net is adequate are considered in Chapter 2 also. For more difficult classification problems, a multilayer net, such as that trained by backpropagation (presented in Chapter 6) may be better.

Pattern association is another special form of a mapping problem, one in which the desired output is not just a "yes" or "no," but rather a pattern. A neural net that is trained to associate a set of input vectors with a corresponding

set of output vectors is called an *associative memory*. If the desired output vector is the same as the input vector, the net is an *autoassociative memory*; if the output target vector is different from the input vector, the net is a *heteroassociative memory*. After training, an associative memory can recall a stored pattern when it is given an input vector that is sufficiently similar to a vector it has learned. Associative memory neural nets, both feedforward and recurrent, are discussed in Chapter 3.

Multilayer neural nets can be trained to perform a nonlinear mapping from an n-dimensional space of input vectors (n-tuples) to an m-dimensional output space—i.e., the output vectors are m-tuples.

The single-layer nets in Chapter 2 (pattern classification nets) and Chapter 3 (pattern association nets) use supervised training (the Hebb rule or the delta rule). Backpropagation (the generalized delta rule) is used to train the multilayer nets in Chapter 6. Other forms of supervised learning are used for some of the nets in Chapter 4 (learning vector quantization and counterpropagation) and Chapter 7. Each learning algorithm will be described in detail, along with a description of the net for which it is used.

Unsupervised training

Self-organizing neural nets group similar input vectors together without the use of training data to specify what a typical member of each group looks like or to which group each vector belongs. A sequence of input vectors is provided, but no target vectors are specified. The net modifies the weights so that the most similar input vectors are assigned to the same output (or cluster) unit. The neural net will produce an exemplar (representative) vector for each cluster formed. Self-organizing nets are described in Chapters 4 (Kohonen self-organizing maps) and Chapter 5 (adaptive resonance theory).

Unsupervised learning is also used for other tasks, in addition to clustering. Examples are included in Chapter 7.

Fixed-weight nets

Still other types of neural nets can solve constrained optimization problems. Such nets may work well for problems that can cause difficulty for traditional techniques, such as problems with conflicting constraints (i.e., not all constraints can be satisfied simultaneously). Often, in such cases, a nearly optimal solution (which the net can find) is satisfactory. When these nets are designed, the weights are set to represent the constraints and the quantity to be maximized or minimized. The Boltzmann machine (without learning) and the continuous Hopfield net (Chapter 7) can be used for constrained optimization problems.

Fixed weights are also used in contrast-enhancing nets (see Section 4.1).

1.4.3 Common Activation Functions

As mentioned before, the basic operation of an artificial neuron involves summing its weighted input signal and applying an output, or activation, function. For the input units, this function is the identity function (see Figure 1.7). Typically, the same activation function is used for all neurons in any particular layer of a neural net, although this is not required. In most cases, a nonlinear activation function is used. In order to achieve the advantages of multilayer nets, compared with the limited capabilities of single-layer nets, nonlinear functions are required (since the results of feeding a signal through two or more layers of linear processing elements—i.e., elements with linear activation functions—are no different from what can be obtained using a single layer).

(*i*) Identity function:

$$f(x) = x \qquad \text{for all } x.$$

Single-layer nets often use a step function to convert the net input, which is a continuously valued variable, to an output unit that is a binary (1 or 0) or bipolar (1 or -1) signal (see Figure 1.8). The use of a *threshold* in this regard is discussed in Section 2.1.2. The binary step function is also known as the threshold function or Heaviside function.

(*ii*) Binary step function (with threshold θ):

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

Sigmoid functions (S-shaped curves) are useful activation functions. The logistic function and the hyperbolic tangent functions are the most common. They are especially advantageous for use in neural nets trained by backpropagation, because the simple relationship between the value of the function at a point and the value of the derivative at that point reduces the computational burden during training.

The logistic function, a sigmoid function with range from 0 to 1, is often

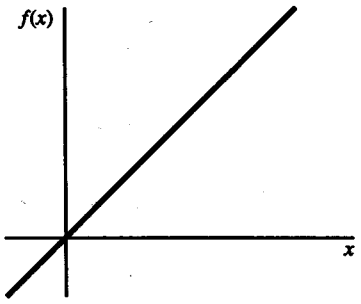

Figure 1.7 Identity function.

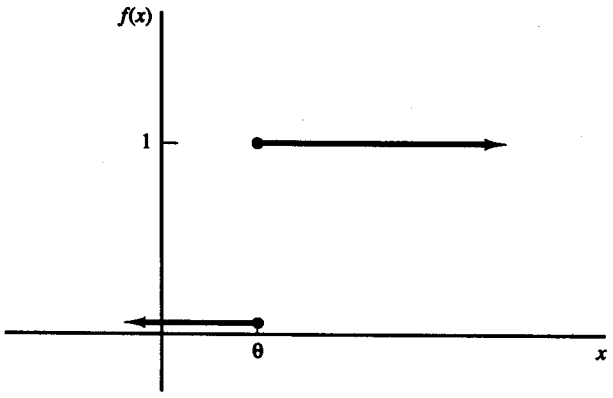

Figure 1.8 Binary step function.

used as the activation function for neural nets in which the desired output values either are binary or are in the interval between 0 and 1. To emphasize the range of the function, we will call it the *binary sigmoid*; it is also called the *logistic sigmoid*. This function is illustrated in Figure 1.9 for two values of the *steepness parameter* σ.

(*iii*) Binary sigmoid:

$$f(x) = \frac{1}{1 + \exp(-\sigma x)}.$$

$$f'(x) = \sigma f(x)\,[1 - f(x)].$$

As is shown in Section 6.2.3, the logistic sigmoid function can be scaled to have any range of values that is appropriate for a given problem. The most common range is from -1 to 1; we call this sigmoid the *bipolar sigmoid*. It is illustrated in Figure 1.10 for $\sigma = 1$.

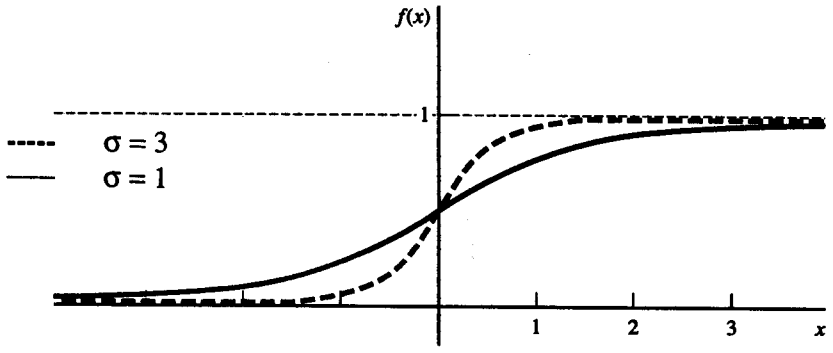

Figure 1.9 Binary sigmoid. Steepness parameters $\sigma = 1$ and $\sigma = 3$.

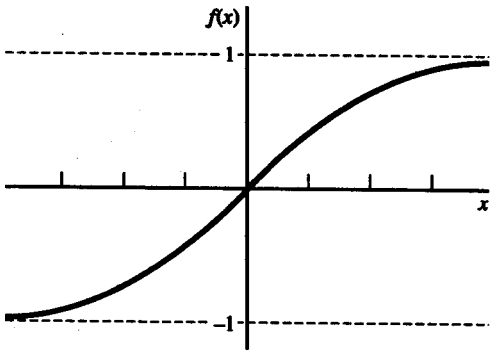

Figure 1.10 Bipolar sigmoid.

(*iv*) Bipolar sigmoid:

$$g(x) = 2f(x) - 1 = \frac{2}{1 + \exp(-\sigma x)} - 1$$

$$= \frac{1 - \exp(-\sigma x)}{1 + \exp(-\sigma x)}.$$

$$g'(x) = \frac{\sigma}{2}[1 + g(x)][1 - g(x)].$$

The bipolar sigmoid is closely related to the hyperbolic tangent function, which is also often used as the activation function when the desired range of output values is between -1 and 1. We illustrate the correspondence between the two for $\sigma = 1$. We have

$$g(x) = \frac{1 - \exp(-x)}{1 + \exp(-x)}.$$

The hyperbolic tangent is

$$h(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$= \frac{1 - \exp(-2x)}{1 + \exp(-2x)}.$$

The derivative of the hyperbolic tangent is

$$h'(x) = [1 + h(x)][1 - h(x)].$$

For binary data (rather than continuously valued data in the range from 0 to 1), it is usually preferable to convert to bipolar form and use the bipolar sigmoid or hyperbolic tangent. A more extensive discussion of the choice of activation functions and different forms of sigmoid functions is given in Section 6.2.2.

1.4.4 Summary of Notation

The following notation will be used throughout the discussions of specific neural nets, unless indicated otherwise for a particular net (appropriate values for the parameter depend on the particular neural net model being used and will be discussed further for each model):

x_i, y_j Activations of units X_i, Y_j, respectively:
For input units X_i,

$$x_i = \text{input signal};$$

for other units Y_j,

$$y_j = f(y\_in_j).$$

w_{ij} Weight on connection from unit X_i to unit Y_j:
Beware: Some authors use the opposite convention, with w_{ij} denoting the weight from unit Y_j to unit X_i.

b_j Bias on unit Y_j:
A bias acts like a weight on a connection from a unit with a constant activation of 1 (see Figure 1.11).

y_in_j Net input to unit Y_j:

$$y\_in_j = b_j + \sum_i x_i w_{ij}$$

W Weight matrix:

$$W = \{w_{ij}\}.$$

$\mathbf{w}_{\cdot j}$ Vector of weights:

$$\mathbf{w}_{\cdot j} = (w_{1j}, w_{2j}, \ldots, w_{nj})^T.$$

This is the jth column of the weight matrix.

$\| \mathbf{x} \|$ Norm or magnitude of vector $\mathbf{x}$.

θ_j Threshold for activation of neuron Y_j:
A step activation function sets the activation of a neuron to 1 whenever its net input is greater than the specified threshold value θ_j; otherwise its activation is 0 (see Figure 1.8).

$\mathbf{s}$ Training input vector:

$$\mathbf{s} = (s_1, \ldots, s_i, \ldots, s_n).$$

$\mathbf{t}$ Training (or target) output vector:

$$\mathbf{t} = (t_1, \ldots, t_j, \ldots, t_m).$$

$\mathbf{x}$ Input vector (for the net to classify or respond to):

$$\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_n).$$

Δw_{ij} Change in w_{ij}:

$$\Delta w_{ij} = [w_{ij}\,(\text{new}) - w_{ij}\,(\text{old})].$$

α Learning rate:

The learning rate is used to control the amount of weight adjustment at each step of training.

Matrix multiplication method for calculating net input

If the connection weights for a neural net are stored in a matrix $\mathbf{W} = (w_{ij})$, the net input to unit Y_j (with no bias on unit j) is simply the dot product of the vectors $\mathbf{x} = (x_1, \ldots, x_i, \ldots, x_n)$ and $\mathbf{w}_{\cdot j}$ (the jth column of the weight matrix):

$$\begin{aligned} y\_in_j &= \mathbf{x}\cdot\mathbf{w}_{\cdot j} \\ &= \sum_{i=1}^{n} x_i w_{ij}\,. \end{aligned}$$

Bias

A bias can be included by adding a component $x_0 = 1$ to the vector $\mathbf{x}$, i.e., $\mathbf{x} = (1, x_1, \ldots, x_i, \ldots, x_n)$. The bias is treated exactly like any other weight, i.e., $w_{0j} = b_j$. The net input to unit Y_j is given by

$$\begin{aligned} y\_in_j &= \mathbf{x}\cdot\mathbf{w}_{\cdot j} \\ &= \sum_{i=0}^{n} x_i w_{ij} \\ &= w_{0j} + \sum_{i=1}^{n} x_i w_{ij} \\ &= b_j + \sum_{i=1}^{n} x_i w_{ij}\,. \end{aligned}$$

The relation between a bias and a threshold is considered in Section 2.1.2.

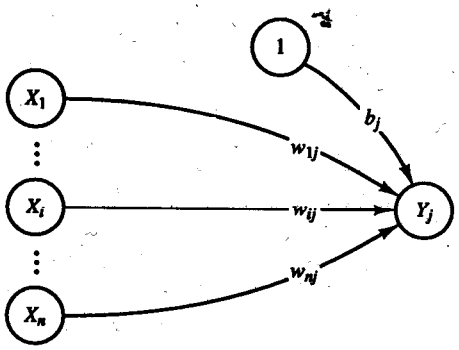

Figure 1.11 Neuron with a bias.

1.5 WHO IS DEVELOPING NEURAL NETWORKS?

This section presents a very brief summary of the history of neural networks, in terms of the development of architectures and algorithms that are widely used today. Results of a primarily biological nature are not included, due to space constraints. They have, however, served as the inspiration for a number of networks that are applicable to problems beyond the original ones studied. The history of neural networks shows the interplay among biological experimentation, modeling, and computer simulation/hardware implementation. Thus, the field is strongly interdisciplinary.

1.5.1 The 1940s: The Beginning of Neural Nets

McCulloch-Pitts neurons

Warren McCulloch and Walter Pitts designed what are generally regarded as the first neural networks [McCulloch & Pitts, 1943]. These researchers recognized that combining many simple neurons into neural systems was the source of increased computational power. The weights on a McCulloch-Pitts neuron are set so that the neuron performs a particular simple logic function, with different neurons performing different functions. The neurons can be arranged into a net to produce any output that can be represented as a combination of logic functions. The flow of information through the net assumes a unit time step for a signal to travel from one neuron to the next. This time delay allows the net to model some physiological processes, such as the perception of hot and cold.

The idea of a threshold such that if the net input to a neuron is greater than the threshold then the unit fires is one feature of a McCulloch-Pitts neuron that is used in many artificial neurons today. However, McCulloch-Pitts neurons are used most widely as logic circuits [Anderson & Rosenfeld, 1988].

McCulloch and Pitts subsequent work [Pitts & McCulloch, 1947] addressed issues that are still important research areas today, such as translation and rotation invariant pattern recognition.

Hebb learning

Donald Hebb, a psychologist at McGill University, designed the first learning law for artificial neural networks [Hebb, 1949]. His premise was that if two neurons were active simultaneously, then the strength of the connection between them should be increased. Refinements were subsequently made to this rather general statement to allow computer simulations [Rochester, Holland, Haibt & Duda, 1956]. The idea is closely related to the correlation matrix learning developed by Kohonen (1972) and Anderson (1972) among others. An expanded form of Hebb learning [McClelland & Rumelhart, 1988] in which units that are simultaneously off also reinforce the weight on the connection between them will be presented in Chapters 2 and 3.

1.5.2 The 1950s and 1960s: The First Golden Age of Neural Networks

Although today neural networks are often viewed as an alternative to (or complement of) traditional computing, it is interesting to note that John von Neumann, the "father of modern computing," was keenly interested in modeling the brain [von Neumann, 1958]. Johnson and Brown (1988) and Anderson and Rosenfeld (1988) discuss the interaction between von Neumann and early neural network researchers such as Warren McCulloch, and present further indication of von Neumann's views of the directions in which computers would develop.

Perceptrons

Together with several other researchers [Block, 1962; Minsky & Papert, 1988 (originally published 1969)], Frank Rosenblatt (1958, 1959, 1962) introduced and developed a large class of artificial neural networks called *perceptrons*. The most typical perceptron consisted of an input layer (the retina) connected by paths with fixed weights to associator neurons; the weights on the connection paths were adjustable. The perceptron learning rule uses an iterative weight adjustment that is more powerful than the Hebb rule. Perceptron learning can be proved to converge to the correct weights if there are weights that will solve the problem at hand (i.e., allow the net to reproduce correctly all of the training input and target output pairs). Rosenblatt's 1962 work describes many types of perceptrons. Like the neurons developed by McCulloch and Pitts and by Hebb, perceptrons use a threshold output function.

The early successes with perceptrons led to enthusiastic claims. However, the mathematical proof of the convergence of iterative learning under suitable assumptions was followed by a demonstration of the limitations regarding what the perceptron type of net can learn [Minsky & Papert, 1969].

ADALINE

Bernard Widrow and his student, Marcian (Ted) Hoff [Widrow & Hoff, 1960], developed a learning rule (which usually either bears their names, or is designated the least mean squares or delta rule) that is closely related to the perceptron learning rule. The perceptron rule adjusts the connection weights to a unit whenever the response of the unit is incorrect. (The response indicates a classification of the input pattern.) The delta rule adjusts the weights to reduce the difference between the net input to the output unit and the desired output. This results in the smallest mean squared error. The similarity of models developed in psychology by Rosenblatt to those developed in electrical engineering by Widrow and Hoff is evidence of the interdisciplinary nature of neural networks. The difference in learning rules, although slight, leads to an improved ability of the net to generalize (i.e., respond to input that is similar, but not identical, to that on which it was trained). The Widrow-Hoff learning rule for a single-layer network is a precursor of the backpropagation rule for multilayer nets.

Work by Widrow and his students is sometimes reported as neural network research, sometimes as adaptive linear systems. The name *ADALINE*, interpreted as either ADAptive LINear NEuron or ADAptive LINEar system, is often given to these nets. There have been many interesting applications of ADALINES, from neural networks for adaptive antenna systems [Widrow, Mantey, Griffiths, & Goode, 1967] to rotation-invariant pattern recognition to a variety of control problems, such as broom balancing and backing up a truck [Widrow, 1987; Tolat & Widrow, 1988; Nguyen & Widrow, 1989]. MADALINES are multilayer extensions of ADALINES [Widrow & Hoff, 1960; Widrow & Lehr, 1990].

1.5.3 The 1970s: The Quiet Years

In spite of Minsky and Papert's demonstration of the limitations of perceptrons (i.e., single-layer nets), research on neural networks continued. Many of the current leaders in the field began to publish their work during the 1970s. (Widrow, of course, had started somewhat earlier and is still active.)

Kohonen

The early work of Teuvo Kohonen (1972), of Helsinki University of Technology, dealt with associative memory neural nets. His more recent work [Kohonen, 1982] has been the development of self-organizing feature maps that use a topological structure for the cluster units. These nets have been applied to speech recognition (for Finnish and Japanese words) [Kohonen, Torkkola, Shozakai, Kangas, & Venta, 1987; Kohonen, 1988], the solution of the "Traveling Salesman Problem" [Angeniol, Vaubois, & Le Texier, 1988], and musical composition [Kohonen, 1989b].

Anderson

James Anderson, of Brown University, also started his research in neural networks with associative memory nets [Anderson, 1968, 1972]. He developed these ideas into his "Brain-State-in-a-Box" [Anderson, Silverstein, Ritz, & Jones, 1977], which truncates the linear output of earlier models to prevent the output from becoming too large as the net iterates to find a stable solution (or memory). Among the areas of application for these nets are medical diagnosis and learning multiplication tables. Anderson and Rosenfeld (1988) and Anderson, Pellionisz, and Rosenfeld (1990) are collections of fundamental papers on neural network research. The introductions to each are especially useful.

Grossberg

Stephen Grossberg, together with his many colleagues and coauthors, has had an extremely prolific and productive career. Klimasauskas (1989) lists 146 publications by Grossberg from 1967 to 1988. His work, which is very mathematical and very biological, is widely known [Grossberg, 1976, 1980, 1982, 1987, 1988]. Grossberg is director of the Center for Adaptive Systems at Boston University.

Carpenter

Together with Stephen Grossberg, Gail Carpenter has developed a theory of self-organizing neural networks called *adaptive resonance theory* [Carpenter & Grossberg, 1985, 1987a, 1987b, 1990]. Adaptive resonance theory nets for binary input patterns (ART1) and for continuously valued inputs (ART2) will be examined in Chapter 5.

1.5.4 The 1980s: Renewed Enthusiasm

Backpropagation

Two of the reasons for the "quiet years" of the 1970s were the failure of single-layer perceptrons to be able to solve such simple problems (mappings) as the XOR function and the lack of a general method of training a multilayer net. A method for propagating information about errors at the output units back to the hidden units had been discovered in the previous decade [Werbos, 1974], but had not gained wide publicity. This method was also discovered independently by David Parker (1985) and by LeCun (1986) before it became widely known. It is very similar to yet an earlier algorithm in optimal control theory [Bryson & Ho, 1969]. Parker's work came to the attention of the Parallel Distributed Processing Group led by psychologists David Rumelhart, of the University of California at San Diego, and James McClelland, of Carnegie-Mellon University, who refined and publicized it [Rumelhart, Hinton, & Williams, 1986a, 1986b; McClelland & Rumelhart, 1988].

