Chapter 9: The Perceptron

9.1 INTRODUCTION

At this point in the book, we have completed all of the exercises that we are going to do with the James program. These exercises have shown that distributed associative memories are capable of storing several different pairs of associations in the same set of connection weights. However, we have also seen that these memories are limited in power. These limitations have led researchers to propose alternative connectionist architectures that are intended to be more powerful than the memories that we have been studying to this point. One of these architectures is the perceptron (Rosenblatt, 1962), and this chapter provides a brief introduction to it. A more detailed introduction to the perceptron as an elaboration of the distributed associative memory can be found in (Dawson, 2003).

9.2 THE LIMITS OF DISTRIBUTED ASSOCIATIVE MEMORIES, AND BEYOND

Why are distributed associative memories subject to the limitations that we have discovered in the previous exercises? One possible answer to this question is that even the delta rule is not powerful enough to learn all of the associations of interest to us. However, this turns out not to be the problem. All of the learning rules that we will encounter in this chapter, and in later chapters of this book, are very similar – if not identical – to the delta rule.

A second possibility is that the basic building blocks that make up the distributed associative memory are not powerful enough. In the sections that follow, we will explore this possibility in more detail by focusing upon one architectural property, the activation function used by the output units of the distributed associative memory. We will briefly describe this function, consider its potential weakness, and propose some alternatives to it that will be incorporated into our next type of connectionist model.

9.2.1 Linear Activation Functions

Recall that the output units in the distributed associative memory compute their net input by adding up the weighted signals that are being sent to them by the input units through the network's connections. After net input is computed, its value is used as the activity of the output units in order to represent a memory's response to some cue pattern.

In a more general account of output unit processing, it is useful to consider an output unit as computing two different mathematical equations. The first is the net input function, which is the equation used to compute the signal that enters the output unit. In all of the networks that we will consider in this book, the net input equation is a simple sum. The second equation is the activation function, and it is used to convert the net input that has been computed into some level of internal activity in the unit. There are many different kinds of activation functions that can be found in modern connectionist networks (Duch & Jankowski, 1999). The activation function that is used by the output units in the distributed associative memory is particularly weak, and one way in which the power of such a memory could be extended would be to replace this function with one that is more powerful.

In the simulations that we have been conducting to this point in the book, the identity function is used to compute the activity of an output unit. In other words, the output unit's activity is exactly equal to its net input. Unfortunately, the identity function implements a particularly weak relationship between net input and activity, and this weak relationship is to the detriment of the distributed associative memory. When net input is small, output activity is small. When net input is medium, output activity is medium. When net input is large, output activity is large. In short, when the identity function is used to determine unit activity, the relationship between net input and output unit activity is linear.

Many researchers would argue that connectionist models are important because they are biologically inspired, and would also claim that the processing units in such models are analogous to neurons in the brain. Interestingly, in the behavior of a neuron, there is *not* a linear relationship between net input and activity. When net input is small, a neuron's activity is small in the sense that it does not generate an action potential. As net input gradually increases, the neuron's activity does not change -- it would still fail to generate an action potential. It is only when the net input becomes sufficiently large – when it exceeds some threshold – that an action potential is generated. In short, in neurons there is a nonlinear relationship between net input and activation. In order to increase the power of the distributed memory, this kind of nonlinearity has to be introduced into the output units by replacing the identity function with some other, nonlinear, activation function.

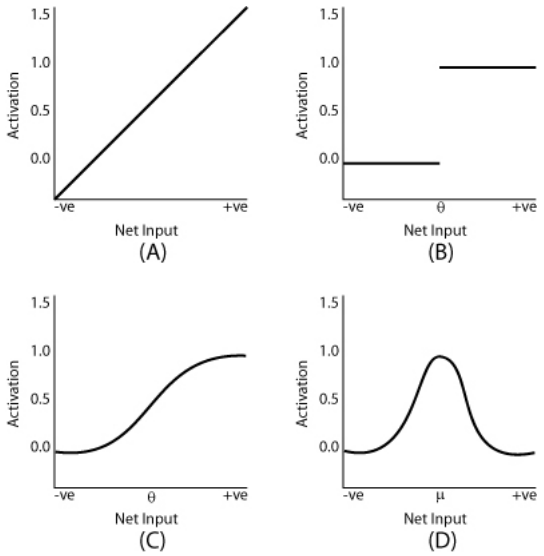
9.2.2 Nonlinear Activation Functions

In general terms, neurons process information by detecting weak electrical signals, called graded potentials, that stimulate, and travel through, their dendrites. If enough of these weak graded potentials arrive at the cell body of the neuron at the same time, then their cumulative effect disrupts the resting electrical state of the neuron. This results in a massive depolarization of the membrane of the neuron's axon, called an action potential, which travels along the axon to eventually stimulate some other neuron.