Hopfield nets

Another key player in the increased visibility of and respect for neural nets is prominent physicist John Hopfield, of the California Institute of Technology. Together with David Tank, a researcher at AT&T, Hopfield has developed a number of neural networks based on fixed weights and adaptive activations [Hopfield, 1982, 1984; Hopfield & Tank, 1985, 1986; Tank & Hopfield, 1987]. These nets can serve as associative memory nets and can be used to solve constraint satisfaction problems such as the "Traveling Salesman Problem." An article in *Scientific American* [Tank & Hopfield, 1987] helped to draw popular attention to neural nets, as did the message of a Nobel prize–winning physicist that, in order to make machines that can do what humans do, we need to study human cognition.

Neocognitron

Kunihiko Fukushima and his colleagues at NHK Laboratories in Tokyo have developed a series of specialized neural nets for character recognition. One example of such a net, called a *neocognitron,* is described in Chapter 7. An earlier self-organizing network, called the cognitron [Fukushima, 1975], failed to recognize position- or rotation-distorted characters. This deficiency was corrected in the neocognitron [Fukushima, 1988; Fukushima, Miyake, & Ito, 1983].

Boltzmann machine

A number of researchers have been involved in the development of nondeterministic neural nets, that is, nets in which weights or activations are changed on the basis of a probability density function [Kirkpatrick, Gelatt, & Vecchi, 1983; Geman & Geman, 1984; Ackley, Hinton, & Sejnowski, 1985; Szu & Hartley, 1987]. These nets incorporate such classical ideas as simulated annealing and Bayesian decision theory.

Hardware implementation

Another reason for renewed interest in neural networks (in addition to solving the problem of how to train a multilayer net) is improved computational capabilities. Optical neural nets [Farhat, Psaltis, Prata, & Paek, 1985] and VLSI implementations [Sivilatti, Mahowald, & Mead, 1987] are being developed.

Carver Mead, of California Institute of Technology, who also studies motion detection, is the coinventor of software to design microchips. He is also cofounder of Synaptics, Inc., a leader in the study of neural circuitry.

Nobel laureate Leon Cooper, of Brown University, introduced one of the first multilayer nets, the *reduced coulomb energy network*. Cooper is chairman of Nestor, the first public neural network company [Johnson & Brown, 1988], and the holder of several patents for information-processing systems [Klimasauskas, 1989].

Robert Hecht-Nielsen and Todd Gutschow developed several digital neurocomputers at TRW, Inc., during 1983–85. Funding was provided by the Defense Advanced Research Projects Agency (DARPA) [Hecht-Nielsen, 1990]. DARPA (1988) is a valuable summary of the state of the art in artificial neural networks (especially with regard to successful applications). To quote from the preface to his book, *Neurocomputing*, Hecht-Nielsen is "an industrialist, an adjunct academic, and a philanthropist without financial portfolio" [Hecht-Nielsen, 1990]. The founder of HNC, Inc., he is also a professor at the University of California, San Diego, and the developer of the counterpropagation network.

1.6 WHEN NEURAL NETS BEGAN: THE McCULLOCH-PITTS NEURON

The McCulloch-Pitts neuron is perhaps the earliest artificial neuron [McCulloch & Pitts, 1943]. It displays several important features found in many neural networks. The requirements for McCulloch-Pitts neurons may be summarized as follows:

1. The activation of a McCulloch-Pitts neuron is binary. That is, at any time step, the neuron either fires (has an activation of 1) or does not fire (has an activation of 0).
2. McCulloch-Pitts neurons are connected by directed, weighted paths.

3. A connection path is excitatory if the weight on the path is positive; otherwise it is inhibitory. All excitatory connections into a particular neuron have the same weights.
4. Each neuron has a fixed threshold such that if the net input to the neuron is greater than the threshold, the neuron fires.
5. The threshold is set so that inhibition is absolute. That is, any nonzero inhibitory input will prevent the neuron from firing.
6. It takes one time step for a signal to pass over one connection link.

The simple example of a McCulloch-Pitts neuron shown in Figure 1.12 illustrates several of these requirements. The connection from X_1 to Y is excitatory, as is the connection from X_2 to Y. These excitatory connections have the same (positive) weight because they are going into the same unit.

The threshold for unit Y is 4; for the values of the excitatory and inhibitory weights shown, this is the only integer value of the threshold that will allow Y to fire sometimes, but will prevent it from firing if it receives a nonzero signal over the inhibitory connection.

It takes one time step for the signals to pass from the X units to Y; the activation of Y at time t is determined by the activations of X_1, X_2, and X_3 at the previous time, $t - 1$. The use of discrete time steps enables a network of McCulloch-Pitts neurons to model physiological phenomena in which there is a time delay; such an example is given in Section 1.6.3.

1.6.1 Architecture

In general, a McCulloch-Pitts neuron Y may receive signals from any number of other neurons. Each connection path is either excitatory, with weight $w > 0$, or inhibitory, with weight $-p$ ($p > 0$). For convenience, in Figure 1.13, we assume there are n units, $X_1, \ldots, X_n$, which send excitatory signals to unit Y, and m units, $X_{n+1}, \ldots, X_{n+m}$, which send inhibitory signals. The activation function for unit Y is

$$f(y\_in) = \begin{cases} 1 & \text{if } y\_in \geq \theta \\ 0 & \text{if } y\_in < \theta \end{cases}$$

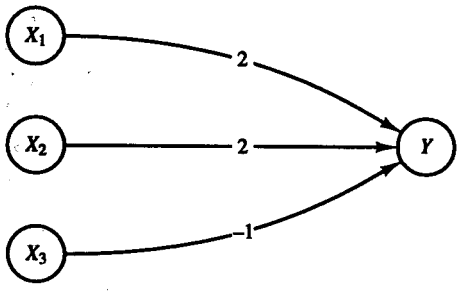

Figure 1.12 A simple McCulloch-Pitts neuron Y.

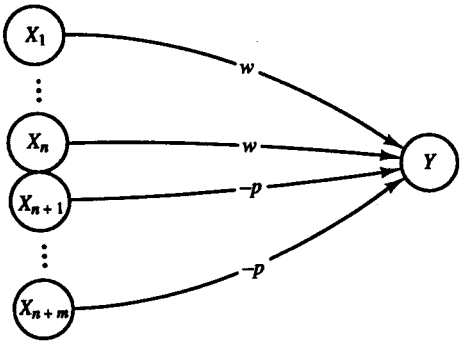

Figure 1.13 Architecture of a McCulloch-Pitts neuron Y.

where y_in is the total input signal received and θ is the threshold. The condition that inhibition is absolute requires that θ for the activation function satisfy the inequality

$$\theta > nw - p.$$

Y will fire if it receives k or more excitatory inputs and no inhibitory inputs, where

$$kw \geq \theta > (k - 1)w.$$

Although all excitatory weights coming into any particular unit must be the same, the weights coming into one unit, say, Y_1, do not have to be the same as the weights coming into another unit, say Y_2.

1.6.2 Algorithm

The weights for a McCulloch-Pitts neuron are set, together with the threshold for the neuron's activation function, so that the neuron will perform a simple logic function. Since analysis, rather than a training algorithm, is used to determine the values of the weights and threshold, several examples of simple McCulloch-Pitts neurons are presented in this section. Using these simple neurons as building blocks, we can model any function or phenomenon that can be represented as a logic function. In Section 1.6.3, an example is given of how several of these simple neurons can be combined to model an interesting physiological phenomenon.

Simple networks of McCulloch-Pitts neurons, each with a threshold of 2, are shown in Figures 1.14–1.17. The activation of unit X_i at time t is denoted $x_i(t)$. The activation of a neuron X_i at time t is determined by the activations, at time $t - 1$, of the neurons from which it receives signals.

Logic functions will be used as simple examples for a number of neural nets. The binary form of the functions for AND, OR, and AND NOT are defined here for reference. Each of these functions acts on two input values, denoted x_1 and x_2, and produces a single output value y.

AND

The AND function gives the response "true" if both input values are "true"; otherwise the response is "false." If we represent "true" by the value 1, and "false" by 0, this gives the following four training input, target output pairs:

x_1	x_2	$\rightarrow$	y
1	1		1
1	0		0
0	1		0
0	0		0

Example 1.1 A McCulloch-Pitts Neuron for the AND Function

The network in Figure 1.14 performs the mapping of the logical AND function. The threshold on unit Y is 2.

OR

The OR function gives the response "true" if either of the input values is "true"; otherwise the response is "false." This is the "inclusive or," since both input values may be "true" and the response is still "true." Representing "true" as 1, and "false" as 0, we have the following four training input, target output pairs:

x_1	x_2	$\rightarrow$	y
1	1		1
1	0		1
0	1		1
0	0		0

Example 1.2 A McCulloch-Pitts Neuron for the OR Function

The network in Figure 1.15 performs the logical OR function. The threshold on unit Y is 2.

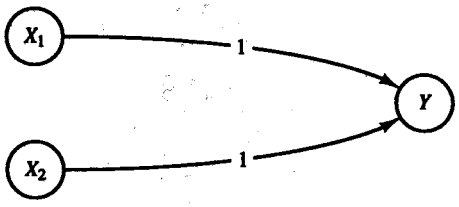

Figure 1.14 A McCulloch-Pitts neuron to perform the logical AND function.

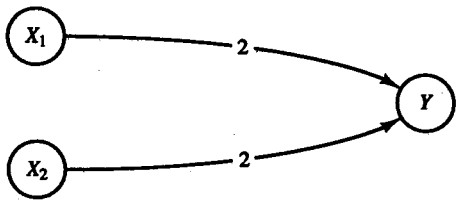

Figure 1.15 A McCulloch-Pitts neuron to perform the logical OR function.

AND NOT

The AND NOT function is an example of a logic function that is not symmetric in its treatment of the two input values. The response is "true" if the first input value, x_1, is "true" and the second input value, x_2, is "false"; otherwise the response is "false." Using a binary representation of the logical input and response values, the four training input, target output pairs are:

x_1	x_2	$\rightarrow$	y
1	1		0
1	0		1
0	1		0
0	0		0

Example 1.3 A McCulloch-Pitts Neuron for the AND NOT Function

The net in Figure 1.16 performs the function x_1 AND NOT x_2. In other words, neuron Y fires at time t if and only if unit X_1 fires at time $t - 1$ and unit X_2 does not fire at time $t - 1$. The threshold for unit Y is 2.

1.6.3 Applications

XOR

The XOR (exclusive or) function gives the response "true" if exactly one of the input values is "true"; otherwise the response is "false." Using a binary representation, the four training input, target output pairs are:

x_1	x_2	$\rightarrow$	y
1	1		0
1	0		1
0	1		1
0	0		0

Example 1.4 A McCulloch-Pitts Net for the XOR Function

The network in Figure 1.17 performs the logical XOR function. XOR can be expressed as

$$x_1 \text{ XOR } x_2 \leftrightarrow (x_1 \text{ AND NOT } x_2) \text{ OR } (x_2 \text{ AND NOT } x_1).$$

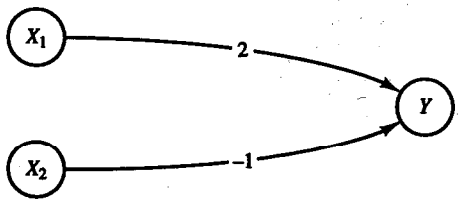

Figure 1.16 A McCulloch-Pitts neuron to perform the logical AND NOT function.

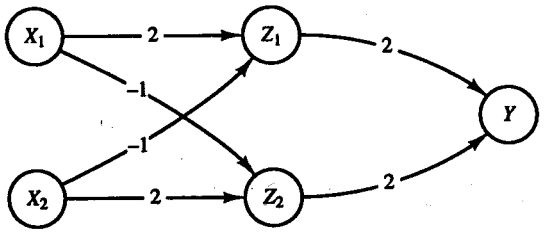

Figure 1.17 A McCulloch-Pitts neural net to perform the logical XOR function.

Thus, $y = x_1$ XOR x_2 is found by a two-layer net. The first layer forms

$$z_1 = x_1 \text{ AND NOT } x_2$$

and

$$z_2 = x_2 \text{ AND NOT } x_1.$$

The second layer consists of

$$y = z_1 \text{ OR } z_2.$$

Units Z_1, Z_2, and Y each have a threshold of 2.

Hot and cold

Example 1.5 Modeling the Perception of Hot and Cold with a McCulloch-Pitts Net

It is a well-known and interesting physiological phenomenon that if a cold stimulus is applied to a person's skin for a very short period of time, the person will perceive heat. However, if the same stimulus is applied for a longer period, the person will perceive cold. The use of discrete time steps enables the network of McCulloch-Pitts neurons shown in Figure 1.18 to model this phenomenon. The example is an elaboration of one originally presented by McCulloch and Pitts [1943]. The model is designed to give only the first perception of heat or cold that is received by the perceptor units.

In the figure, neurons X_1 and X_2 represent receptors for heat and cold, respectively, and neurons Y_1 and Y_2 are the counterpart perceptors. Neurons Z_1

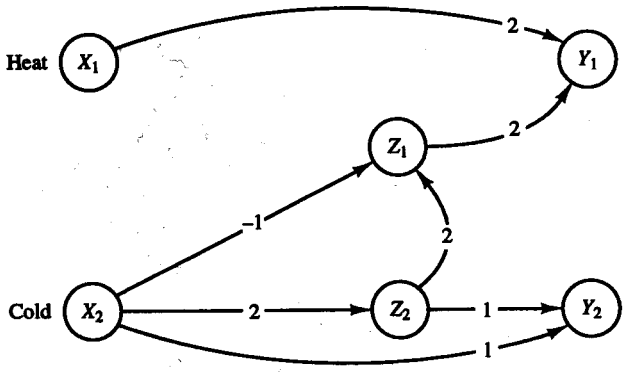

Figure 1.18 A network of McCulloch-Pitts neurons to model the perception of heat and cold.

and Z_2 are auxiliary units needed for the problem. Each neuron has a threshold of 2, i.e., it fires (sets its activation to 1) if the net input it receives is ≥ 2. Input to the system will be (1,0) if heat is applied and (0,1) if cold is applied. The desired response of the system is that cold is perceived if a cold stimulus is applied for two time steps, i.e.,

$$y_2(t) = x_2(t - 2) \text{ AND } x_2(t - 1).$$

The activation of unit Y_2 at time t is $y_2(t)$; $y_2(t) = 1$ if cold is perceived, and $y_2(t) = 0$ if cold is not perceived.

In order to model the physical phenomenon described, it is also required that heat be perceived if either a hot stimulus is applied or a cold stimulus is applied briefly (for one time step) and then removed. This condition is expressed as

$$y_1(t) = \{x_1(t - 1)\} \text{ OR } \{x_2(t - 3) \text{ AND NOT } x_2(t - 2)\}.$$

To see that the net shown in Figure 1.18 does in fact represent the two logical statements required, consider first the neurons that determine the response of Y_1 at time t (illustrated in Figure 1.19). The figure shows that

$$y_1(t) = x_1(t - 1) \text{ OR } z_1(t - 1).$$

Now consider the neurons (illustrated in Figure 1.20) that determine the response of unit Z_1 at time $t - 1$. This figure shows that

$$z_1(t - 1) = z_2(t - 2) \text{ AND NOT } x_2(t - 2).$$

Finally, the response of unit Z_2 at time $t - 2$ is simply the value of X_2 at the previous time step:

$$z_2(t - 2) = x_2(t - 3).$$

Substituting in the preceding expression for $y_1(t)$ gives

$$y_1(t) = \{x_1(t - 1)\} \text{ OR } \{x_2(t - 3) \text{ AND NOT } x_2(t - 2)\}.$$

The analysis for the response of neuron Y_2 at time t proceeds in a similar manner. Figure 1.21 shows that $y_2(t) = z_2(t - 1)$ AND $x_2(t - 1)$. However, as

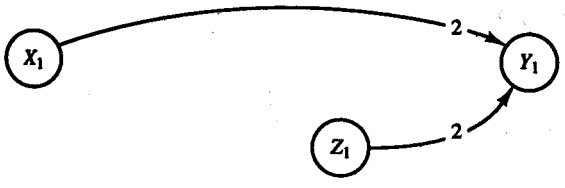

Figure 1.19 The neurons that determine the response of unit Y_1.

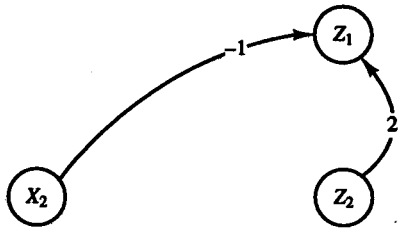

Figure 1.20 The neurons that determine the response of unit Z_1.

before, $z_2(t-1) = x_2(t-2)$; substituting in the expression for $y_2(t)$ then gives

$$y_2(t) = x_2(t-2) \text{ AND } x_2(t-1),$$

as required.

It is also informative to trace the flow of activations through the net, starting with the presentation of a stimulus at $t = 0$. Case 1, a cold stimulus applied for one time step and then removed, is illustrated in Figure 1.22. Case 2, a cold stimulus applied for two time steps, is illustrated in Figure 1.23, and case 3, a hot stimulus applied for one time step, is illustrated in Figure 1.24. In each case, only the activations that are known at a particular time step are indicated. The weights on the connections are as in Figure 1.18.

Case 1: A cold stimulus applied for one time step.

The activations that are known at $t = 0$ are shown in Figure 1.22(a).

The activations that are known at $t = 1$ are shown in Figure 1.22(b). The activations of the input units are both 0, signifying that the cold stimulus presented at $t = 0$ was removed after one time step. The activations of Z_1 and Z_2 are based on the activations of X_2 at $t = 0$.

The activations that are known at $t = 2$ are shown in Figure 1.22(c). Note that the activations of the input units are not specified, since their value at $t = 2$ does not determine the first response of the net to the situation being modeled. Although the responses of the perceptor units are determined, no perception of hot or cold has reached them yet.

The activations that are known at $t = 3$ are shown in Figure 1.22(d). A perception of heat is indicated by the fact that unit Y_1 has an activation of 1 and unit Y_2 has an activation of 0.

Case 2: A cold stimulus applied for two time steps.

The activations that are known at $t = 0$ are shown in Figure 1.23(a), and those that are known at $t = 1$ are shown in Figure 1.23(b).

The activations that are known at $t = 2$ are shown in Figure 1.23(c). Note that the activations of the input units are not specified, since the first response of the net to the cold stimulus being applied for two time steps is not influenced by whether or not the stimulus is removed after the two steps. Although the responses of the auxiliary units Z_1 and Z_2 are indicated, the responses of the perceptor units are determined by the activations of all of the units at $t = 1$.

Case 3: A hot stimulus applied for one time step.

The activations that are known at $t = 0$ are shown in Figure 1.24(a).

The activations that are known at $t = 1$ are shown in Figure 1.24(b). Unit Y_1 fires because it has received a signal from X_1. Y_2 does not fire because it requires input signals from both X_2 and Z_2 in order to fire, and X_2 had an activation of 0 at $t = 0$.

Figure 1.21 The neurons that determine the response of unit Y_2.