While graded potentials gradually decrease in intensity over time and distance, an action potential does not. An action potential is an electrical signal of constant intensity. The fact that neurons generate action potentials of fixed intensity is one of the fundamental discoveries of neuroscience, and has been called the all-or-none law. "The all-or-none law guarantees that once an action potential is generated it is always full size, minimizing the possibility that information will be lost along the way" (Levitan & Kaczmarek, 1991).

(McCulloch & Pitts, 1943) realized that the all-or-none law enabled them to ignore the detailed biology of neural function, and allowed them to instead describe neurons as devices that made true or false logical assertions about input information. "The all-or-none law of nervous activity is sufficient to ensure that the activity of any neuron may be represented as a proposition. Physiological relations existing among nervous activities correspond, of course, to relations among the propositions; and the utility of the representation depends upon the identity of these relations with those of the logical propositions. To each reaction of any neuron there is a corresponding assertion of a simple proposition." (McCulloch & Pitts, 1988).

In order to define connectionist processing units in a fashion consistent with the logical view of (McCulloch & Pitts, 1943), we need to define an activation function that has two different (but related) properties. First, the function has to implement a nonlinear relationship between net input and activation. Second, the function has to implement some maximum and minimum levels of activity that can be logically interpreted as a "true" or "false" response to a proposition. Many different activation functions meet these two requirements. In our exploration of the perceptron, we will be concerned with three of these, which are all illustrated in Figure 9-1, which also includes the linear activation function (Figure 9-1A) for comparison. These three functions are the step function (Figure 9-1B), the logistic function (Figure 9-1C), and the Gaussian function (Figure 9-1D). The sections that follow briefly describe each of these activation functions.



9.2.3 THE STEP FUNCTION

The first nonlinear activation function to consider is called the step function. It was the activation function that was originally used to model the all-or-none law in artificial neural networks (McCulloch & Pitts, 1943), and was also the activation function that was used in the original perceptron architecture (Rosenblatt, 1962).

With the step function, a processing unit is considered to be in one of only two possible states: on or off. We will be representing the on state with an activation value of 1, and the off state with an activation value of 0. However, sometimes in the literature one might find that the off state is represented with an activation value of –1.

A unit's net input is converted into one of these two activation values by comparing the net input to some threshold value θ. If the net input is less than or equal to θ, then the unit's activity is assigned a value of 0. If the net input is greater than θ, then the unit's activity is assigned a value of 1. Note with this activation function that the relationship between net input and activity is clearly nonlinear (i.e., compare Figure 9-1A to Figure 9-1B). Also note that this activation function implements the all-or-none law, because no matter how high above θ that the net input becomes, the resulting unit activation is always equal to 1.

9.2.4 THE LOGISTIC FUNCTION

The second nonlinear activation function that we will be using in the perceptron is called logistic function. It is a continuous approximation of the step function (see Figure 9-1C); its continuous nature permits calculus to be used to derive learning rules for perceptrons and multilayer perceptrons (Rumelhart, Hinton, & Williams, 1986). Units that use this type of activation function are sometimes called *integration devices* (Ballard, 1986).

The logistic function converts net input into activation according to the following equation: $f(net_i) = 1 / (1 + \exp(-net_i + \theta_j))$. In this equation, the activity of output unit i is represented as $f(net_i)$, the net input of this unit is represented as net_i, and the bias of this unit is represented as θ_j. The bias of a unit that uses the logistic function is analogous to the threshold of a unit that uses the step function. Bias is the value of net input that produces an activation value of 0.50.

9.2.5 THE GAUSSIAN FUNCTION

Both the step function and the logistic function are monotonic, in the sense that increases in net input never result in decreases in either of these activation functions. This is consistent with the behavior of some, but not all, neurons. Some neurons – such as the cone receptors in the retina – have nonmonotonic activation profiles. What this means is that they are tuned to respond to a narrow range of net input values (Ballard, 1986), and can therefore be called *value units*. If the net input is below this range, the unit will not respond. However, if the net input is above this range, the unit will also not respond. Another way to describe this nonmonotonicity is to say that the unit has two thresholds: a lower threshold for turning on, and an upper threshold for turning off.