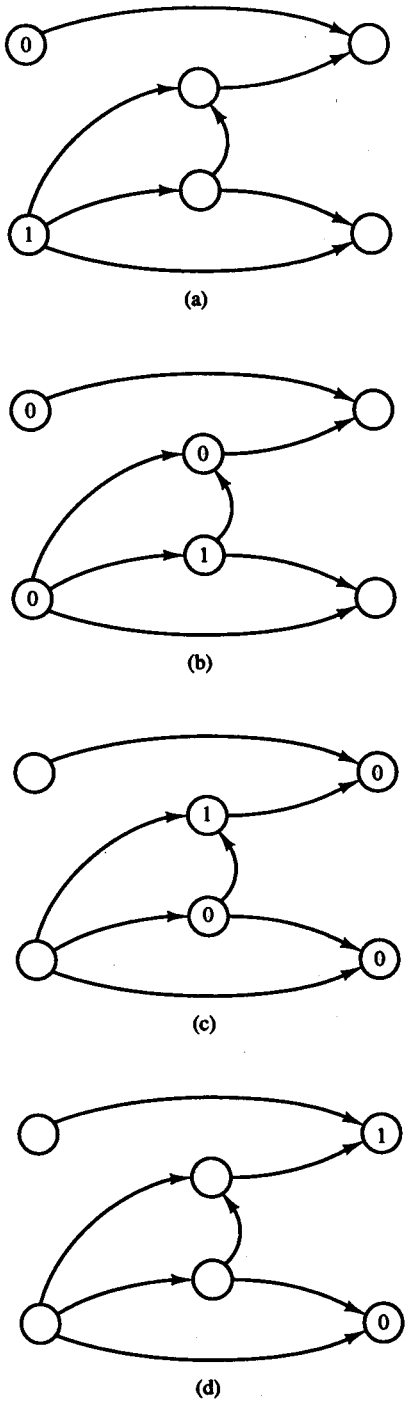

Figure 1.22 A cold stimulus applied for one time step.
Activations at (a) $t = 0$, (b) $t = 1$, (c) $t = 2$, and (d) $t = 3$.

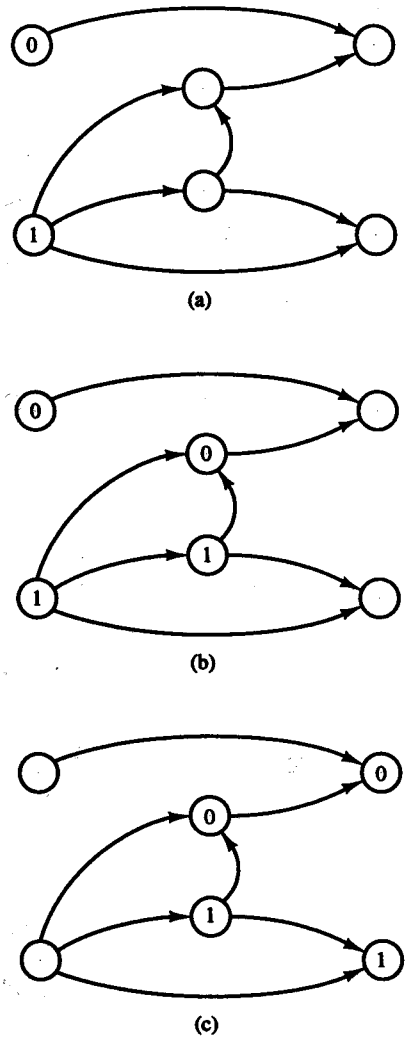

Figure 1.23 A cold stimulus applied for two time steps. Activations at (a) $t = 0$, (b) $t = 1$, and (c) $t = 2$.

1.7 SUGGESTIONS FOR FURTHER STUDY

1.7.1 Readings

Many of the applications and historical developments we have summarized in this chapter are described in more detail in two collections of original research:

- *Neurocomputing: Foundations of Research* [Anderson & Rosenfeld, 1988].
- *Neurocomputing 2: Directions for Research* [Anderson, Pellionisz & Rosenfeld, 1990].

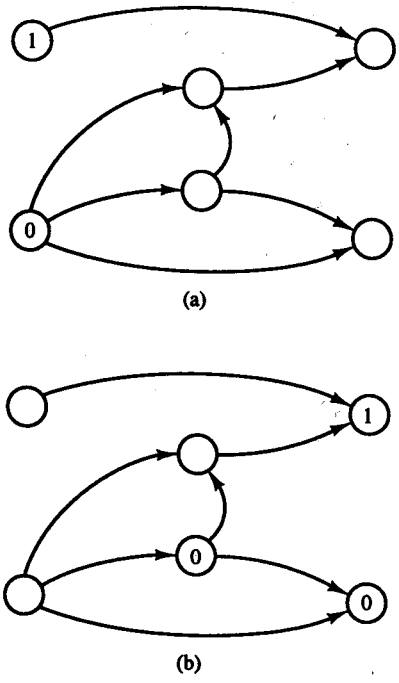

Figure 1.24 A hot stimulus applied for one time step.
Activations at (a) $t = 0$ and (b) $t = 1$.

These contain useful papers, along with concise and insightful introductions explaining the significance and key results of each paper.

The *DARPA Neural Network Study* (1988) also provides descriptions of both the theoretical and practical state of the art of neural networks that year.

Nontechnical introductions

Two very readable nontechnical introductions to neural networks, with an emphasis on the historical development and the personalities of the leaders in the field, are:

- *Cognizers* [Johnson & Brown, 1988].
- *Apprentices of Wonder: Inside the Neural Network Revolution* [Allman, 1989].

Applications

Among the books dealing with neural networks for particular types of applications are:

- *Neural Networks for Signal Processing* [Kosko, 1992b].
- *Neural Networks for Control* [Miller, Sutton, & Werbos, 1990].

- *Simulated Annealing and Boltzmann Machines* [Aarts & Korst, 1989].
- *Adaptive Pattern Recognition and Neural Networks* [Pao, 1989].
- *Adaptive Signal Processing* [Widrow & Sterns, 1985].

History

The history of neural networks is a combination of progress in experimental work with biological neural systems, computer modeling of biological neural systems, the development of mathematical models and their applications to problems in a wide variety of fields, and hardware implementation of these models. In addition to the collections of original papers already mentioned, in which the introductions to each paper provide historical perspectives, *Embodiments of Mind* [McCulloch, 1988] is a wonderful selection of some of McCulloch's essays. *Perceptrons* [Minsky & Papert, 1988] also places the development of neural networks into historical context.

Biological neural networks

Introduction to Neural and Cognitive Modeling [Levine, 1991] provides extensive information on the history of neural networks from a mathematical and psychological perspective. For additional writings from a biological point of view, see *Neuroscience and Connectionist Theory* [Gluck & Rumelhart, 1990] and *Neural and Brain Modeling* [MacGregor, 1987].

1.7.2 Exercises

1.1 Consider the neural network of McCulloch-Pitts neurons shown in Figure 1.25. Each neuron (other than the input neurons, N_1 and N_2) has a threshold of 2.

a. Define the response of neuron N_5 at time t in terms of the activations of the input neurons, N_1 and N_2, at the appropriate time.

b. Show the activation of each neuron that results from an input signal of $N_1 = 1$, $N_2 = 0$ at $t = 0$.

1.2 There are at least two ways to express XOR in terms of other simple logic functions that can be represented by McCulloch-Pitts neurons. One such example is presented in Section 1.6. Find another representation and the net for it. How do the two nets (yours and the one in Section 1.6) compare?

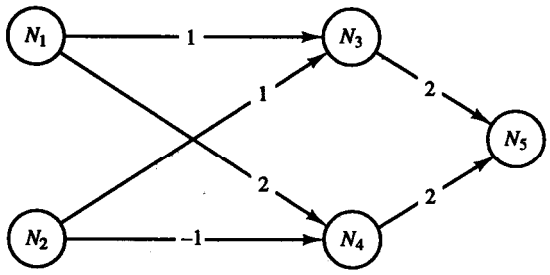

Figure 1.25 Neural network for Exercise 1.1.

1.3 In the McCulloch-Pitts model of the perception of heat and cold, a cold stimulus applied at times $t - 2$ and $t - 1$ is perceived as cold at time t. Can you modify the net to require the cold stimulus to be applied for three time steps before cold is felt?

1.4 In the hot and cold model, consider what is felt after the first perception. (That is, if the first perception of hot or cold is at time t, what is felt at time $t + 1$?) State clearly any further assumptions as to what happens to the inputs (stimuli) that may be necessary or relevant.

1.5 Design a McCulloch-Pitts net to model the perception of simple musical patterns. Use three input units to correspond to the three pitches, "do," "re," and "mi." Assume that only one pitch is presented at any time. Use two output units to correspond to the perception of an "upscale segment" and a "downscale segment"—specifically,

a. the pattern of inputs "do" at time $t = 1$, "re" at $t = 2$, and "mi" at $t = 3$ should elicit a positive response from the "upscale segment" unit;

b. the pattern of inputs "mi" at time $t = 1$, "re" at $t = 2$, and "do" at $t = 3$ should elicit a positive response from the "downscale segment" unit;

c. any other pattern of inputs should generate no response.

You may wish to elaborate on this example, allowing for more than one input unit to be "on" at any instant of time, designing output units to detect chords, etc.

CHAPTER 2

Simple Neural Nets for Pattern Classification

2.1 GENERAL DISCUSSION

One of the simplest tasks that neural nets can be trained to perform is pattern classification. In pattern classification problems, each input vector (pattern) belongs, or does not belong, to a particular class or category. For a neural net approach, we assume we have a set of training patterns for which the correct classification is known. In the simplest case, we consider the question of membership in a single class. The output unit represents membership in the class with a response of 1; a response of -1 (or 0 if binary representation is used) indicates that the pattern is not a member of the class. For the single-layer nets described in this chapter, extension to the more general case in which each pattern may or may not belong to any of several classes is immediate. In such case, there is an output unit for each class. Pattern classification is one type of pattern recognition; the associative recall of patterns (discussed in Chapter 3) is another.

Pattern classification problems arise in many areas. In 1963, Donald Specht (a student of Widrow) used neural networks to detect heart abnormalities with EKG types of data as input (46 measurements). The output was "normal" or "abnormal," with an "on" response signifying normal [Specht, 1967; Caudill & Butler, 1990]. In the early 1960s, Minsky and Papert used neural nets to classify input patterns as convex or not convex and connected or not connected [Minsky & Papert, 1988]. There are many other examples of pattern classification problems

being solved by neural networks, both the simple nets described in this chapter, other early nets not discussed here, and multilayer nets (especially the backpropagation nets described in Chapter 6).

In this chapter, we shall discuss three methods of training a simple single-layer neural net for pattern classification: the Hebb rule, the perceptron learning rule, and the delta rule (used by Widrow in his ADALINE neural net). First, however, we discuss some issues that are common to all single-layer nets that perform pattern classification. We conclude the chapter with some comparisons of the nets discussed and an extension to a multilayer net, MADALINE.

Many real-world examples need more sophisticated architecture and training rules because the conditions for a single-layer net to be adequate (see Section 2.1.3) are not met. However, if the conditions are met approximately, the results may be sufficiently accurate. Also, insight can be gained from the more simple nets, since the meaning of the weights may be easier to interpret.

2.1.1 Architecture

The basic architecture of the simplest possible neural networks that perform pattern classification consists of a layer of input units (as many units as the patterns to be classified have components) and a single output unit. Most neural nets we shall consider in this chapter use the single-layer architecture shown in Figure 2.1. This allows classification of vectors, which are n-tuples, but considers membership in only one category.

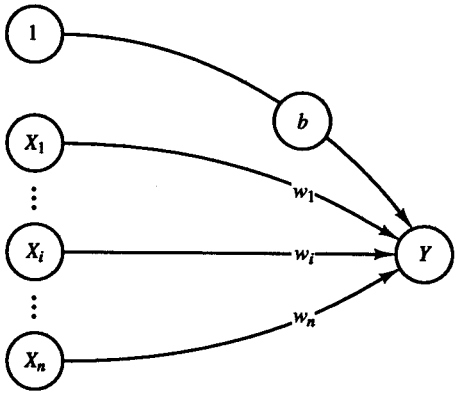

Figure 2.1 Single-layer net for pattern classification.

An example of a net that classifies the input into several categories is considered in Section 2.3.3. This net is a simple extension of the nets that perform

a single classification. The MADALINE net considered in Section 2.4.5 is a multilayer extension of the single-layer ADALINE net.

2.1.2 Biases and Thresholds

A bias acts exactly as a weight on a connection from a unit whose activation is always 1. Increasing the bias increases the net input to the unit. If a bias is included, the activation function is typically taken to be

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

where

$$\text{net} = b + \sum_i x_i w_i.$$

Some authors do not use a bias weight, but instead use a fixed threshold θ for the activation function. In that case,

$$f(\text{net}) = \begin{cases} 1 & \text{if net} \geq \theta; \\ -1 & \text{if net} < \theta; \end{cases}$$

where

$$\text{net} = \sum_i x_i w_i.$$

However, as the next example will demonstrate, this is essentially equivalent to the use of an adjustable bias.

Example 2.1 The role of a bias or a threshold

In this example and in the next section, we consider the separation of the input space into regions where the response of the net is positive and regions where the response is negative. To facilitate a graphical display of the relationships, we illustrate the ideas for an input with two components while the output is a scalar (i.e., it has only one component). The architecture of these examples is given in Figure 2.2.

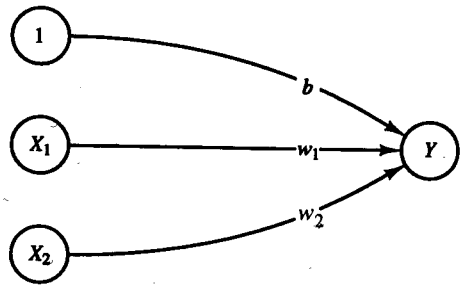

Figure 2.2 Single-layer neural network for logic functions.

The boundary between the values of x_1 and x_2 for which the net gives a positive response and the values for which it gives a negative response is the separating line

$$b + x_1 w_1 + x_2 w_2 = 0,$$

or (assuming that $w_2 \neq 0$),

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}.$$

The requirement for a positive response from the output unit is that the net input it receives, namely, $b + x_1 w_1 + x_2 w_2$, be greater than 0. During training, values of w_1, w_2, and b are determined so that the net will have the correct response for the training data.

If one thinks in terms of a threshold, the requirement for a positive response from the output unit is that the net input it receives, namely, $x_1 w_1 + x_2 w_2$, be greater than the threshold. This gives the equation of the line separating positive from negative output as

$$x_1 w_1 + x_2 w_2 = \theta,$$

or (assuming that $w_2 \neq 0$),

$$x_2 = -\frac{w_1}{w_2} x_1 + \frac{\theta}{w_2}.$$

During training, values of w_1 and w_2 are determined so that the net will have the correct response for the training data. In this case, the separating line cannot pass through the origin, but a line can be found that passes arbitrarily close to the origin.

The form of the separating line found by using an adjustable bias and the form obtained by using a fixed threshold illustrate that there is no advantage to including both a bias and a nonzero threshold for a neuron that uses the step function as its activation function. On the other hand, including neither a bias nor a threshold is equivalent to requiring the separating line (or plane or hyperplane for inputs with more components) to pass through the origin. This may or may not be appropriate for a particular problem.

As an illustration of a pseudopsychological analogy to the use of a bias, consider a simple (artificial) neural net in which the activation of the neuron corresponds to a person's action, "Go to the ball game." Each input signal corresponds to some factor influencing the decision to "go" or "not go" (other possible activities, the weather conditions, information about who is pitching, etc.). The weights on these input signals correspond to the importance the person places on each factor. (Of course, the weights may change with time, but methods for modifying them are not considered in this illustration.) A bias could represent a general inclination to "go" or "not go," based on past experiences. Thus, the bias would be modifiable, but the signal to it would not correspond to information about the specific game in question or activities competing for the person's time.

The threshold for this "decision neuron" indicates the total net input necessary to cause the person to "go," i.e., for the decision neuron to fire. The threshold would be different for different people; however, for the sake of this simple example, it should be thought of as a quantity that remains fixed for each individual. Since it is the relative values of the weights, rather than their actual magnitudes, that determine the response of the neuron, the model can cover all possibilities using either the fixed threshold or the adjustable bias.

2.1.3 Linear Separability

For each of the nets in this chapter, the intent is to train the net (i.e., adaptively determine its weights) so that it will respond with the desired classification when presented with an input pattern that it was trained on or when presented with one that is sufficiently similar to one of the training patterns. Before discussing the particular nets (which is to say, the particular styles of training), it is useful to discuss some issues common to all of the nets. For a particular output unit, the desired response is a "yes" if the input pattern is a member of its class and a "no" if it is not. A "yes" response is represented by an output signal of 1, a "no" by an output signal of -1 (for bipolar signals). Since we want one of two responses, the activation (or transfer or output) function is taken to be a step function. The value of the function is 1 if the net input is positive and -1 if the net input is negative. Since the net input to the output unit is

$$y\_in = b + \sum_i x_i w_i,$$

it is easy to see that the boundary between the region where $y\_in > 0$ and the region where $y\_in < 0$, which we call the *decision boundary*, is determined by the relation

$$b + \sum_i x_i w_i = 0.$$

Depending on the number of input units in the network, this equation represents a line, a plane, or a hyperplane.

If there are weights (and a bias) so that all of the training input vectors for which the correct response is $+1$ lie on one side of the decision boundary and all of the training input vectors for which the correct response is -1 lie on the other side of the decision boundary, we say that the problem is "linearly separable." Minsky and Papert [1988] showed that a single-layer net can learn only linearly separable problems. Furthermore, it is easy to extend this result to show that multilayer nets with linear activation functions are no more powerful than single-layer nets (since the composition of linear functions is linear).

It is convenient, if the input vectors are ordered pairs (or at most ordered triples), to graph the input training vectors and indicate the desired response by the appropriate symbol ("+" or "−"). The analysis also extends easily to nets

with more input units; however, the graphical display is not as convenient. The region where y is positive is separated from the region where it is negative by the line

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}.$$

These two regions are often called *decision regions* for the net. Notice in the following examples that there are many different lines that will serve to separate the input points that have different target values. However, for any particular line, there are also many choices of w_1, w_2, and b that give exactly the same line. The choice of the sign for b determines which side of the separating line corresponds to a $+1$ response and which side to a -1 response.

There are four different bipolar input patterns we can use to train a net with two input units. However, there are two possible responses for each input pattern, so there are 2^4 different functions that we might be able to train a very simple net to perform. Several of these functions are familiar from elementary logic, and we will use them for illustrations, for convenience. The first question we consider is, For this very simple net, do weights exist so that the net will have the desired output for each of the training input vectors?

Example 2.2 Response regions for the AND function

The AND function (for bipolar inputs and target) is defined as follows:

INPUT (x_1, x_2)	OUTPUT (t)
(1, 1)	+1
(1, −1)	−1
(−1, 1)	−1
(−1, −1)	−1

The desired responses can be illustrated as shown in Figure 2.3. One possible decision boundary for this function is shown in Figure 2.4.

An example of weights that would give the decision boundary illustrated in the figure, namely, the separating line

$$x_2 = -x_1 + 1,$$

is

$$\begin{aligned} b &= -1, \\ w_1 &= 1, \\ w_2 &= 1. \end{aligned}$$

The choice of sign for b is determined by the requirement that

$$b + x_1w_1 + x_2w_2 < 0$$

where $x_1 = 0$ and $x_2 = 0$. (Any point that is not on the decision boundary can be used to determine which side of the boundary is positive and which is negative; the origin is particularly convenient to use when it is not on the boundary.)

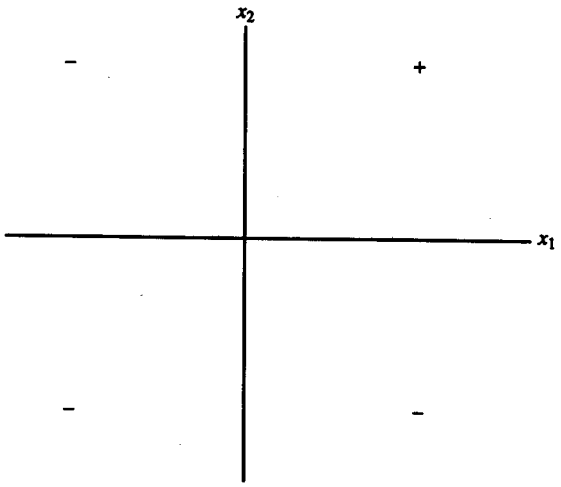

Figure 2.3 Desired response for the logic function AND (for bipolar inputs).

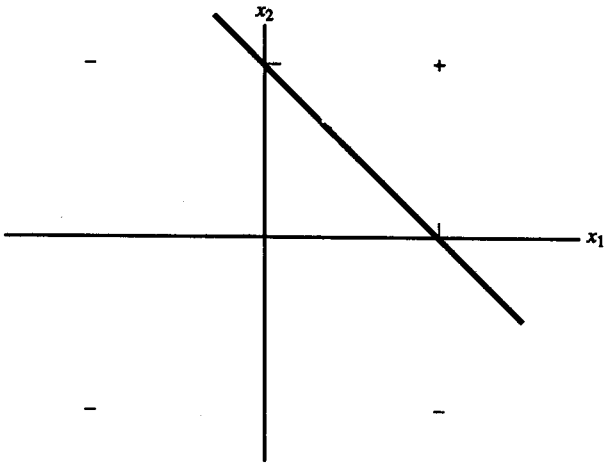

Figure 2.4 The logic function AND, showing a possible decision boundary.

Example 2.3 Response regions for the OR function

The logical OR function (for bipolar inputs and target) is defined as follows:

INPUT (x_1, x_2)	OUTPUT (t)
(1, 1)	+1
(1, −1)	+1
(−1, 1)	+1
(−1, −1)	−1

The weights must be chosen to provide a separating line, as illustrated in Figure 2.5. One example of suitable weights is

$$b = 1,$$

$$w_1 = 1,$$

$$w_2 = 1,$$

giving the separating line

$$x_2 = -x_1 - 1.$$

The choice of sign for b is determined by the requirement that

$$b + x_1 w_1 + x_2 w_2 > 0$$

where $x_1 = 0$ and $x_2 = 0$.

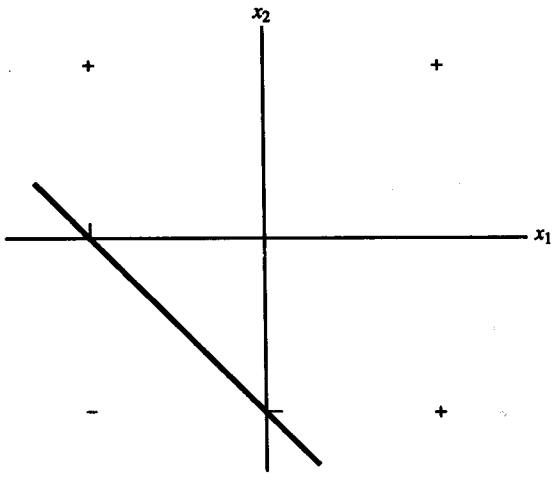

Figure 2.5 The logic function OR, showing a possible decision boundary.

The preceding two mappings (which can each be solved by a single-layer neural net) illustrate graphically the concept of linearly separable input. The input points to be classified positive can be separated from the input points to be classified negative by a straight line. The equations of the decision boundaries are not unique. We will return to these examples to illustrate each of the learning rules in this chapter.

Note that if a bias weight were not included in these examples, the decision boundary would be forced to go through the origin. In many cases (including Examples 2.2 and 2.3), that would change a problem that could be solved (i.e., learned, or one for which weights exist) into a problem that could not be solved.

Not all simple two-input, single-output mappings can be solved by a single-layer net (even with a bias included), as is illustrated in Example 2.4.

Example 2.4 Response regions for the XOR function

The desired response of this net is as follows:

INPUT (x_1, x_2)	OUTPUT (t)
(1, 1)	−1
(1, −1)	+1
(−1, 1)	+1
(−1, −1)	−1

It is easy to see that no single straight line can separate the points for which a positive response is desired from those for which a negative response is desired.

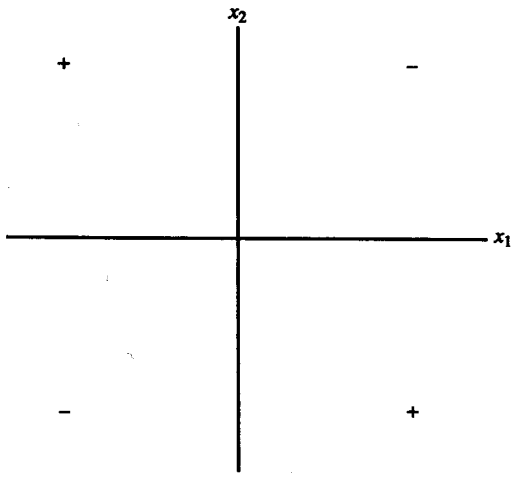

Figure 2.6 Desired response for the logic function XOR.

2.1.4 Data Representation

The previous examples show the use of a bipolar (values 1 and −1) representation of the training data, rather than the binary representation used for the McCulloch-Pitts neurons in Chapter 1. Many early neural network models used binary representation, although in most cases it can be modified to bipolar form. The form of the data may change the problem from one that can be solved by a simple neural net to one that cannot, as is illustrated in Examples 2.5–2.7 for the Hebb rule. Binary representation is also not as good as bipolar if we want the net to generalize (i.e., respond to input data similar, but not identical to, training data). Using bipolar input, missing data can be distinguished from mistaken data. Missing values can be represented by "0" and mistakes by reversing the input value from +1 to −1, or vice versa. We shall discuss some of the issues relating to the choice of binary versus bipolar representation further as they apply to particular neural nets. In general, bipolar representation is preferable.

The remainder of this chapter focuses on three methods of training single-layer neural nets that are useful for pattern classification: the Hebb rule, the perceptron learning rule, and the delta rule (or least mean squares). The Hebb rule, or correlational learning, is extremely simple but limited (even for linearly separable problems); the training algorithms for the perceptron and for ADALINE (adaptive linear neuron, trained by the delta rule) are closely related. Both are iterative techniques that are guaranteed to converge under suitable circumstances. The generalization of an ADALINE to a multilayer net (MADALINE) also will be examined.

2.2 HEBB NET

The earliest and simplest learning rule for a neural net is generally known as the Hebb rule. Hebb proposed that learning occurs by modification of the synapse strengths (weights) in a manner such that if two interconnected neurons are both "on" at the same time, then the weight between those neurons should be increased. The original statement only talks about neurons firing at the same time (and does not say anything about reinforcing neurons that do not fire at the same time). However, a stronger form of learning occurs if we also increase the weights if both neurons are "off" at the same time. We use this extended Hebb rule [McClelland & Rumelhart, 1988] because of its improved computational power and shall refer to it as the Hebb rule.

We shall refer to a single-layer (feedforward) neural net trained using the (extended) Hebb rule as a *Hebb net*. The Hebb rule is also used for training other specific nets that are discussed later. Since we are considering a single-layer net, one of the interconnected neurons will be an input unit and one an output unit (since no input units are connected to each other, nor are any output units in-

terconnected). If data are represented in bipolar form, it is easy to express the desired weight update as

$$w_i(\text{new}) = w_i(\text{old}) + x_i y.$$

If the data are binary, this formula does not distinguish between a training pair in which an input unit is "on" and the target value is "off" and a training pair in which both the input unit and the target value are "off." Examples 2.5 and 2.6 (in Section 2.2.2) illustrate the extreme limitations of the Hebb rule for binary data. Example 2.7 shows the improved performance achieved by using bipolar representation for both the input and target values.

2.2.1 Algorithm

Step 0. Initialize all weights:

$$w_i = 0 \quad (i = 1 \text{ to } n).$$

Step 1. For each input training vector and target output pair, **s** : t, do steps 2–4.

Step 2. Set activations for input units:

$$x_i = s_i \quad (i = 1 \text{ to } n).$$

Step 3. Set activation for output unit:

$$y = t.$$

Step 4. Adjust the weights for

$$w_i(\text{new}) = w_i(\text{old}) + x_i y \quad (i = 1 \text{ to } n).$$

Adjust the bias:

$$b(\text{new}) = b(\text{old}) + y.$$

Note that the bias is adjusted exactly like a weight from a "unit" whose output signal is always 1. The weight update can also be expressed in vector form as

$$\mathbf{w}(\text{new}) = \mathbf{w}(\text{old}) + \mathbf{x}y.$$

This is often written in terms of the weight change, $\Delta\mathbf{w}$, as

$$\Delta\mathbf{w} = \mathbf{x}y$$

and

$$\mathbf{w}(\text{new}) = \mathbf{w}(\text{old}) + \Delta\mathbf{w}.$$

There are several methods of implementing the Hebb rule for learning. The foregoing algorithm requires only one pass through the training set; other equivalent methods of finding the weights are described in Section 3.1.1, where the Hebb rule for pattern association (in which the target is a vector) is presented.

2.2.2 Application

Bias types of inputs are not explicitly used in the original formulation of Hebb learning. However, they are included in the examples in this section (shown as a third input component that is always 1) because without them, the problems discussed cannot be solved.

Logic functions

Example 2.5 A Hebb net for the AND function: binary inputs and targets

INPUT			TARGET
$(x_1$	x_2	1)	
(1	1	1)	1
(1	0	1)	0
(0	1	1)	0
(0	0	1)	0

For each training input: target, the weight change is the product of the input vector and the target value, i.e.,

$$\Delta w_1 = x_1 t, \qquad \Delta w_2 = x_2 t, \qquad \Delta b = t.$$

The new weights are the sum of the previous weights and the weight change. Only one iteration through the training vectors is required. The weight updates for the first input are as follows:

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)		$(\Delta w_1$	Δw_2	$\Delta b)$	$(w_1$	w_2	$b)$
							(0	0	0)
(1	1	1)	1	(1	1	1)	(1	1	1)

The separating line (see Section 2.1.3) becomes

$$x_2 = -x_1 - 1.$$

The graph, presented in Figure 2.7, shows that the response of the net will now be correct for the first input pattern. Presenting the second, third, and fourth training inputs shows that because the target value is 0, no learning occurs. Thus, using binary target values prevents the net from learning any pattern for which the target is "off":

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)		$(\Delta w_1$	Δw_2	$\Delta b)$	$(w_1$	w_2	$b)$
(1	0	1)	0	(0	0	0)	(1	1	1)
(0	1	1)	0	(0	0	0)	(1	1	1)
(0	0	1)	0	(0	0	0)	(1	1	1)

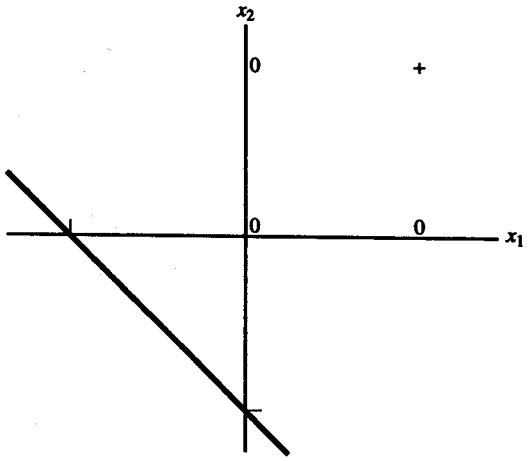

Figure 2.7 Decision boundary for binary AND function using Hebb rule after first training pair.

Example 2.6 A Hebb net for the AND function: binary inputs, bipolar targets

INPUT			TARGET
(x_1	x_2	1)	
(1	1	1)	1
(1	0	1)	−1
(0	1	1)	−1
(0	0	1)	−1

Presenting the first input, including a value of 1 for the third component, yields the following:

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
(x_1	x_2	1)		(Δw_1	Δw_2	Δb)	(w_1	w_2	b)
							(0	0	0)
(1	1	1)	1	(1	1	1)	(1	1	1)

The separating line becomes

$$x_2 = -x_1 - 1.$$

Figure 2.8 shows that the response of the net will now be correct for the first input pattern.

Presenting the second, third, and fourth training patterns shows that learning continues for each of these patterns (since the target value is now −1, rather than 0, as in Example 2.5).

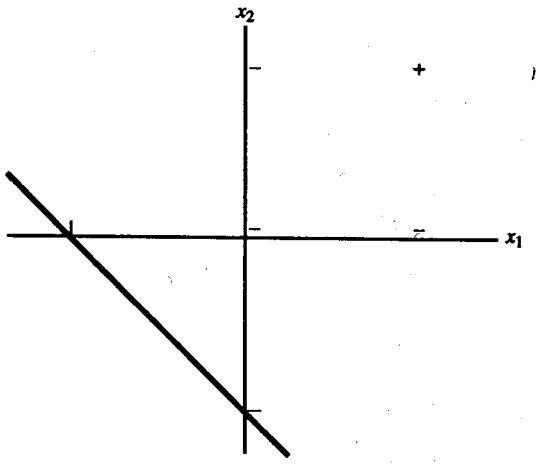

Figure 2.8 Decision boundary for AND function using Hebb rule after first training pair (binary inputs, bipolar targets).

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)		$(\Delta w_1$	Δw_2	$\Delta b)$	$(w_1$	w_2	$b)$
(1	0	1)	−1	(−1	0	−1)	(0	1	0)
(0	1	1)	−1	(0	−1	−1)	(0	0	−1)
(0	0	1)	−1	(0	0	−1)	(0	0	−2)

However, these weights do not provide the correct response for the first input pattern.

The choice of training patterns can play a significant role in determining which problems can be solved using the Hebb rule. The next example shows that the AND function can be solved if we modify its representation to express the inputs as well as the targets in bipolar form. Bipolar representation of the inputs and targets allows modification of a weight when the input unit and the target value are both "on" at the same time and when they are both "off" at the same time. The algorithm is the same as that just given, except that now all units will learn whenever there is an error in the output.

Example 2.7 A Hebb net for the AND function: bipolar inputs and targets

INPUT			TARGET
$(x_1$	x_2	1)	
(1	1	1)	1
(1	−1	1)	−1
(−1	1	1)	−1
(−1	−1	1)	−1

Presenting the first input, including a value of 1 for the third component, yields the following:

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			$(0 \quad 0 \quad 0)$
$(1 \quad 1 \quad 1)$	1	$(1 \quad 1 \quad 1)$	$(1 \quad 1 \quad 1)$

The separating line becomes

$$x_2 = -x_1 - 1.$$

The graph in Figure 2.9 shows that the response of the net will now be correct for the first input point (and also, by the way, for the input point $(-1, -1)$). Presenting the second input vector and target results in the following situation:

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			$(1 \quad 1 \quad 1)$
$(1 \quad -1 \quad 1)$	-1	$(-1 \quad 1 \quad -1)$	$(0 \quad 2 \quad 0)$

The separating line becomes

$$x_2 = 0.$$

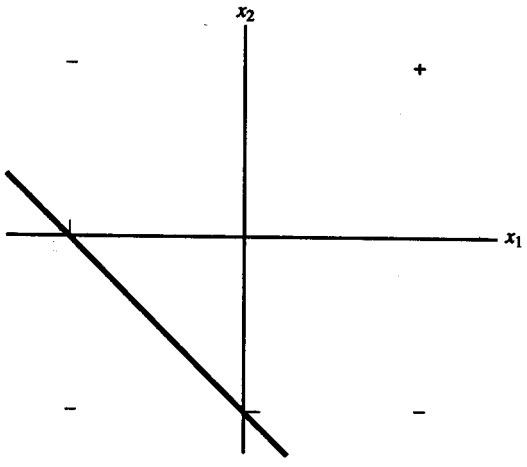

Figure 2.9 Decision boundary for the AND function using Hebb rule after first training pair (bipolar inputs and targets).

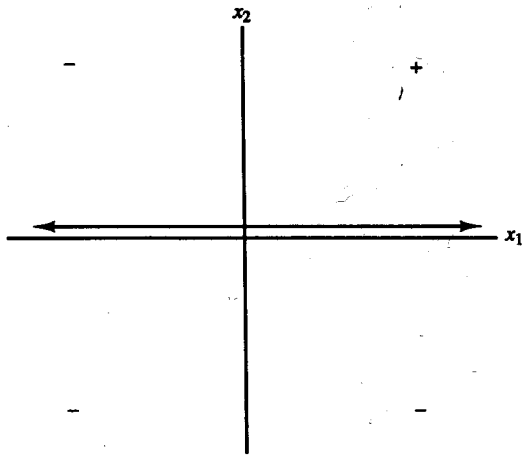

Figure 2.10 Decision boundary for bipolar AND function using Hebb rule after second training pattern (boundary is x_1-axis).

The graph in Figure 2.10 shows that the response of the net will now be correct for the first two input points, (1, 1) and (1, −1), and also, incidentally, for the input point (−1, −1). Presenting the third input vector and target yields the following:

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			(0 2 0)
(−1 1 1)	−1	(1 −1 −1)	(1 1 −1)

The separating line becomes

$$x_2 = -x_1 + 1.$$

The graph in Figure 2.11 shows that the response of the net will now be correct for the first three input points (and also, by the way, for the input point (−1, −1)). Presenting the last point, we obtain the following:

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			(1 1 −1)
(−1 −1 1)	−1	(1 1 −1)	(2 2 −2)

Even though the weights have changed, the separating line is still

$$x_2 = -x_1 + 1,$$

so the graph of the decision regions (the positive response and the negative response) remains as in Figure 2.11.

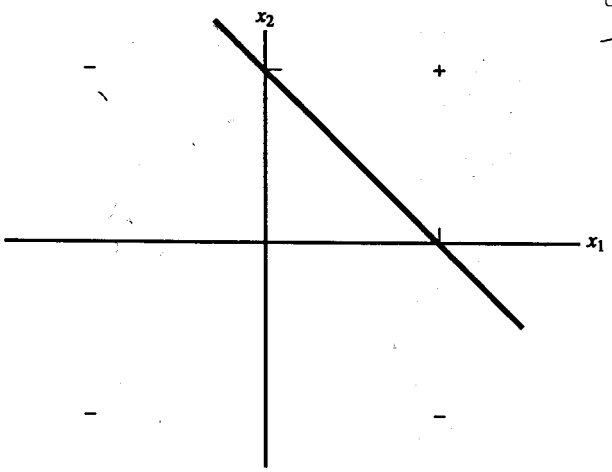

Figure 2.11 Decision boundary for bipolar AND function using Hebb rule after third training pattern.

Character recognition

Example 2.8 A Hebb net to classify two-dimensional input patterns (representing letters)

A simple example of using the Hebb rule for character recognition involves training the net to distinguish between the pattern "X" and the pattern "O". The patterns can be represented as

```
# . . . #          . # # # .
. # . # .          # . . . #
. . # . .    and   # . . . #
. # . # .          # . . . #
# . . . #          . # # # .
```

Pattern 1 Pattern 2

To treat this example as a pattern classification problem with one output class, we will designate that class "X" and take the pattern "O" to be an example of output that is not "X."

The first thing we need to do is to convert the patterns to input vectors. That is easy to do by assigning each # the value 1 and each "." the value −1. To convert from the two-dimensional pattern to an input vector, we simply concatenate the rows, i.e., the second row of the pattern comes after the first row, the third row follows, ect. Pattern 1 then becomes

1 −1 −1 −1 1, −1 1 −1 1 −1, −1 −1 1 −1 −1, −1 1 −1 1 −1,
1 −1 −1 −1 1,

and pattern 2 becomes

−1 1 1 1 −1, 1 −1 −1 −1 1, 1 −1 −1 −1 1, 1 −1 −1 −1 1, −1 1 1 1 −1,

where a comma denotes the termination of a line of the original matrix. For computer simulations, the program can be written so that the vector is read in from the two-dimensional format.

The correct response for the first pattern is "on," or +1, so the weights after presenting the first pattern are simply the input pattern. The bias weight after presenting this is +1. The correct response for the second pattern is "off," or −1, so the weight change when the second pattern is presented is

1 −1 −1 −1 1, −1 1 1 1 −1, −1 1 1 1 −1, −1 1 1 1 −1, 1 −1 −1 −1 1.

In addition, the weight change for the bias weight is −1.

Adding the weight change to the weights representing the first pattern gives the final weights:

2 −2 −2 −2 2, −2 2 0 2 −2, −2 0 2 0 −2, −2 2 0 2 −2, 2 −2 −2 −2 2.

The bias weight is 0.

Now, we compute the output of the net for each of the training patterns. The net input (for any input pattern) is the dot product of the input pattern with the weight vector. For the first training vector, the net input is 42, so the response is positive, as desired. For the second training pattern, the net input is −42, so the response is clearly negative, also as desired.