The Gaussian function is an equation that implements this particular form of nonmonotonicity, as is illustrated in Figure 9-1D. It was originally proposed in a network architecture that represented an elaboration of the multilayered perceptrons that were proposed in the 1980s (Dawson & Schopflocher, 1992). The equation for the Gaussian is: $G(net_i) = \exp(-\pi(net_i - \mu_j)^2$. In this equation, the activity of output unit *i* is represented as $G(net_i)$, the net input of this unit is represented as net_i, and the bias of this unit is represented as μ_j. The value μ is similar to a threshold, in the sense that it indicates the net input value to which the unit is tuned. However, it is not a threshold. Instead, it is the value of the net input that results in the unit generating a maximum activity of 1. When net input becomes moderately smaller than μ, or moderately larger than μ, then unit activity drops off significantly, and asymptotes to a value of 0.

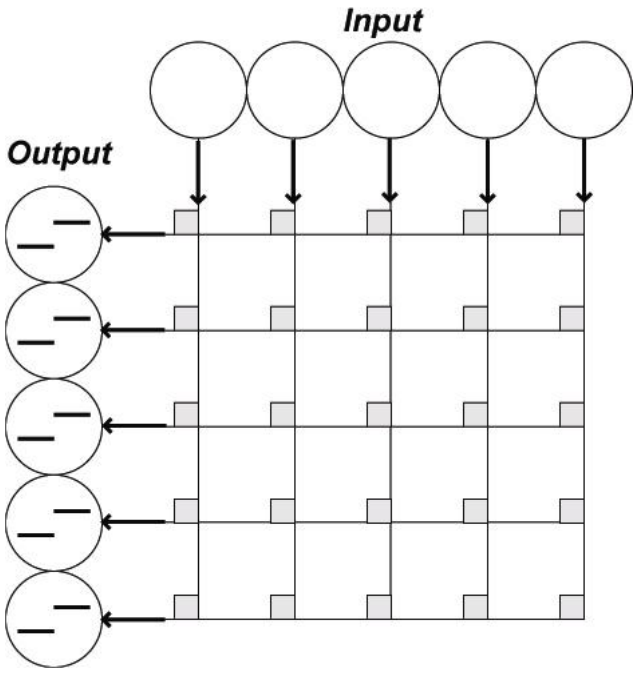
The fact that the Gaussian function can be described as having an upper and lower threshold produces some interesting changes to the behavior of perceptrons built from value units. Value units can solve some problems that cannot be solved by perceptrons built from integration devices. However, integration devices can solve some problems that pose problems to value units. We will be exploring these sorts of issues in the upcoming exercises.

9.3 PROPERTIES OF THE PERCEPTRON

9.3.1 WHAT DO PERCEPTRONS DO?

The perceptron is very similar to the distributed associated memory. It too consists of a bank of input units, a bank of one or more output units, and a set of modifiable connections that link every input unit to every output unit. A learning rule is used to modify the connection weights in order to train the perceptron to create an association between an input pattern and an output pattern. The only crucial difference between the two architectures is the fact that the output units in a perceptron use a nonlinear activation function. As was discussed earlier, the purpose of the nonlinear activation function is to model the all-or-none law governing the generation of action potentials.

The similarity between the two architectures is emphasized in Figure 9-2, which illustrates the basic architecture of a perceptron. This figure renders the perceptron as if it were a distributed memory of the type that we have studied with the James program. The only difference between the two types of networks is the activation function of the output units. This difference has been added to the figure by drawing a step function inside each of the the output units in the figure.



The nonlinear activation function in the output units of a perceptron leads to a slight difference in interpreting the kind of task that a perceptron should be trained to perform. The output units of a perceptron are trained to generate a response that will be interpreted as being either on or off. This means that the output units can be assigned a logical interpretation, in the sense of McCulloch and Pitts. As a result, while a perceptron can be viewed as a kind of associative memory, the kinds of associations that it learns to make will usually be interpreted in a different fashion than were the associations that were described in previous chapters. The logical nature of an output unit's activity means that a perceptron is usually described as a device that makes decisions – it classifies input patterns. The nonlinear activation function in perceptron is used to assign input patterns to a particular category, where this assignment is all or none.

For example, consider a simple kind of problem called the majority problem. In a majority problem, a perceptron would have *N* input units, and a single output unit. If the majority of the input units were turned on, then the output unit of the perceptron would be trained to turn on to those patterns. If less than the majority of the input units were turned on, then the output unit of the perceptron would be trained to turn off. Imagine that *N* was equal to 5. In this case, whenever three, four, or five of the input units were activated, then the perceptron would be trained to turn on. If zero, one, or two of the input units were activated, then the perceptron would be trained to turn off. Thus while it is perfectly legitimate to view the perceptron as learning to associate one kind of response with some inputs, and a different kind of response with others, more specifically we can say that the perceptron has learned to decide that some patterns have the majority of their input units turned on, while others do not. Our account of the perceptron as a pattern classifier is almost completely due to the fact that it uses a nonlinear activation function that is binary in nature.

9.3.2 The Basic Architecture

The basic architecture of a perceptron was illustrated above in Figure 9-2. It consists of a bank of two or more input units, a bank of one or more output units, and a set of connection weights that link input units directly to output units. Each connection weight is associated with a weight, and associations between inputs and outputs are stored in this architecture by using a learning rule to modify these weights.

The input units in a perceptron are identical in nature to the input units for the distributed associative memory. The input units are used to represent patterns that are to be presented as stimuli to the perceptron. The activities of the input units can either be binary or continuous, depending on the desired interpretation of what each input unit represents. Input unit activities can be used to represent features that are either very simple ore very complicated, depending on the problem to be presented to the network. As an example of a simple input, an input unit could be turned on or off to represent whether some simple stimulus was present or absent in the environment.

The output units in a perceptron represent an elaboration of the output units in a distributed associative memory. The two are identical with respect to their net input function. The output units in a perceptron calculate their net input by summing the signal being sent by each input unit after the signal has been scaled by a connection weight. As we have been emphasizing this chapter, the difference between the output units in the two different kinds of networks is with respect to the activation function that is used to convert net input into internal activity. One consequence of using nonlinear activation functions in the output units is that we have to pay attention to the kind of learning rule that is used to modify connection weights. Three different learning rules will be explored, one for each of the three different activation functions that were described earlier.

9.3.3 THE ROSENBLATT LEARNING RULE

In the original description of the perceptron architecture, (Rosenblatt, 1962) developed a learning rule that could be used when output units used the step function. The logic of this learning rule is that connection weight modifications are contingent upon network performance. Let us define the error of some output unit j as the value $(t_j - a_j)$, where t_j is the desired or target value of the output, and a_j is the actual activity that the output unit generates. In calculating $(t_j - a_j)$ there are three possible outcomes. First, the value of $(t_j - a_j)$ could be equal to 0. In this case, the output unit has generated the correct response to an input pattern and no connection weight changes are required. Second, the value of $(t_j - a_j)$ could be equal to 1. In this case, the output unit has generated an error by turning off when it was desired that the unit actually turn on. In order to deal with this situation, it is necessary to increase the net input to the output unit. This could be accomplished by increasing the size of the connection weights. Third, the value of $(t_j - a_j)$ could be equal to -1. In this case, the output unit has made an error by turning on when it should have turned off. The remedy for this problem would be to decrease the unit's net input by subtracting from the values of the connection weights.

An examination of the three possible values for error, and of the resulting change that these values imply for connection weights, indicates that the delta rule that we saw used in the distributed associative memory can also be used for the perceptron. (Rosenblatt, 1962) proposed that the desired change to the weight connecting input unit i to output unit j can be expressed as: $\Delta w_{ij} = \eta(t_j - a_j)\, a_i$. In this equation, η is a learning rate that will ordinarily range between 0 and1, $(t_j - a_j)$ is the error calculated for output unit j, under the assumption that a_j is calculated using the step function, and a_i is the activity of input unit i.

An output unit that uses the step function can be described as a classifier that makes a single straight cut through a pattern space. Each input pattern is represented as a point in that pattern space, with the position of each point being defined by the activity of each input unit. The input unit activities are used to define coordinates in the pattern space. Patterns that fall on one side of the cut the output unit makes will result in the output unit turning off. Patterns that fall on the other side of the cut will result in the output unit turning on. When a perceptron's weights are trained using the delta rule, the result is that the cut through pattern space made by the output unit is rotated. However, to solve some problems we also need to be able to translate this cut through space instead of just rotating it. In order to translate the cut, we need to be able to modify the threshold θ_j of the output unit. This can easily be done by assuming that the threshold is the value of the connection weight that comes from an additional input unit that is always on.