However, the net can also give reasonable responses to input patterns that are similar, but not identical, to the training patterns. There are two types of changes that can be made to one of the input patterns that will generate a new input pattern for which it is reasonable to expect a response. The first type of change is usually referred to as "mistakes in the data." In this case, one or more components of the input vector (corresponding to one or more pixels in the original pattern) have had their sign reversed, changing a 1 to a −1, or vice versa. The second type of change is called "missing data." In this situation, one or more of the components of the input vector have the value 0, rather than 1 or −1. In general, a net can handle more missing components than wrong components; in other words, with input data, "It's better not to guess."

Other simple examples

Example 2.9 Limitations of Hebb rule training for binary patterns

This example shows that the Hebb rule may fail, even if the problem is linearly separable (and even if 0 is not the target).

Consider the following input and target output pairs:

1	1	1	→	1
1	1	0	→	0
1	0	1	→	0
0	1	1	→	0

It is easy to see that the Hebb rule cannot learn any pattern for which the target is 0. So we must at least convert the targets to +1 and −1. Now consider

1	1	1	→	1
1	1	0	→	−1
1	0	1	→	−1
0	1	1	→	−1

Figure 2.12 shows that the problem is now solvable, i.e., the input points classified in one class (with target value +1) are linearly separable from those not in the class (with target value −1). The figure also shows that a nonzero bias will be necessary, since the separating plane does not pass through the origin. The plane pictured is $x_1 + x_2 + x_3 + (-2.5) = 0$, i.e., a weight vector of (1 1 1) and a bias of −2.5.

The weights (and bias) are found by taking the sum of the weight changes that occur at each stage of the algorithm. The weight change is simply the input pattern (augmented by the fourth component, the input to the bias weight, which is always 1) multiplied by the target value for the pattern. We obtain:

Weight change for first input pattern:	1	1	1	1
Weight change for second input pattern:	−1	−1	0	−1
Weight change for third input pattern:	−1	0	−1	−1
Weight change for fourth input pattern:	0	−1	−1	−1
Final weights (and bias)	−1	−1	−1	−2

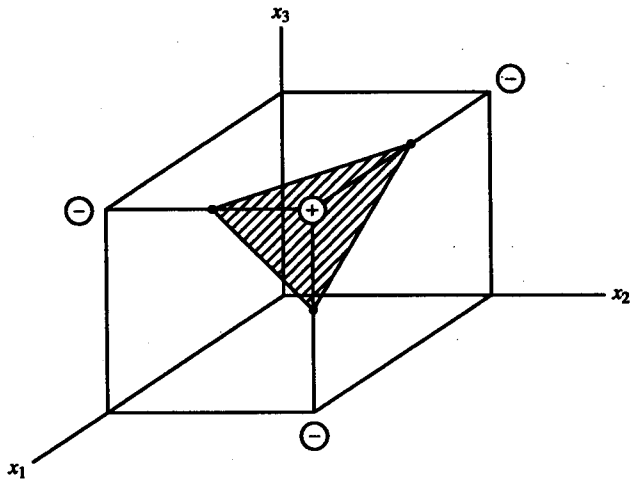

Figure 2.12 Linear separation of binary training inputs.

It is easy to see that these weights do not produce the correct output for the first pattern.

Example 2.10 Limitation of Hebb rule training for bipolar patterns

Examples 2.5, 2.6, and 2.7 show that even if the representation of the vectors does not change the problem from unsolvable to solvable, it can affect whether the Hebb rule works. In this example, we consider the same problem as in Example 2.9, but with the input points (and target classifications) in bipolar form. Accordingly, we have the following arrangement of values:

INPUT				TARGET	WEIGHT CHANGE				WEIGHT			
(x_1	x_2	x_3	1)		(Δw_1	Δw_2	Δw_3	Δb)	(w_1	w_2	w_3	b)
									(0	0	0	0)
(1	1	1	1)	1	(1	1	1	1)	(1	1	1	1)
(1	1	−1	1)	−1	(−1	−1	1	−1)	(0	0	2	0)
(1	−1	1	1)	−1	(−1	1	−1	−1)	(−1	1	1	−1)
(−1	1	1	1)	−1	(1	−1	−1	−1)	(0	0	0	−2)

Again, it is clear that the weights do not give the correct output for the first input pattern.

Figure 2.13 shows that the input points are linearly separable; one posssible plane, $x_1 + x_2 + x_3 + (-2) = 0$, to perform the separation is shown. This plane corresponds to a weight vector of (1 1 1) and a bias of −2.

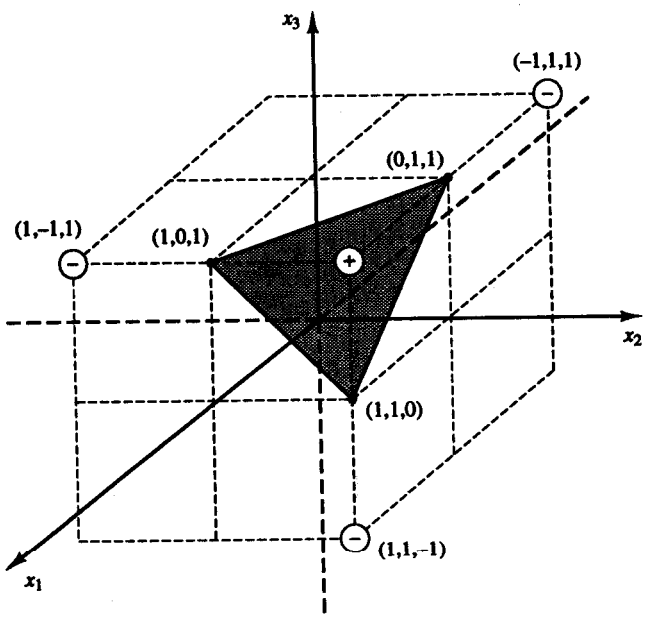

Figure 2.13 Linear separation of bipolar training inputs.

2.3 PERCEPTRON

Perceptrons had perhaps the most far-reaching impact of any of the early neural nets. The perceptron learning rule is a more powerful learning rule than the Hebb rule. Under suitable assumptions, its iterative learning procedure can be proved to converge to the correct weights, i.e., the weights that allow the net to produce the correct output value for each of the training input patterns. Not too surprisingly, one of the necessary assumptions is that such weights exist.

A number of different types of perceptrons are described in Rosenblatt (1962) and in Minsky and Papert (1969, 1988). Although some perceptrons were self-organizing, most were trained. Typically, the original perceptrons had three layers of neurons—sensory units, associator units, and a response unit—forming an approximate model of a retina. One particular simple perceptron [Block, 1962] used binary activations for the sensory and associator units and an activation of $+1$, 0, or -1 for the response unit. The sensory units were connected to the associator units by connections with fixed weights having values of $+1, 0$, or -1, assigned at random.

The activation function for each associator unit was the binary step function with an arbitrary, but fixed, threshold. Thus, the signal sent from the associator units to the output unit was a binary (0 or 1) signal. The output of the perceptron is $y = f(y\_in)$, where the activation function is

$$f(y\_in) = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \leq y\_in \leq \theta \\ -1 & \text{if } y\_in < -\theta \end{cases}$$

The weights from the associator units to the response (or output) unit were adjusted by the perceptron learning rule. For each training input, the net would calculate the response of the output unit. Then the net would determine whether an error occurred for this pattern (by comparing the calculated output with the target value). The net did not distinguish between an error in which the calculated output was zero and the target -1, as opposed to an error in which the calculated output was $+1$ and the target -1. In either of these cases, the sign of the error denotes that the weights should be changed in the direction indicated by the target value. However, only the weights on the connections from units that sent a non-zero signal to the output unit would be adjusted (since only these signals contributed to the error). If an error occurred for a particular training input pattern, the weights would be changed according to the formula

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i,$$

where the target value t is $+1$ or -1 and α is the learning rate. If an error did not occur, the weights would not be changed.

Training would continue until no error occurred. The perceptron learning rule convergence theorem states that if weights exist to allow the net to respond correctly to all training patterns, then the rule's procedure for adjusting the weights will find values such that the net does respond correctly to all training patterns (i.e., the net solves the problem or learns the classification). Moreover, the net will find these weights in a finite number of training steps. We will consider a proof of this theorem in Section 2.3.4, since it helps clarify which aspects of the many variations on perceptron learning are significant.

2.3.1 Architecture

Simple perceptron for pattern classification

The output from the associator units in the original simple perceptron was a binary vector; that vector is treated as the input signal to the output unit in the sections that follow. As the proof of the perceptron learning rule convergence theorem given in Section 2.3.4 illustrates, the assumption of a binary input is not necessary. Since only the weights from the associator units to the output unit could be adjusted, we limit our consideration to the single-layer portion of the net, shown in Figure 2.14. Thus, the associator units function like input units, and the architecture is as given in the figure.

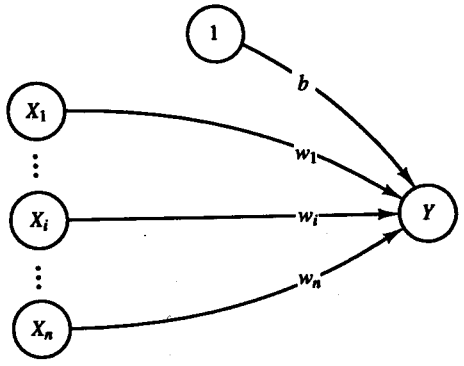

Figure 2.14 Perceptron to perform single classification.

The goal of the net is to classify each input pattern as belonging, or not belonging, to a particular class. Belonging is signified by the output unit giving a response of $+1$; not belonging is indicated by a response of -1. The net is trained to perform this classification by the iterative technique described earlier and given in the algorithm that follows.

2.3.2 Algorithm

The algorithm given here is suitable for either binary or bipolar input vectors (n-tuples), with a bipolar target, fixed θ, and adjustable bias. The threshold θ does not play the same role as in the step function illustrated in Section 2.1.2; thus, both a bias and a threshold are needed. The role of the threshold is discussed following the presentation of the algorithm. The algorithm is not particularly sensitive to the initial values of the weights or the value of the learning rate.

Step 0. Initialize weights and bias.
(For simplicity, set weights and bias to zero.)
Set learning rate α ($0 < \alpha \leq 1$).
(For simplicity, α can be set to 1.)

Step 1. While stopping condition is false, do Steps 2–6.

Step 2. For each training pair s:t, do Steps 3–5.

Step 3. Set activations of input units:

$$x_i = s_i.$$

Step 4. Compute response of output unit:

$$y\_in = b + \sum_i x_i w_i;$$

$$y = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \leq y\_in \leq \theta \\ -1 & \text{if } y\_in < -\theta \end{cases}$$

Step 5. Update weights and bias if an error occurred for this pattern.

If $y \neq t$,

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i,$$

$$b(\text{new}) = b(\text{old}) + \alpha t.$$

else

$$w_i(\text{new}) = w_i(\text{old}),$$

$$b(\text{new}) = b(\text{old}).$$

Step 6. Test stopping condition:
If no weights changed in Step 2, stop; else, continue.

Note that only weights connecting active input units ($x_i \neq 0$) are updated. Also, weights are updated only for patterns that do not produce the correct value of y. This means that as more training patterns produce the correct response, less learning occurs. This is in contrast to the training of the ADALINE units described in Section 2.4, in which learning is based on the difference between y_in and t.

The threshold on the activation function for the response unit is a fixed, non-negative value θ. The form of the activation function for the output unit (response unit) is such that there is an "undecided" band (of fixed width determined by θ) separating the region of positive response from that of negative response. Thus, the previous analysis of the interchangeability of bias and threshold does not apply, because changing θ would change the width of the band, not just the position.

Note that instead of one separating line, we have a line separating the region of positive response from the region of zero response, namely, the line bounding the inequality

$$w_1x_1 + w_2x_2 + b > \theta,$$

and a line separating the region of zero response from the region of negative response, namely, the line bounding the inequality

$$w_1x_1 + w_2x_2 + b < -\theta.$$

2.3.3 Application

Logic functions

Example 2.11 A Perceptron for the AND function: binary inputs, bipolar targets

Let us consider again the AND function with binary input and bipolar target, now using the perceptron learning rule. The training data are as given in Example 2.6 for the Hebb rule. An adjustable bias is included, since it is necessary if a single-layer net is to be able to solve this problem. For simplicity, we take $\alpha = 1$ and set the initial weights and bias to 0, as indicated. However, to illustrate the role of the threshold, we take $\theta = .2$.

The weight change is $\Delta \mathbf{w} = t(x_1, x_2, 1)$ if an error has occurred and zero otherwise. Presenting the first input, we have:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	$1)$							$(w_1$	w_2	$b)$
									(0	0	0)
(1	1	1)	0	0	1	(1	1	1)	(1	1	1)

The separating lines become

$$x_1 + x_2 + 1 = .2$$

and

$$x_1 + x_2 + 1 = -.2.$$

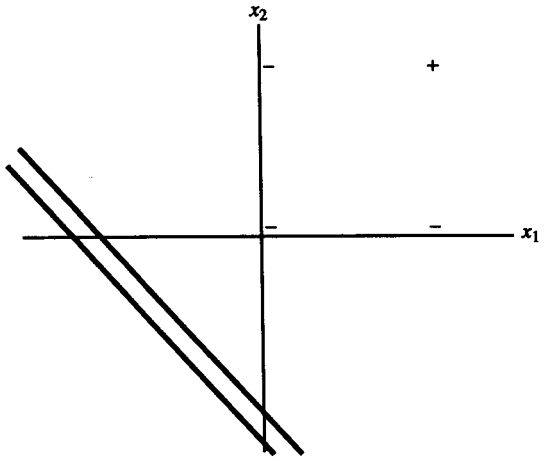

Figure 2.15 Decision boundary for logic function AND after first training input.

The graph in Figure 2.15 shows that the response of the net will now be correct for the first input pattern.

Presenting the second input yields the following:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(1	1	1)
(1	0	1)	2	1	−1	(−1	0	−1)	(0	①	0)

The separating lines become

$$x_2 = .2$$

and

$$x_2 = -.2$$

The graph in Figure 2.16 shows that the response of the net will now (still) be correct for the first input point.

For the third input, we have:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(0	1	0)
(0	1	1)	1	1	−1	(0	−1	−1)	(0	0	−1)

Since the components of the input patterns are nonnegative and the components of the weight vector are nonpositive, the response of the net will be negative (or zero).

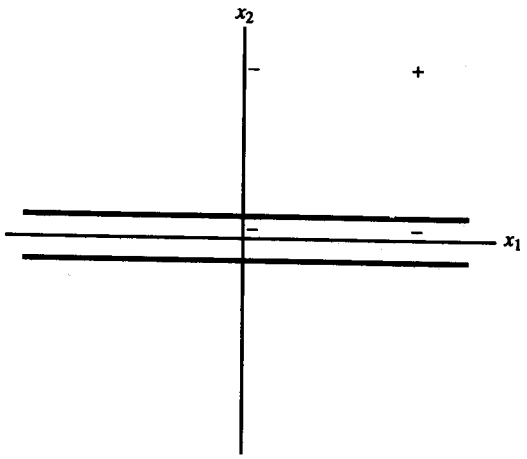

Figure 2.16 Decision boundary after second training input.

To complete the first epoch of training, we present the fourth training pattern:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(0	0	−1)
(0	0	1)	−1	−1	−1	(0	0	−1)	(0	0	−1)

The response for all of the input patterns is negative for the weights derived; but since the response for input pattern (1, 1) is not correct, we are not finished.

The second epoch of training yields the following weight updates for the first input:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(0	0	−1)
(1	1	1)	−1	−1	1	(1	1	1)	(1	1	0)

The separating lines become

$$x_1 + x_2 = .2$$

and

$$x_1 + x_2 = -.2.$$

The graph in Figure 2.17 shows that the response of the net will now be correct for (1, 1).

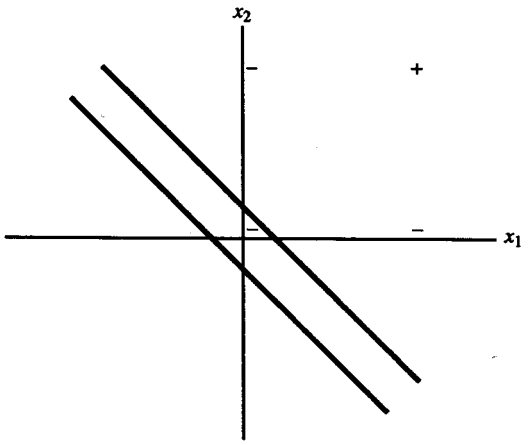

Figure 2.17 Boundary after first training input of second epoch.

For the second input in the second epoch, we have:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
(x_1	x_2	1)							(w_1	w_2	b)
									(1	1	0)
(1	0	1)	1	1	−1	(−1	0	−1)	(0	1	−1)

The separating lines become

$$x_2 - 1 = .2$$

and

$$x_2 - 1 = -.2.$$

The graph in Figure 2.18 shows that the response of the net will now be correct (negative) for the input points (1, 0) and (0, 0); the response for input points (0, 1) and (1, 1) will be 0, since the net input would be 0, which is between −.2 and .2 ($\theta = .2$).

In the second epoch, the third input yields:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
(x_1	x_2	1)							(w_1	w_2	b)
									(0	1	−1)
(0	1	1)	0	0	−1	(0	−1	−1)	(0	0	−2)

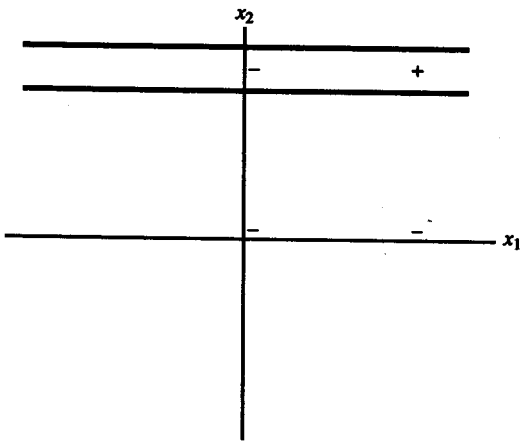

Figure 2.18 Boundary after second input of second epoch.

Again, the response will be negative for all of the input.

To complete the second epoch of training, we present the fourth training pattern:

INPUT			NET	OUT	TARGET	WEIGHTS CHANGE			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(0	0	−2)
(0	0	1)	−2	−1	−1	(0	0	0)	(0	0	−2)

The results for the third epoch are:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	1)							$(w_1$	w_2	$b)$
									(0	0	−2)
(1	1	1)	−2	−1	1	(1	1	1)	(1	1	−1)
(1	0	1)	0	0	−1	(−1	0	−1)	(0	1	−2)
(0	1	1)	−1	−1	−1	(0	0	0)	(0	1	−2)
(0	0	1)	−2	−1	−1	(0	0	0)	(0	1	−2)

The results for the fourth epoch are:

(1	1	1)	−1	−1	1	(1	1	1)	(1	2	−1)
(1	0	1)	0	0	−1	(−1	0	−1)	(0	2	−2)
(0	1	1)	0	0	−1	(0	−1	−1)	(0	1	−3)
(0	0	1)	−3	−1	−1	(0	0	0)	(0	1	−3)

For the fifth epoch, we have

(1	1	1)	−2	−1	1	(1	1	1)	(1	2	−2)
(1	0	1)	−1	−1	−1	(0	0	0)	(1	2	−2)
(0	1	1)	0	0	−1	(0	−1	−1)	(1	1	−3)
(0	0	1)	−3	−1	−1	(0	0	0)	(1	1	−3)

and for the sixth epoch,

(1	1	1)	−1	−1	1	(1	1	1)	(2	2	−2)
(1	0	1)	0	0	−1	(−1	0	−1)	(1	2	−3)
(0	1	1)	−1	−1	−1	(0	0	0)	(1	2	−3)
(0	0	1)	−3	−1	−1	(0	0	0)	(1	2	−3)

The results for the seventh epoch are:

(1	1	1)	0	0	1	(1	1	1)	(2	3	−2)
(1	0	1)	0	0	−1	(−1	0	−1)	(1	3	−3)
(0	1	1)	0	0	−1	(0	−1	−1)	(1	2	−4)
(0	0	1)	−4	−1	−1	(0	0	0)	(1	2	−4)

The eighth epoch yields

(1	1	1)	−1	−1	1	(1	1	1)	(2	3	−3)
(1	0	1)	−1	−1	−1	(0	0	0)	(2	3	−3)
(0	1	1)	0	0	−1	(0	−1	−1)	(2	2	−4)
(0	0	1)	−4	−1	−1	(0	0	0)	(2	2	−4)

and the ninth

(1	1	1)	0	0	1	(1	1	1)	(3	3	−3)
(1	0	1)	0	0	−1	(−1	0	−1)	(2	3	−4)
(0	1	1)	−1	−1	−1	(0	0	0)	(2	3	−4)
(0	0	1)	−4	−1	−1	(0	0	0)	(2	3	−4)

Finally, the results for the tenth epoch are:

(1	1	1)	1	1	1	(0	0	0)	(2	3	−4)
(1	0	1)	−2	−1	−1	(0	0	0)	(2	3	−4)
(0	1	1)	−1	−1	−1	(0	0	0)	(2	3	−4)
(0	0	1)	−4	−1	−1	(0	0	0)	(2	3	−4)

Thus, the positive response is given by all points such that

$$2x_1 + 3x_2 - 4 > .2,$$

with boundary line

$$x_2 = -\frac{2}{3}x_1 + \frac{7}{5},$$

and the negative response is given by all points such that

$$2x_1 + 3x_2 - 4 < -.2,$$

with boundary line

$$x_2 = -\frac{2}{3}x_1 + \frac{19}{15}$$

(see Figure 2.19.)

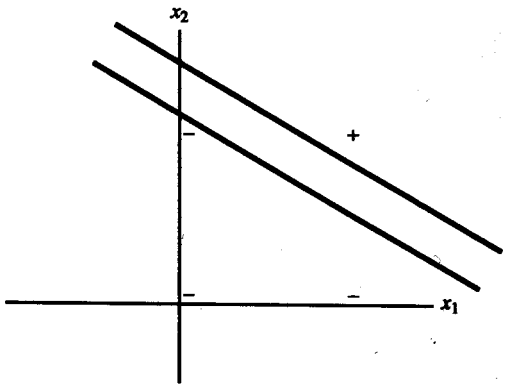

Figure 2.19 Final decision boundaries for AND function in perceptron learning.

Since the proof of the perceptron learning rule convergence theorem (Section 2.3.4) shows that binary input is not required, and in previous examples bipolar input was often preferable, we consider again the previous example, but with bipolar inputs, an adjustable bias, and $\theta = 0$. This variation provides the most direct comparison with Widrow-Hoff learning (an ADALINE net), which we consider in the next section. Note that it is not necessary to modify the training set so that all patterns are mapped to $+1$ (as is done in the proof of the perceptron learning rule convergence theorem); the weight adjustment is $t\mathbf{x}$ whenever the response of the net to input vector $\mathbf{x}$ is incorrect. The target value is still bipolar.

Example 2.12 A Perceptron for the AND function: bipolar inputs and targets

The training process for bipolar input, $\alpha = 1$, and threshold and initial weights = 0 is:

INPUT			NET	OUT	TARGET	WEIGHT CHANGES			WEIGHTS		
(x_1	x_2	1)							(w_1	w_2	b)
									(0	0	0)
(1	1	1)	0	0	1	(1	1	1)	(1	1	1)
(1	−1	1)	1	1	−1	(−1	1	−1)	(0	2	0)
(−1	1	1)	2	1	−1	(1	−1	−1)	(1	1	−1)
(−1	−1	1)	−3	−1	−1	(0	0	0)	(1	1	−1)

In the second epoch of training, we have:

(1	1	1)	1	1	1	(0	0	0)	(1	1	−1)
(1	−1	1)	−1	−1	−1	(0	0	0)	(1	1	−1)
(−1	1	1)	−1	−1	−1	(0	0	0)	(1	1	−1)
(−1	−1	1)	−3	−1	−1	(0	0	0)	(1	1	−1)

Since all the Δw's are 0 in epoch 2, the system was fully trained after the first epoch.

It seems intuitively obvious that a procedure that could continue to learn to improve its weights even after the classifications are all correct would be better than a learning rule in which weight updates cease as soon as all training patterns are classified correctly. However, the foregoing example shows that the change from binary to bipolar representation improves the results rather spectacularly.

We next show that the perceptron with $\alpha = 1$ and $\theta = .1$ can solve the problem the Hebb rule could not.

Other simple examples

Example 2.13 Perceptron training is more powerful than Hebb rule training

The mapping of interest maps the first three components of the input vector onto a target value that is 1 if there are no zero inputs and that is −1 if there is one zero input. (If there are two or three zeros in the input, we do not specify the target value.) This is a portion of the parity problem for three inputs. The fourth component of the input vector is the input to the bias weight and is therefore always 1. The weight change vector is left blank if no error has occurred for a particular pattern. The learning rate is $\alpha = 1$, and the threshold $\theta = .1$. We show the following selected epochs:

INPUT				NET	OUT	TARGET	WEIGHT CHANGE				WEIGHTS			
x_1	x_2	x_3	1								(w_1	w_2	w_3	b)
											(0	0	0	0)
Epoch 1:														
(1	1	1	1)	0	0	1	(1	1	1	1)	(1	1	1	1)
(1	1	0	1)	3	1	−1	(−1	−1	0	−1)	(0	0	1	0)
(1	0	1	1)	1	1	−1	(−1	0	−1	−1)	(−1	0	0	−1)
(0	1	1	1)	−1	−1	−1	(			)	(−1	0	0	−1)
Epoch 2:														
(1	1	1	1)	−2	−1	1	(1	1	1	1)	(0	1	1	0)
(1	1	0	1)	1	1	−1	(−1	−1	0	−1)	(−1	0	1	−1)
(1	0	1	1)	−1	−1	−1	(			)	(−1	0	1	−1)
(0	1	1	1)	0	0	−1	(0	−1	−1	−1)	(−1	−1	0	−2)
Epoch 3:														
(1	1	1	1)	−4	−1	1	(1	1	1	1)	(0	0	1	−1)
(1	1	0	1)	−1	−1	−1	(			)	(0	0	1	−1)
(1	0	1	1)	0	0	−1	(−1	0	−1	−1)	(−1	0	0	−2)
(0	1	1	1)	−2	−1	−1	(			)	(−1	0	0	−2)
Epoch 4:														
(1	1	1	1)	−3	−1	1	(1	1	1	1)	(0	1	1	−1)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(−1	0	1	−2)
(1	0	1	1)	−2	−1	−1	(			)	(−1	0	1	−2)
(0	1	1	1)	−1	−1	−1	(			)	(−1	0	1	−2)
Epoch 5:														
(1	1	1	1)	−2	−1	1	(1	1	1	1)	(0	1	2	−1)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(−1	0	2	−2)
(1	0	1	1)	−1	−1	−1	(			)	(−1	0	2	−2)
(0	1	1	1)	0	0	−1	(0	−1	−1	−1)	(−1	−1	1	−3)
Epoch 10:														
(1	1	1	1)	−3	−1	1	(1	1	1	1)	(1	1	2	−3)
(1	1	0	1)	−1	−1	−1	(			)	(1	1	2	−3)
(1	0	1	1)	0	0	−1	(−1	0	−1	−1)	(0	1	1	−4)
(0	1	1	1)	−2	−1	−1	(			)	(0	1	1	−4)

Epoch 15:

(1	1	1	1)	−1	−1	1	(1	1	1	1)	(1	3	3	−4)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(0	2	3	−5)
(1	0	1	1)	−2	−1	−1	(			)	(0	2	3	−5)
(0	1	1	1)	0	0	−1	(0	−1	−1	−1)	(0	1	2	−6)

Epoch 20:

(1	1	1	1)	−2	−1	1	(1	1	1	1)	(2	2	4	−6)
(1	1	0	1)	−2	−1	−1	(			)	(2	2	4	−6)
(1	0	1	1)	0	0	−1	(−1	0	−1	−1)	(1	2	3	−7)
(0	1	1	1)	−2	−1	−1	(			)	(1	2	3	−7)

Epoch 25:

(1	1	1	1)	0	0	1	(1	1	1	1)	(3	4	4	−7)
(1	1	0	1)	0	0	−1	(−1	−1	0	−1)	(2	3	4	−8)
(1	0	1	1)	−2	−1	−1	(			)	(2	3	4	−8)
(0	1	1	1)	−1	−1	−1	(			)	(2	3	4	−8)

Epoch 26:

(1	1	1	1)	1	1	1	(			)	(2	3	4	−8)
(1	1	0	1)	−3	−1	−1	(			)	(2	3	4	−8)
(1	0	1	1)	−2	−1	−1	(			)	(2	3	4	−8)
(0	1	1	1)	−1	−1	−1	(			)	(2	3	4	−8)

Character recognition

Application Procedure

Step 0. Apply training algorithm to set the weights.

Step 1. For each input vector **x** to be classified, do Steps 2–3.

Step 2. Set activations of input units.

Step 3. Compute response of output unit:

$$y\_in = \sum_i x_i w_i;$$

$$y = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \leq y\_in \leq \theta \\ -1 & \text{if } y\_in < -\theta \end{cases}$$

Example 2.14 A Perceptron to classify letters from different fonts: one output class

As the first example of using the perceptron for character recognition, consider the 21 input patterns in Figure 2.20 as examples of *A* or not-*A*. In other words, we train the perceptron to classify each of these vectors as belonging, or not belonging, to

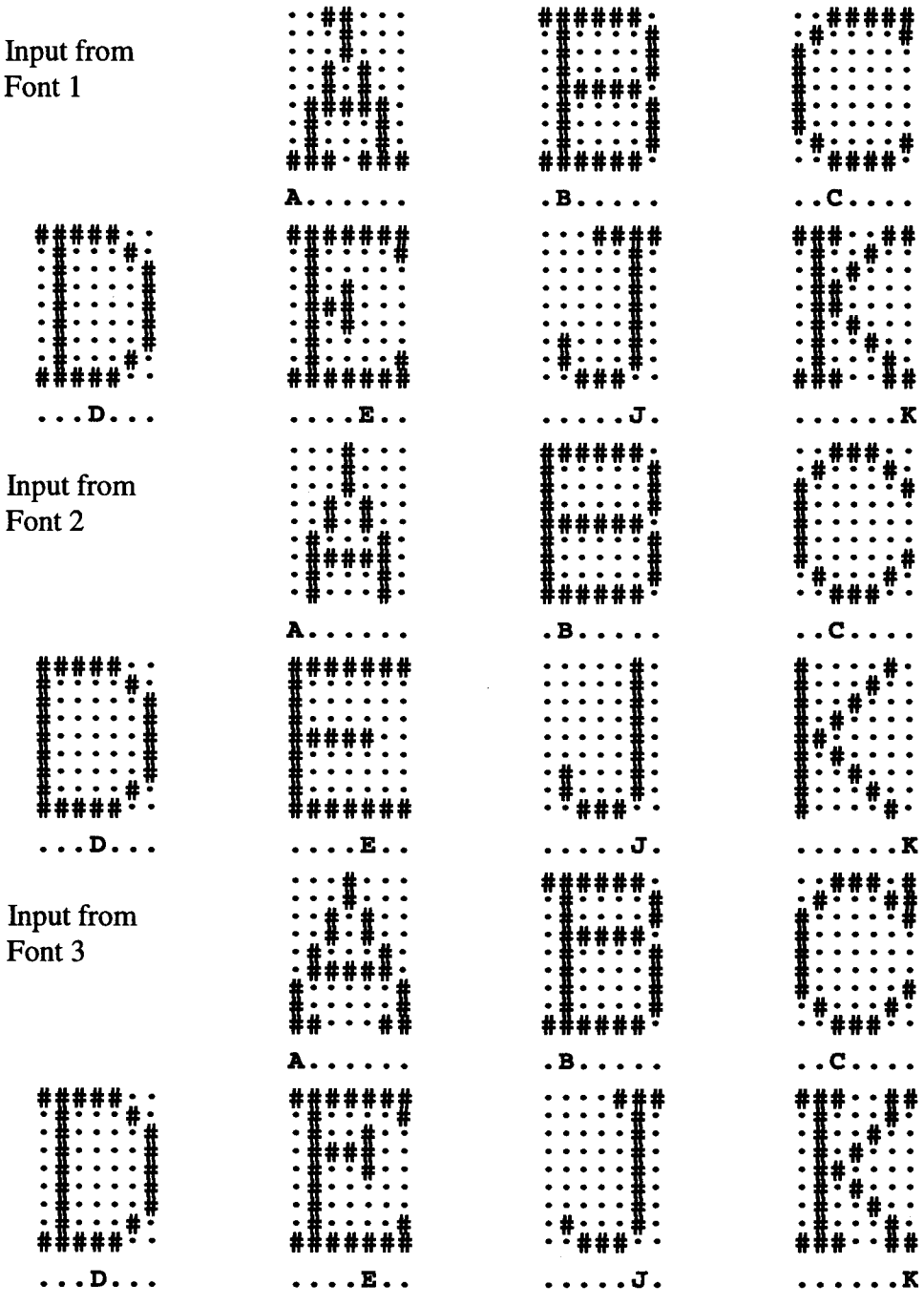

Figure 2.20 Training input and target output patterns.

the class A. In that case, the target value for each pattern is either 1 or -1; only the first component of the target vector shown is applicable. The net is as shown in Figure 2.14, and $n = 63$. There are three examples of A and 18 examples of not-A in Figure 2.20.

We could, of course, use the same vectors as examples of B or not-B and train the net in a similar manner. Note, however, that because we are using a single-layer net, the weights for the output unit signifying A do not have any interaction with the weights for the output unit signifying B. Therefore, we can solve these two problems at the same time, by allowing a column of weights for each output unit. Our net would have 63 input units and 2 output units. The first output unit would correspond to "A or not-A", the second unit to "B or not-B." Continuing this idea, we can identify 7 output units, one for each of the 7 categories into which we wish to classify our input.

Ideally, when an unknown character is presented to the net, the net's output consists of a single "yes" and six "nos." In practice, that may not happen, but the net may produce several guesses that can be resolved by other methods, such as considering the strengths of the activations of the various output units prior to setting the threshold or examining the context in which the ill-classified character occurs.

Example 2.15 A Perceptron to classify letters from different fonts: several output classes

The perceptron shown in Figure 2.14 can be extended easily to the case where the input vectors belong to one (or more) of several categories. In this type of application, there is an output unit representing each of the categories to which the input vectors may belong. The architecture of such a net is shown in Figure 2.21.

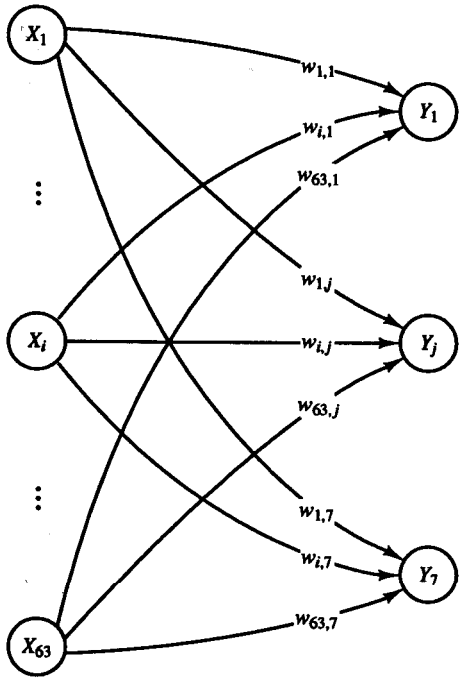

Figure 2.21 Perceptron to classify input into seven categories.

For this example, each input vector is a 63-tuple representing a letter expressed as a pattern on a 7 × 9 grid of pixels. The training patterns are illustrated in Figure 2.20. There are seven categories to which each input vector may belong, so there are seven components to the output vector, each representing a letter: A, B, C, D, E, K, or J. For ease of reading, we show the target output pattern indicating that the input was an "A" as (A · · · · · · ·), a "B" (· B · · · · · ·), etc.

The training input patterns and target responses must be converted to an appropriate form for the neural net to process. A bipolar representation has better computational characteristics than does a binary representation. The input patterns may be converted to bipolar vectors as described in Example 2.8; the target output pattern (A · · · · · · ·) becomes the bipolar vector (1, −1, −1, −1, −1, −1, −1) and the target pattern (· B · · · · · ·) is represented by the bipolar vector (−1, 1, −1, −1, −1, −1, −1).

A modified training algorithm for several output categories (threshold = 0, learning rate = 1, bipolar training pairs) is as follows:

Step 0. Initialize weights and biases
(0 or small random values).

Step 1. While stopping condition is false, do Steps 1–6.

Step 2. For each bipolar training pair **s : t**, do Steps 3–5.

Step 3. Set activation of each input unit, $i = 1, \ldots, n$:

$$x_i = s_i.$$

Step 4. Compute activation of each output unit, $j = 1, \ldots, m$:

$$y\_in_j = b_j + \sum_i x_i w_{ij}.$$

$$y_j = \begin{cases} 1 & \text{if } y\_in_j > \theta \\ 0 & \text{if } -\theta \le y\_in_j \le \theta \\ -1 & \text{if } y\_in_j < -\theta \end{cases}$$

Step 5. Update biases and weights, $j = 1, \ldots, m$; $i = 1, \ldots, n$:
If $t_j \neq y_j$, then

$$b_j(\text{new}) = b_j(\text{old}) + t_j;$$

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + t_j x_i.$$

Else, biases and weights remain unchanged.

Step 6. Test for stopping condition:
If no weight changes occurred in Step 2, stop; otherwise, continue.

After training, the net correctly classifies each of the training vectors.

The performance of the net shown in Figure 2.21 in classifying input vectors that are similar to the training vectors is shown in Figure 2.22. Each of the input

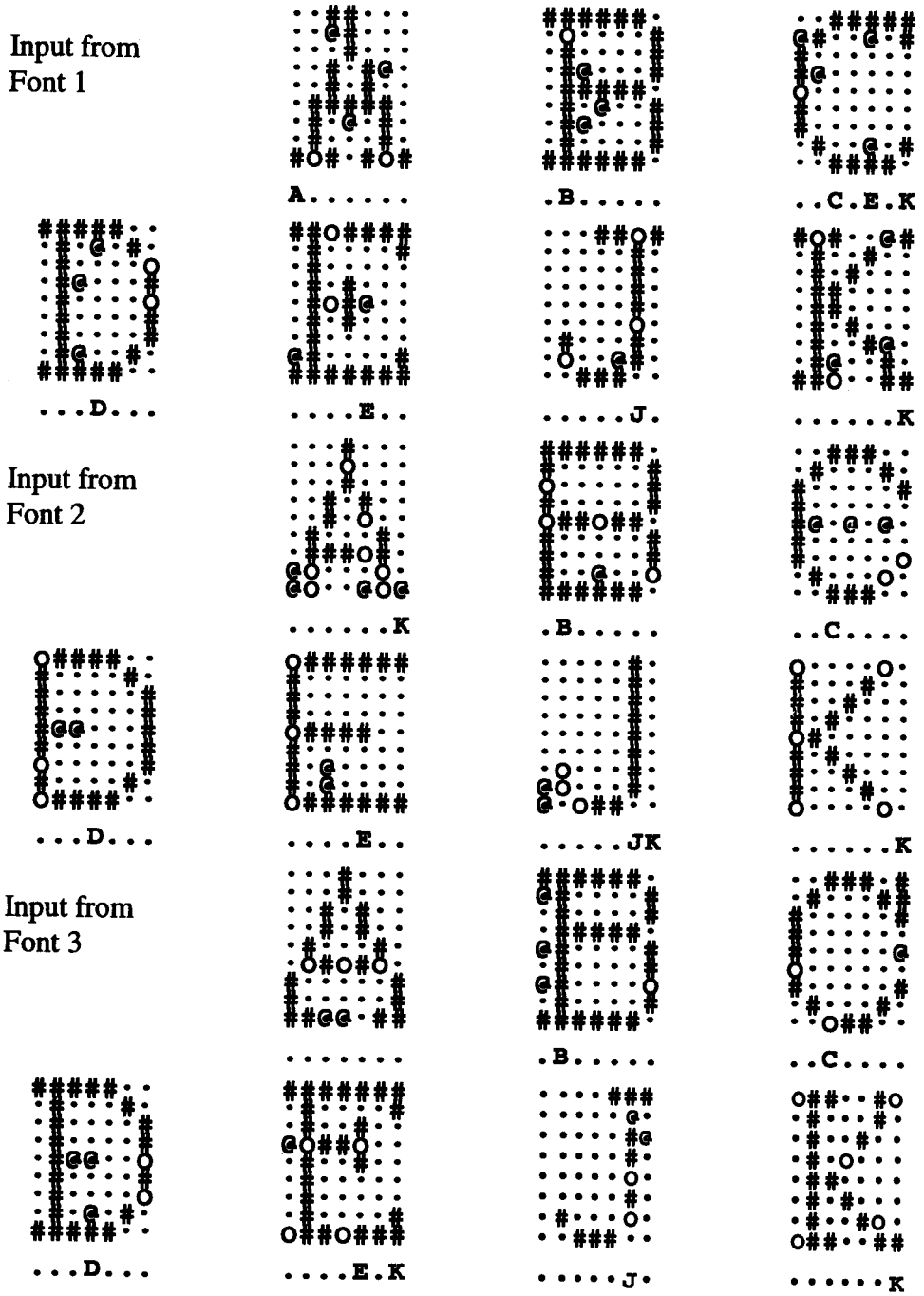

Figure 2.22 Classification of noisy input patterns using a perceptron.

patterns is a training input pattern with a few of its pixels changed. The pixels where the input pattern differs from the training pattern are indicated by @ for a pixel that is "on" now but was "off" in the training pattern, and O for a pixel that is "off" now but was originally "on."