With this interpretation, the desired change in the threshold θ_j of some output unit *j* can be defined as: $\Delta\theta_j = \eta\ (t_j - a_j)\ 1$.

The delta rule, when applied to a perceptron, is very powerful. (Rosenblatt, 1962) used it to derive his famous perceptron convergence theorem. This theorem proved that if a solution to a pattern classification problem could be represented in the connection weights of a perceptron, then the delta rule was guaranteed to find a set of connection weights that solved the problem. For our purposes, the fact that the delta rule can be used to train a perceptron also provides additional evidence about the similarity between perceptrons and distributed associative memories.

9.3.4 THE GRADIENT DESCENT RULE

Rumelhart, Hinton, and Williams (1986) defined the total error for a network whose output units are integration devices as the sum of squared error, *E*, where the squared error is totaled over every output unit and every pattern in the training set: $E = \frac{1}{2} \Sigma\Sigma(t_{jp} - a_{jp})^2$. In this equation, t_{jp} represents the target activity for output unit *j* when it is presented pattern *p*, and a_{jp} represents the observed activity for output unit *j* when it is presented pattern *p*. The first summation sign is performed over the total number of patterns in the training set, and the second summation sign is performed over the total number of output units in the perceptron.

With network error defined as above, and with the logistic equation serving as a continuous approximation of the step function, Rumelhart, Hinton, and Williams (1986) were in a position to use calculus to determine how a weight should be altered in order to decrease error. They derived equations that determined how a change in a weight changed the net input to an output unit, how the resulting change in net input affected the output unit's activity, and how altering the output unit's activity affected error as defined above. They then used these equations to define how to change a weight, when a given pattern has been presented, in order to have the maximum effect of learning. This definition was a new statement of the error for an output unit *j*, which we will represent as δ_j. They found that the fastest way to decrease network error was to take the error that was used in the delta rule, and to multiply this error by the first derivative of the logistic equation, $f'(net_j)$. The first derivative of the logistic equation is equal to the value $a_j\ (1 - a_j)$. So, the new equation for output unit error became: $\delta_j = (t_j - a_j)\ f'(net_j) = (t_j - a_j)\ a_j\ (1 - a_j)$.

A new learning rule for a perceptron that uses the logistic activation function can be defined by inserting this new error term into the delta rule equation. This results in what we will call the gradient descent rule for training a perceptron: $\Delta w_{ij} = \eta\ \delta_j\ a_i = \eta\ (t_j - a_j)\ a_j\ (1 - a_j)\ a_i$.

As was the case with the delta rule, the bias of the logistic can also be modified by the learning rule. To do this, the bias is treated as if it were equal to the weight of a connection between the output unit and an additional input unit that is always activated with a value of 1 for every training pattern in the training set. With this assumption, the gradient descent rule for modifying bias can be stated as: $\Delta\theta_j = \eta\ \delta_j\ 1 = \eta\ (t_j - a_j)\ a_j\ (1 - a_j)\ 1$.

What is the purpose of multiplying the output unit's error value by the derivative of the activation function before modifying the weight? At any point in time during learning, a perceptron can be represented as a single point or location on a surface. The coordinates of the location are given by the current values of all of the perceptron's weights (and of its bias). Each point on this surface has a height, which is equal to the value of total network error. One can think about learning as a process that moves the perceptron along this error surface, always seeking a minimum error value. Every time that the perceptron changes its connection weights, it takes a step "downhill" on the error surface, moving to a location that has lower height (i.e., a lower error value). The size of the step that is taken is determined by the size of the learning rate. The direction in which the step is taken is dictated by the error calculated for an output unit. In order to minimize total network error as quickly as possible, it is desirable that at each step the perceptron move in the direction that is the steepest "downhill". The first derivative of the activation function

is the part of the equation that determines the direction from the current location on the space that has the steepest downhill slope. By multiplying output unit error by the derivative, the network is permitted to take the shortest "diagonal" path along the error surface.