2.3.4 Perceptron Learning Rule Convergence Theorem

The statement and proof of the perceptron learning rule convergence theorem given here are similar to those presented in several other sources [Hertz, Krogh, & Palmer, 1991; Minsky & Papert, 1988; Arbib, 1987]. Each of these provides a slightly different perspective and insights into the essential aspects of the rule. The fact that the weight vector is perpendicular to the plane separating the input patterns at each step of the learning processes [Hertz, Krogh, & Palmer, 1991] can be used to interpret the degree of difficulty of training a perceptron for different types of input.

The perceptron learning rule is as follows:

Given a finite set of P input training vectors

$$\mathbf{x}(p), \quad p = 1, \ldots, P,$$

each with an associated target value

$$t(p), \quad p = 1, \ldots, P,$$

which is either $+1$ or -1, and an activation function $y = f(y\_in)$, where

$$y = \begin{cases} 1 & \text{if } y\_in > \theta \\ 0 & \text{if } -\theta \leq y\_in \leq \theta \\ -1 & \text{if } y\_in < -\theta, \end{cases}$$

the weights are updated as follows:

If $y \neq t$, then

$$\mathbf{w}\ (\text{new}) = \mathbf{w}\ (\text{old}) + t\mathbf{x};$$

else

no change in the weights.

The perceptron learning rule convergence theorem is:

If there is a weight vector $\mathbf{w}^*$ such that $f(\mathbf{x}(p) \cdot \mathbf{w}^*) = t(p)$ for all p, then for any starting vector $\mathbf{w}$, the perceptron learning rule will converge to a weight vector (not necessarily unique and not necessarily $\mathbf{w}^*$) that gives the correct response for all training patterns, and it will do so in a finite number of steps.

The proof of the theorem is simplified by the observation that the training set can be considered to consist of two parts:

$$F^+ = \{\mathbf{x} \text{ such that the target value is } +1\}$$

and

$$F^- = \{\mathbf{x} \text{ such that the target value is } -1\}.$$

A new training set is then defined as

$$F = F^+ \cup -F^-,$$

where

$$-F^- = \{-\mathbf{x} \text{ such that } \mathbf{x} \text{ is in } F^-\}.$$

In order to simplify the algebra slightly, we shall assume, without loss of generality, that $\theta = 0$ and $\alpha = 1$ in the proof. The existence of a solution of the original problem, namely the existence of a weight vector $\mathbf{w}^*$ for which

$$\mathbf{x}\cdot\mathbf{w}^* > 0 \qquad \text{if } \mathbf{x} \text{ is in } F^+$$

and

$$\mathbf{x}\cdot\mathbf{w}^* < 0 \qquad \text{if } \mathbf{x} \text{ is in } F^-,$$

is equivalent to the existence of a weight vector $\mathbf{w}^*$ for which

$$\mathbf{x}\cdot\mathbf{w}^* > 0 \qquad \text{if } \mathbf{x} \text{ is in } F.$$

All target values for the modified training set are $+1$. If the response of the net is incorrect for a given training input, the weights are updated according to

$$\mathbf{w}(\text{new}) = \mathbf{w}(\text{old}) + \mathbf{x}.$$

Note that the input training vectors must each have an additional component (which is always 1) included to account for the signal to the bias weight.

We now sketch the proof of this remarkable convergence theorem, because of the light that it sheds on the wide variety of forms of perceptron learning that are guaranteed to converge. As mentioned, we assume that the training set has been modified so that all targets are $+1$. Note that this will involve reversing the sign of all components (including the input component corresponding to the bias) for any input vectors for which the target was originally -1.

We now consider the sequence of input training vectors for which a weight change occurs. We must show that this sequence is finite.

Let the starting weights be denoted by $\mathbf{w}(0)$, the first new weights by $\mathbf{w}(1)$, etc. If $\mathbf{x}(0)$ is the first training vector for which an error has occurred, then

$$\mathbf{w}(1) = \mathbf{w}(0) + \mathbf{x}(0) \qquad (\text{where, by assumption, } \mathbf{x}(0)\cdot\mathbf{w}(0) \leq 0).$$

If another error occurs, we denote the vector $\mathbf{x}(1)$; $\mathbf{x}(1)$ may be the same as $\mathbf{x}(0)$ if no errors have occurred for any other training vectors, or $\mathbf{x}(1)$ may be different from $\mathbf{x}(0)$. In either case,

$$\mathbf{w}(2) = \mathbf{w}(1) + \mathbf{x}(1) \qquad (\text{where, by assumption, } \mathbf{x}(1)\cdot\mathbf{w}(1) \leq 0).$$

At any stage, say, k, of the process, the weights are changed if and only if the current weights fail to produce the correct (positive) response for the current input vector, i.e., if $\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) \leq 0$. Combining the successive weight changes gives

$$\mathbf{w}(k) = \mathbf{w}(0) + \mathbf{x}(0) + \mathbf{x}(1) + \mathbf{x}(2) + \cdots + \mathbf{x}(k-1).$$

We now show that k cannot be arbitrarily large.

Let $\mathbf{w}^*$ be a weight vector such that $\mathbf{x}\cdot\mathbf{w}^* > 0$ for all training vectors in F. Let $m = \min\{\mathbf{x}\cdot\mathbf{w}^*\}$, where the minimum is taken over all training vectors in F; this minimum exists as long as there are only finitely many training vectors. Now,

$$\begin{aligned}\mathbf{w}(k)\cdot\mathbf{w}^* &= [\mathbf{w}(0) + \mathbf{x}(0) + \mathbf{x}(1) + \mathbf{x}(2) + \cdots + \mathbf{x}(k-1)]\cdot\mathbf{w}^* \\ &\geq \mathbf{w}(0)\cdot\mathbf{w}^* + km\end{aligned}$$

since $\mathbf{x}(i)\cdot\mathbf{w}^* \geq m$ for each i, $1 \leq i \leq P$.

The Cauchy-Schwartz inequality states that for any vectors $\mathbf{a}$ and $\mathbf{b}$,

$$(\mathbf{a}\cdot\mathbf{b})^2 \leq \|\mathbf{a}\|^2 \|\mathbf{b}\|^2,$$

or

$$\|\mathbf{a}\|^2 \geq \frac{(\mathbf{a}\cdot\mathbf{b})^2}{\|\mathbf{b}\|^2} \qquad (\text{for } \|\mathbf{b}\|^2 \neq 0).$$

Therefore,

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &\geq \frac{(\mathbf{w}(k)\cdot\mathbf{w}^*)^2}{\|\mathbf{w}^*\|^2} \\ &\geq \frac{(\mathbf{w}(0)\cdot\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2}.\end{aligned}$$

This shows that the squared length of the weight vector grows faster than k^2, where k is the number of time the weights have changed.

However, to show that the length cannot continue to grow indefinitely, consider

$$\mathbf{w}(k) = \mathbf{w}(k-1) + \mathbf{x}(k-1),$$

together with the fact that

$$\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) \leq 0.$$

By simple algebra,

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &= \|\mathbf{w}(k-1)\|^2 + 2\mathbf{x}(k-1)\cdot\mathbf{w}(k-1) + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(k-1)\|^2 + \|\mathbf{x}(k-1)\|^2.\end{aligned}$$

Now let $M = \max \{\| \mathbf{x} \|^2 \text{ for all } \mathbf{x} \text{ in the training set}\}$; then

$$\begin{aligned}\|\mathbf{w}(k)\|^2 &\leq \|\mathbf{w}(k-1)\|^2 + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(k-2)\|^2 + \|\mathbf{x}(k-2)\|^2 + \|\mathbf{x}(k-1)\|^2 \\ &\vdots \\ &\leq \|\mathbf{w}(0)\|^2 + \|\mathbf{x}(0)\|^2 + \cdots + \|\mathbf{x}(k-1)\|^2 \\ &\leq \|\mathbf{w}(0)\|^2 + kM.\end{aligned}$$

Thus, the squared length grows less rapidly than linearly in k.

Combining the inequalities

$$\|\mathbf{w}(k)\|^2 \geq \frac{(\mathbf{w}(0)\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2}$$

and

$$\|\mathbf{w}(k)\|^2 \leq \|\mathbf{w}(0)\|^2 + kM$$

shows that the number of times that the weights may change is bounded. Specifically,

$$\frac{(\mathbf{w}(0)\cdot\mathbf{w}^* + km)^2}{\|\mathbf{w}^*\|^2} \leq \|\mathbf{w}(k)\|^2 \leq \|\mathbf{w}(0)\|^2 + kM.$$

Again, to simplify the algebra, assume (without loss of generality) that $\mathbf{w}(0) = 0$. Then the maximum possible number of times the weights may change is given by

$$\frac{(km)^2}{\|\mathbf{w}^*\|^2} \leq kM,$$

or

$$k \leq \frac{M\,\|\mathbf{w}^*\|^2}{m^2}.$$

Since the assumption that $\mathbf{w}^*$ exists can be restated, without loss of generality, as the assumption that there is a solution weight vector of unit length (and the definition of m is modified accordingly), the maximum number of weight updates is M/m^2. Note, however, that many more computations may be required, since very few input vectors may generate an error during any one epoch of training. Also, since $\mathbf{w}^*$ is unknown (and therefore, so is m), the number of weight updates cannot be predicted from the preceding inequality.

The foregoing proof shows that many variations in the perceptron learning rule are possible. Several of these variations are explicitly mentioned in Chapter 11 of Minsky and Papert (1988).

The original restriction that the coefficients of the patterns be binary is un-

necessary. All that is required is that there be a finite maximum norm of the training vectors (or at least a finite upper bound to the norm). Training may take a long time (a large number of steps) if there are training vectors that are very small in norm, since this would cause small m to have a small value. The argument of the proof is unchanged if a nonzero value of θ is used (although changing the value of θ may change a problem from solvable to unsolvable or vice versa). Also, the use of a learning rate other than 1 will not change the basic argument of the proof (see Exercise 2.8). Note that there is no requirement that there can be only finitely many training vectors, as long as the norm of the training vectors is bounded (and bounded away from 0 as well). The actual target values do not matter, either; the learning law simply requires that the weights be incremented by the input vector (or a multiple of it) whenever the response of the net is incorrect (and that the training vectors can be stated in such a way that they all should give the same response of the net).

Variations on the learning step include setting the learning rate α to any nonnegative constant (Minsky starts by setting it specifically to 1), setting α to $1/\|\mathbf{x}\|$ so that the weight change is a unit vector, and setting α to $(\mathbf{x}\cdot\mathbf{w})/\|\mathbf{x}\|^2$ (which makes the weight change just enough for the pattern $\mathbf{x}$ to be classified correctly at this step).

Minsky sets the initial weights equal to an arbitrary training pattern. Others usually indicate small random values.

Note also that since the procedure will converge from an arbitrary starting set of weights, the process is error correcting, as long as the errors do not occur too often (and the process is not stopped before error correction occurs).

2.4 ADALINE

The ADALINE (ADaptive Linear NEuron) [Widrow & Hoff, 1960] typically uses bipolar (1 or -1) activations for its input signals and its target output (although it is not restricted to such values). The weights on the connections from the input units to the ADALINE are adjustable. The ADALINE also has a bias, which acts like an adjustable weight on a connection from a unit whose activation is always 1.

In general, an ADALINE can be trained using the delta rule, also known as the least mean squares (LMS) or Widrow-Hoff rule. The rule (Section 2.4.2) can also be used for single-layer nets with several output units; an ADALINE is a special case in which there is only one output unit. During training, the activation of the unit is its net input, i.e., the activation function is the identity function. The learning rule minimizes the mean squared error between the activation and the target value. This allows the net to continue learning on all training patterns, even after the correct output value is generated (if a threshold function is applied) for some patterns.

After training, if the net is being used for pattern classification in which the desired output is either a $+1$ or a -1, a threshold function is applied to the net

input to obtain the activation. If the net input to the ADALINE is greater than or equal to 0, then its activation is set to 1; otherwise it is set to −1. Any problem for which the input patterns corresponding to the output value +1 are linearly separable from input patterns corresponding to the output value −1 can be modeled successfully by an ADALINE unit. An application algorithm is given in Section 2.4.3 to illustrate the use of the activation function after the net is trained.

In Section 2.4.4, we shall see how a heuristic learning rule can be used to train a multilayer combination of ADALINES, known as a MADALINE.

2.4.1 Architecture

An ADALINE is a single unit (neuron) that receives input from several units. It also receives input from a "unit" whose signal is always +1, in order for the bias weight to be trained by the same process (the delta rule) as is used to train the other weights. A single ADALINE is shown in Figure 2.23.

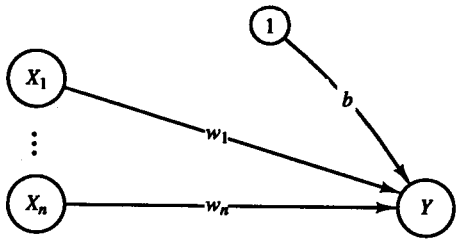

Figure 2.23 Architecture of an ADALINE.

Several ADALINES that receive signals from the same input units can be combined in a single-layer net, as described for the perceptron (Section 2.3.3). If, however, ADALINES are combined so that the output from some of them becomes input for others of them, then the net becomes multilayer, and determining the weights is more difficult. Such a multilayer net, known as a MADALINE, is considered in Section 2.4.5.

2.4.2 Algorithm

A training algorithm for an ADALINE is as follows:

Step 0. Initialize weights.
(Small random values are usually used.)
Set learning rate α.
(See comments following algorithm.)

Step 1. While stopping condition is false, do Steps 2–6.

Step 2. For each bipolar training pair s:t, do Steps 3–5.

Step 3. Set activations of input units, $i = 1, \ldots, n$:

$$x_i = s_i.$$

Step 4. Compute net input to output unit:

$$y\_in = b + \sum_i x_i w_i.$$

Step 5. Update bias and weights, $i = 1, \ldots, n$:

$$b(\text{new}) = b(\text{old}) + \alpha(t - y\_in).$$

$$w_i(\text{new}) = w_i(\text{old}) + \alpha(t - y\_in)x_i.$$

Step 6. Test for stopping condition:
If the largest weight change that occurred in Step 2 is smaller than a specified tolerance, then stop; otherwise continue.

Setting the learning rate to a suitable value requires some care. According to Hecht-Nielsen (1990), an upper bound for its value can be found from the largest eigenvalue of the correlation matrix R of the input (row) vectors $\mathbf{x}(p)$:

$$R = \frac{1}{P} \sum_{p=1}^{P} \mathbf{x}(p)^T \mathbf{x}(p),$$

namely,

$$\alpha < \text{one-half the largest eigenvalue of } R.$$

However, since R does not need to be calculated to compute the weight updates, it is common simply to take a small value for α (such as $\alpha = .1$) initially. If too large a value is chosen, the learning process will not converge; if too small a value is chosen, learning will be extremely slow [Hecht-Nielsen, 1990]. The choice of learning rate and methods of modifying it are considered further in Chapter 6. For a single neuron, a practical range for the learning rate α is $0.1 \leq n\alpha \leq 1.0$, where n is the number of input units [Widrow, Winter & Baxter, 1988].

The proof of the convergence of the ADALINE training process is essentially contained in the derivation of the delta rule, which is given in Section 2.4.4.

2.4.3 Applications

After training, an ADALINE unit can be used to classify input patterns. If the target values are bivalent (binary or bipolar), a step function can be applied as the

activation function for the output unit. The following procedure shows the step function for bipolar targets, the most common case:

Step 0. Initialize weights
(from ADALINE training algorithm given in Section 2.4.2).
Step 1. For each bipolar input vector **x**, do Steps 2–4.
Step 2. Set activations of the input units to **x**.
Step 3. Compute net input to output unit:

$$y\_in = b + \sum_i x_i w_i.$$

Step 4. Apply the activation function:

$$y = \begin{cases} 1 & \text{if } y\_in \geq 0; \\ -1 & \text{if } y\_in < 0. \end{cases}$$

Simple examples

The weights (and biases) in Examples 2.16–2.19 give the minimum total squared error for each set of training patterns. Good approximations to these values can be found using the algorithm in Section 2.4.2 with a small learning rate.

Example 2.16 An ADALINE for the AND function: binary inputs, bipolar targets

Even though the ADALINE was presented originally for bipolar inputs and targets, the delta rule also applies to binary input. In this example, we consider the AND function with binary input and bipolar targets. The function is defined by the following four training patterns:

x_1	x_2	t
1	1	1
1	0	−1
0	1	−1
0	0	−1

As indicated in the derivation of the delta rule (Section 2.4.4), an ADALINE is designed to find weights that minimize the total error

$$E = \sum_{p=1}^{4} (x_1(p)w_1 + x_2(p)w_2 + w_0 - t(p))^2,$$

where

$$x_1(p)w_1 + x_2(p)w_2 + w_0$$

is the net input to the output unit for pattern p and $t(p)$ is the associated target for pattern p.

Weights that minimize this error are

$$w_1 = 1$$

and

$$w_2 = 1,$$

with the bias

$$w_0 = -\frac{3}{2}.$$

Thus, the separating line is

$$x_1 + x_2 - \frac{3}{2} = 0.$$

The total squared error for the four training patterns with these weights is 1.

A minor modification to Example 2.11 (setting $\theta = 0$) shows that for the perceptron, the boundary line is

$$x_2 = -\frac{2}{3}x_1 + \frac{4}{3}.$$

(The two boundary lines coincide when $\theta = 0$.) The total squared error for the minimizing weights found by the perceptron is 10/9.

Example 2.17 An ADALINE for the AND function: bipolar inputs and targets

The weights that minimize the total error for the bipolar form of the AND function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = \frac{1}{2},$$

with the bias

$$w_0 = -\frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 - \frac{1}{2} = 0,$$

which is of course the same line as

$$x_1 + x_2 - 1 = 0,$$

as found by the perceptron in Example 2.12.