9.3.5 THE DAWSON-SCHOPFLOCHER RULE

How would one train a perceptron whose output units are value units? The first plausible approach would be to adopt the gradient descent rule. To do this, one would define a new error term by taking the gradient descent rule described in Section 9.3.4 and replacing the first derivative of the logistic ($f'(net_j)$) with the first derivative of the Gaussian ($G'(net_j)$), which is equal to $-2\pi(net_j)G(net_j) = -2\pi(net_j)(\exp(-\pi(net_i - \mu_j)^2))$. However, (Dawson & Schopflocher, 1992) found that when they did this, learning was very inconsistent. In some cases, training proceeded very quickly. However, in the majority of cases, the network did not learn to solve the problem. Instead, its connection weights were changed in such a way that the network learned to turn off to all of the training patterns by moving all of the net inputs into one of the tails of the Gaussian function.

To correct this problem, Dawson and Schopflocher (1992) elaborated the equation for total network error by adding a heuristic component to it. This heuristic component was designed to keep some of the net inputs in the middle of the Gaussian function. It was a statement that asserted that when the desired activation value for output unit j was 1, the error term should include an attempt to minimize the difference between the net input to the unit net_j and the unit's mean μ_j. Their elaborated expression for total network error was: $E = \frac{1}{2} \Sigma\Sigma(t_{pj} - a_{pj})^2 + \frac{1}{2} \Sigma\Sigma\, t_{pj} (net_{pj} - \mu_j)^2$.

After defining this elaborated error term, Dawson and Schopflocher (1992) used calculus to determine what kind of weight change was required to decrease total network error. As was the case for the derivation of the gradient descent rule, this resulted in a new expression for output unit error to be included in an expression that was similar to the delta rule. However, because their elaborated error expression had two components, Dawson and Schopflocher found that the error for an output value unit also had two components.

The first component was identical to the expression in the gradient descent rule that defined the term δ_{pj}, with the exception that it used the first derivative of the Gaussian instead of the logistic: $\delta_{pj} = (t_{pj} - a_{pj})\, G'(net_{pj}) = (t_{pj} - a_{pj})(-2\pi(net_{pj})(\exp(-\pi(net_{pj} - \mu_{pj})^2)))$.

The second component was represented with the term ε_j, and was the part of output unit error that was related to the heuristic information that Dawson and Schopflocher (1992) added to the equation for total network error. The equation for this error term was: $\varepsilon_{pj} = t_{pj}\,(net_{pj} - \mu_j)$.

The complete expression for an output unit's error was found to be the difference between these two expressions of error, and Dawson and Schopflocher discovered that a learning rule for a network of value units was defined by a gradient descent rule that used this more complex measure of output unit error: $\Delta w_{ij} = \eta(\delta_j - \varepsilon_j)a_i$.

Similarly, Dawson and Schopflocher (1992) that the mean of an output unit's Gaussian could also be trained. This was done by assuming that the value j was the weight from an additional input unit that was always turned on. This assumption results in a learning expression very similar to the ones that were provided earlier for training the threshold of a step function or the bias of a logistic function: $\Delta\mu = \eta(\delta_j - \varepsilon_j)$.

In summary, Dawson and Schopflocher (1992) demonstrated that a perceptron with output units that used the Gaussian activation function could be trained with a variant of the gradient descent rule that was derived for integration devices. The learning rule that they developed differed from the more traditional gradient descent rule in only two ways. First, it used the first de-

rivative of the Gaussian equation. Second, it used an elaborated expression for output unit error, which included a heuristic component that is not found in the traditional gradient descent rule.

9.4 WHAT COMES NEXT

In this chapter, we have introduced the notion of a perceptron as being an elaboration of the distributed associative memory that was explored in the first chapters of this book. We have also described three different versions of the perceptron: one that uses the delta rule to train output units that use the step activation function, one that uses the gradient descent rule to train output units that are integration devices, and one that uses a modified gradient descent rule to train output units that are value units.

In the chapters that follow, we will be exploring the advantages and disadvantages of these three variations on the generic perceptron architecture. We will be performing a number of exercises that illustrate the kinds of problems that these networks can solve, as well as the kinds of problems that pose difficulty for these networks. We will also be exploring how the perceptron can be used to explore some issues that are current in the animal learning literature. All of these explorations will be conducted with a new program, called Rosenblatt. General instructions for installing and using this program are provided in Chapter 10.