Example 2.18 An ADALINE for the AND NOT function: bipolar inputs and targets

The logic function x_1 AND NOT x_2 is defined by the following bipolar input and target patterns:

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	−1
−1	−1	−1

Weights that minimize the total squared error for the bipolar form of the AND NOT function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = -\frac{1}{2},$$

with the bias

$$w_0 = -\frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 - \frac{1}{2}x_2 - \frac{1}{2} = 0.$$

Example 2.19 An ADALINE for the OR function: bipolar inputs and targets

The logic function x_1 OR x_2 is defined by the following bipolar input and target patterns:

x_1	x_2	t
1	1	1
1	−1	1
−1	1	1
−1	−1	−1

Weights that minimize the total squared error for the bipolar form of the OR function are

$$w_1 = \frac{1}{2}$$

and

$$w_2 = \frac{1}{2},$$

with the bias

$$w_0 = \frac{1}{2}.$$

Thus, the separating line is

$$\frac{1}{2}x_1 + \frac{1}{2}x_2 + \frac{1}{2} = 0.$$

2.4.4 Derivations

Delta rule for single output unit

The delta rule changes the weights of the neural connections so as to minimize the difference between the net input to the output unit, *y_in*, and the target value *t*. The aim is to minimize the error over all training patterns. However, this is accomplished by reducing the error for each pattern, one at a time. Weight corrections can also be accumulated over a number of training patterns (so-called batch updating) if desired. In order to distinguish between the fixed (but arbitrary) index for the weight whose adjustment is being determined in the derivation that follows and the index of summation needed in the derivation, we use the index *I* for the weight and the index *i* for the summation. We shall return to the more standard lowercase indices for weights whenever this distinction is not needed. The delta rule for adjusting the *I*th weight (for each pattern) is

$$\Delta w_I = \alpha(t - y\_in)x_I.$$

The nomenclature we use in the derivation is as follows:

x vector of activations of input units, an *n*-tuple.

y_in the net input to output unit *Y* is

$$y\_in = \sum_{i=1}^{n} x_i w_i.$$

t target output.

Derivation. The squared error for a particular training pattern is

$$E = (t - y\_in)^2.$$

E is a function of all of the weights, w_i, $i = 1, \ldots, n$. The gradient of *E* is the vector consisting of the partial derivatives of *E* with respect to each of the weights. The gradient gives the direction of most rapid increase in *E*; the opposite direction

gives the most rapid decrease in the error. The error can be reduced by adjusting the weight w_I in the direction of $-\frac{\partial E}{\partial w_I}$.

Since $y\_in = \sum_{i=1}^{n} x_i w_i$,

$$\frac{\partial E}{\partial w_I} = -2(t - y\_in)\frac{\partial y\_in}{\partial w_I}$$

$$= -2(t - y\_in)x_I.$$

Thus, the local error will be reduced most rapidly (for a given learning rate) by adjusting the weights according to the delta rule,

$$\Delta w_I = \alpha(t - y\_in)x_I.$$

Delta rule for several output units

The derivation given in this subsection allows for more than one output unit. The weights are changed to reduce the difference between the net input to the output unit, y_in_J, and the target value t_J. This formulation reduces the error for each pattern. Weight corrections can also be accumulated over a number of training patterns (so-called batch updating) if desired.

The delta rule for adjusting the weight from the Ith input unit to the Jth output unit (for each pattern) is

$$\Delta w_{IJ} = \alpha(t_J - y\_in_J)x_I.$$

Derivation. The squared error for a particular training pattern is

$$E = \sum_{j=1}^{m} (t_j - y\_in_j)^2.$$

E is a function of all of the weights. The gradient of E is a vector consisting of the partial derivatives of E with respect to each of the weights. This vector gives the direction of most rapid increase in E; the opposite direction gives the direction of most rapid decrease in the error. The error can be reduced most rapidly by adjusting the weight w_{IJ} in the direction of $-\partial E/\partial w_{IJ}$.

We now find an explicit formula for $\partial E/\partial w_{IJ}$ for the arbitrary weight w_{IJ}. First, note that

$$\frac{\partial E}{\partial w_{IJ}} = \frac{\partial}{\partial w_{IJ}} \sum_{j=1}^{m} (t_j - y\_in_j)^2$$

$$= \frac{\partial}{\partial w_{IJ}} (t_J - y\_in_J)^2,$$

since the weight w_{IJ} influences the error only at output unit Y_J. Furthermore, using the fact that

$$y\_in_J = \sum_{i=1}^{n} x_i w_{iJ},$$

we obtain

$$\frac{\partial E}{\partial w_{IJ}} = -2(t_J - y\_in_J)\frac{\partial y\_in_J}{\partial w_{IJ}}$$
$$= -2(t_J - y\_in_J)x_I.$$

Thus, the local error will be reduced most rapidly (for a given learning rate) by adjusting the weights according to the delta rule,

$$\Delta w_{IJ} = \alpha(t_J - y\_in_J)x_I.$$

The preceding two derivations of the delta rule can be generalized to the case where the training data are only samples from a larger data set, or probability distribution. Minimizing the error for the training set will also minimize the expected value for the error of the underlying probability distribution. (See Widrow & Lehr, 1990 or Hecht-Nielsen, 1990 for a further discussion of the matter.)

2.4.5 MADALINE

As mentioned earlier, a MADALINE consists of Many ADAptive Linear NEurons arranged in a multilayer net. The examples given for the perceptron and the derivation of the delta rule for several output units both indicate there is essentially no change in the process of training if several ADALINE units are combined in a single-layer net. In this section we will discuss a MADALINE with one hidden layer (composed of two hidden ADALINE units) and one output ADALINE unit. Generalizations to more hidden units, more output units, and more hidden layers, are straightforward.

Architecture

A simple MADALINE net is illustrated in Figure 2.24. The outputs of the two hidden ADALINES, z_1 and z_2, are determined by signals from the same input units X_1 and X_2. As with the ADALINES discussed previously, each output signal is the result of applying a threshold function to the unit's net input. Thus, y is a nonlinear function of the input vector (x_1, x_2). The use of the hidden units, Z_1 and Z_2, give the net computational capabilities not found in single layer nets, but also complicate the training process. In the next section we consider two training algorithms for a MADALINE with one hidden layer.

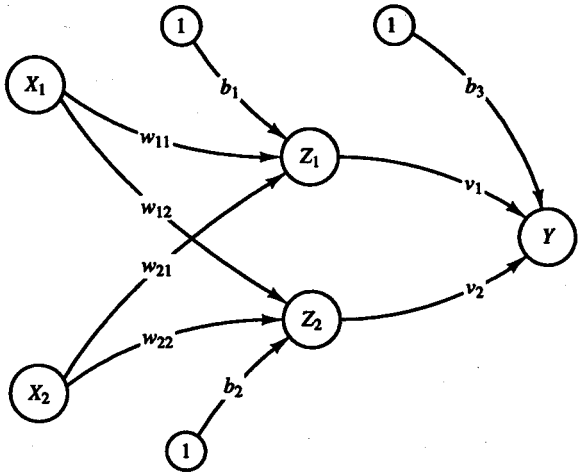

Figure 2.24 A MADALINE with two hidden ADALINES and one output ADALINE.

Algorithm

In the MRI algorithm (the original form of MADALINE training) [Widrow and Hoff, 1960], only the weights for the hidden ADALINES are adjusted; the weights for the output unit are fixed. The MRII algorithm [Widrow, Winter, & Baxter, 1987] provides a method for adjusting all weights in the net.

We consider first the MRI algorithm; the weights v_1 and v_2 and the bias b_3 that feed into the output unit Y are determined so that the response of unit Y is 1 if the signal it receives from either Z_1 or Z_2 (or both) is 1 and is -1 if both Z_1 and Z_2 send a signal of -1. In other words, the unit Y performs the logic function OR on the signals it receives from Z_1 and Z_2. The weights into Y are

$$v_1 = \frac{1}{2}$$

and

$$v_2 = \frac{1}{2},$$

with the bias

$$b_3 = \frac{1}{2}$$

(see Example 2.19). The weights on the first hidden ADALINE (w_{11} and w_{21}) and the weights on the second hidden ADALINE (w_{12} and w_{22}) are adjusted according to the algorithm.

Training Algorithm for MADALINE (MRI). The activation function for units Z_1, Z_2, and Y is

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0; \\ -1 & \text{if } x < 0. \end{cases}$$

Step 0. Initialize weights:
Weights v_1 and v_2 and the bias b_3 are set as described;
small random values are usually used for ADALINE weights.
Set the learning rate α as in the ADALINE training algorithm (a small value).

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each bipolar training pair, s:t, do Steps 3–7.

Step 3. Set activations of input units:

$$x_i = s_i.$$

Step 4. Compute net input to each hidden ADALINE unit:

$$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21},$$

$$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}.$$

Step 5. Determine output of each hidden ADALINE unit:

$$z_1 = f(z\_in_1),$$

$$z_2 = f(z\_in_2).$$

Step 6. Determine output of net:

$$y\_in = b_3 + z_1 v_1 + z_2 v_2;$$

$$y = f(y\_in).$$

Step 7. Determine error and update weights:
If $t = y$, no weight updates are performed.
Otherwise:
If $t = 1$, then update weights on Z_J, the unit whose net input is closest to 0,

$$b_J(\text{new}) = b_J(\text{old}) + \alpha(1 - z\_in_J),$$

$$w_{iJ}(\text{new}) = w_{iJ}(\text{old}) + \alpha(1 - z\_in_J)x_i;$$

If $t = -1$, then update weights on all units Z_k that have positive net input,

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z\_in_k),$$

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z\_in_k)x_i.$$

Step 8. Test stopping condition.
If weight changes have stopped (or reached an acceptable level), or if a specified maximum number of weight update iterations (Step 2) have been performed, then stop; otherwise continue.

Step 7 is motivated by the desire to (1) update the weights only if an error occurred and (2) update the weights in such a way that it is more likely for the net to produce the desired response.

If $t = 1$ and error has occurred, it means that all Z units had value -1 and at least one Z unit needs to have a value of $+1$. Therefore, we take Z_J to be the Z unit whose net input is closest to 0 and adjust its weights (using ADALINE training with a target value of $+1$):

$$b_J(\text{new}) = b_J(\text{old}) + \alpha(1 - z\_in_J),$$

$$w_{iJ}(\text{new}) = w_{iJ}(\text{old}) + \alpha(1 - z\_in_J)x_i.$$

If $t = -1$ and error has occurred, it means that at least one Z unit had value $+1$ and all Z units must have value -1. Therefore, we adjust the weights on all of the Z units with positive net input, (using ADALINE training with a target of -1):

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z\_in_k),$$

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z\_in_k)x_i.$$

MADALINES can also be formed with the weights on the output unit set to perform some other logic function such as AND or, if there are more than two hidden units, the "majority rule" function. The weight update rules would be modified to reflect the logic function being used for the output unit [Widrow & Lehr, 1990].

A more recent MADALINE training rule, called MRII [Widrow, Winter, & Baxter, 1987], allows training for weights in all layers of the net. As in earlier MADALINE training, the aim is to cause the least disturbance to the net at any step of the learning process, in order to cause as little "unlearning" of patterns for which the net had been trained previously. This is sometimes called the "don't rock the boat" principle. Several output units may be used; the total error for any input pattern (used in Step 7b) is the sum of the squares of the errors at each output unit.

Training Algorithm for MADALINE (MRII).

Step 0. Initialize weights:
Set the learning rate α.

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each bipolar training pair, s:t, do Steps 3–7.

Step 3–6. Compute output of net as in the MRI algorithm.

Step 7. Determine error and update weights if necessary:

If $t \neq y$, do Steps 7a–b for each hidden unit whose net input is sufficiently close to 0 (say, between $-.25$ and .25). Start with the unit whose net input is closest to 0, then for the next closest, etc.

Step 7a. Change the unit's ouput (from $+1$ to -1, or vice versa).

Step 7b. Recompute the response of the net.
If the error is reduced:
adjust the weights on this unit
(use its newly assigned output value as target and apply the Delta Rule).

Step 8. Test stopping condition.
If weight changes have stopped (or reached an acceptable level), or if a specified maximum number of weight update iterations (Step 2) have been performed, then stop; otherwise continue.

A further modification is the possibility of attempting to modify pairs of units at the first layer after all of the individual modifications have been attempted. Similarly adaptation could then be attempted for triplets of units.

Application

Example 2.20 Training a MADALINE for the XOR function

This example illustrates the use of the MRI algorithm to train a MADALINE to solve the XOR problem. Only the computations for the first weight updates are shown.

The training patterns are:

x_1	x_2	t
1	1	−1
1	−1	1
−1	1	1
−1	−1	−1

Step 0.

The weights into Z_1 and into Z_2 are small random values; the weights into Y are those found in Example 2.19. The learning rate, α, is .5.

Weights into Z_1			Weights into Z_2			Weights into Y		
w_{11}	w_{21}	b_1	w_{12}	w_{22}	b_2	v_1	v_2	b_3
.05	.2	.3	.1	.2	.15	.5	.5	.5

Step 1. Begin training.

Step 2. For the first training pair, (1, 1): −1

Step 3. $x_1 = 1, \quad x_2 = 1$

Step 4. $z\_in_1 = .3 + .05 + .2 = .55,$
$z\_in_2 = .15 + .1 + .2 = .45.$

Step 5. $z_1 = 1$,
$z_2 = 1$.
Step 6. $y\_in = .5 + .5 + .5$;
$y = 1$.
Step 7. $t - y = -1 - 1 = -2 \neq 0$, so an error occurred.
Since $t = -1$, and both Z units have positive net input,
update the weights on unit Z_1 as follows:

$$b_1(\text{new}) = b_1(\text{old}) + \alpha(-1 - z\_in_1)$$
$$= .3 + (.5)(-1.55)$$
$$= -.475$$
$$w_{11}(\text{new}) = w_{11}(\text{old}) + \alpha(-1 - z\_in_1)x_1$$
$$= .05 + (.5)(-1.55)$$
$$= -.725$$
$$w_{21}(\text{new}) = w_{21}(\text{old}) + \alpha(-1 - z\_in_1)x_2$$
$$= .2 + (.5)(-1.55)$$
$$= -.575$$

update the weights on unit Z_2 as follows:

$$b_2(\text{new}) = b_2(\text{old}) + \alpha(-1 - z\_in_2)$$
$$= .15 + (.5)(-1.45)$$
$$= -.575$$
$$w_{12}(\text{new}) = w_{12}(\text{old}) + \alpha(-1 - z\_in_2)x_1$$
$$= .1 + (.5)(-1.45)$$
$$= -.625$$
$$w_{22}(\text{new}) = w_{22}(\text{old}) + \alpha(-1 - z\_in_2)x_2$$
$$= .2 + (.5)(-1.45)$$
$$= -.525$$

After four epochs of training, the final weights are found to be:

$$w_{11} = -0.73 \qquad w_{12} = 1.27$$
$$w_{21} = 1.53 \qquad w_{22} = -1.33$$
$$b_1 = -0.99 \qquad b_2 = -1.09$$

Example 2.21 Geometric interpretation of MADALINE weights

The positive response region for the Madaline trained in the previous example is the union of the regions where each of the hidden units have a positive response. The

decision boundary for each hidden unit can be calculated as described in Section 2.1.3.

For hidden unit Z_1, the boundary line is

$$x_2 = -\frac{w_{11}}{w_{21}} x_1 - \frac{b_1}{w_{21}}$$

$$= \frac{0.73}{1.53} x_1 + \frac{0.99}{1.53}$$

$$= 0.48\, x_1 + 0.65$$

For hidden unit Z_2, the boundary line is

$$x_2 = -\frac{w_{12}}{w_{22}} x_1 - \frac{b_2}{w_{22}}$$

$$= \frac{1.27}{1.33} x_1 + \frac{1.09}{1.33}$$

$$= 0.96\, x_1 - 0.82$$

These regions are shown in Figures 2.25 and 2.26. The response diagram for the MADALINE is illustrated in Figure 2.27.

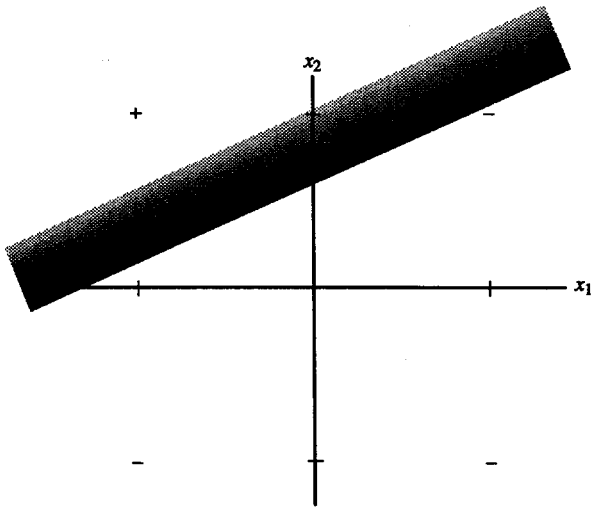

Figure 2.25 Positive response region for Z_1.

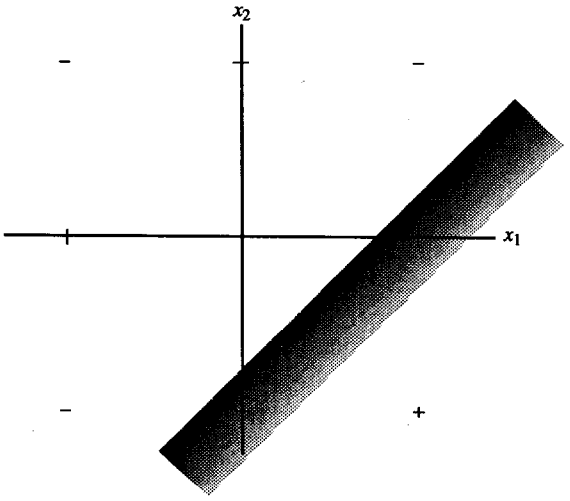

Figure 2.26 Positive response region for Z_2.

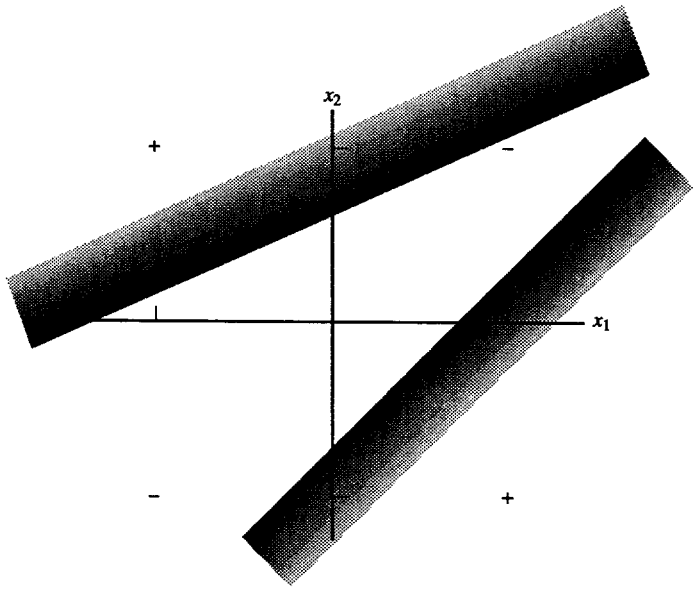

Figure 2.27 Positive response region for MADALINE for XOR function.

Discussion

The construction of sample multilayer nets may provide insights into the appropriate choice of parameters for multilayer nets in general, such as those trained using backpropagation (discussed in Chapter 6). For example, if the input patterns fall into regions that can be bounded (approximately) by a number of lines or planes, then the number of hidden units can be estimated.

It is possible to construct a net with $2p$ hidden units (in a single layer) that will learn p bipolar input training patterns (each with an associated bipolar target value) perfectly. Of course, that is not the primary (or at least not the exclusive) goal of neural nets; generalization is also important and will not be particularly good with so many hidden units. In addition, the training time and the number of interconnections will be unnecessarily large. However, $2p$ certainly gives an upper bound on the number of hidden units we might consider using.

For input that is to be assigned to different categories (the kind of input we have been considering in this chapter), we see that the regions which each neuron separates are bounded by straight lines. Closed regions (convex polygons) can be bounded by taking the intersection of several half-planes (bounded by the separating lines described earlier). Thus a net with one hidden layer (with p units) can learn a response region bounded by p straight lines. If responses in the same category occur in more than one disjoint region of the input space, an additional hidden layer to combine these regions will make training easier.

2.5 SUGGESTIONS FOR FURTHER STUDY

2.5.1 Readings

Hebb rule

The description of the original form of the Hebb rule is found in

HEBB, D. O. (1949). *The Organization of Behavior*. New York: John Wiley & Sons. Introduction and Chapter 4 reprinted in Anderson and Rosenfeld (1988), pp. 45–56.

Perceptrons

The description of the perceptron, as presented in this chapter, is based on

BLOCK, H. D. (1962). "The Perceptron: A Model for Brain Functioning, I." *Reviews of Modern Physics*, 34:123–135. Reprinted in Anderson and Rosenfeld (1988), pp. 138–150.

There are many types of perceptrons; for more complete coverage, see: