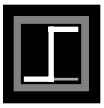
4 Perceptron Learning Rule

Objectives

One of the questions we raised in Chapter 3 was: "How do we determine the weight matrix and bias for perceptron networks with many inputs, where it is impossible to visualize the decision boundaries?" In this chapter we will describe an algorithm for *training* perceptron networks, so that they can *learn* to solve classification problems. We will begin by explaining what a learning rule is and will then develop the perceptron learning rule. We will conclude by discussing the advantages and limitations of the single-layer perceptron network. This discussion will lead us into future chapters.

Theory and Examples



In 1943, Warren McCulloch and Walter Pitts introduced one of the first artificial neurons [McPi43]. The main feature of their neuron model is that a weighted sum of input signals is compared to a threshold to determine the neuron output. When the sum is greater than or equal to the threshold, the output is 1. When the sum is less than the threshold, the output is 0. They went on to show that networks of these neurons could, in principle, compute any arithmetic or logical function. Unlike biological networks, the parameters of their networks had to be designed, as no training method was available. However, the perceived connection between biology and digital computers generated a great deal of interest.

In the late 1950s, Frank Rosenblatt and several other researchers developed a class of neural networks called perceptrons. The neurons in these networks were similar to those of McCulloch and Pitts. Rosenblatt's key contribution was the introduction of a learning rule for training perceptron networks to solve pattern recognition problems [Rose58]. He proved that his learning rule will always converge to the correct network weights, if weights exist that solve the problem. Learning was simple and automatic. Examples of proper behavior were presented to the network, which learned from its mistakes. The perceptron could even learn when initialized with random values for its weights and biases.

Unfortunately, the perceptron network is inherently limited. These limitations were widely publicized in the book *Perceptrons* [MiPa69] by Marvin Minsky and Seymour Papert. They demonstrated that the perceptron networks were incapable of implementing certain elementary functions. It was not until the 1980s that these limitations were overcome with improved (multilayer) perceptron networks and associated learning rules. We will discuss these improvements in Chapters 11 and 12.

Today the perceptron is still viewed as an important network. It remains a fast and reliable network for the class of problems that it can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks. Thus, the perceptron network, and its associated learning rule, are well worth discussion here.

In the remainder of this chapter we will define what we mean by a learning rule, explain the perceptron network and learning rule, and discuss the limitations of the perceptron network.

Learning Rules

Learning Rule

As we begin our discussion of the perceptron learning rule, we want to discuss learning rules in general. By *learning rule* we mean a procedure for modifying the weights and biases of a network. (This procedure may also

be referred to as a training algorithm.) The purpose of the learning rule is to train the network to perform some task. There are many types of neural network learning rules. They fall into three broad categories: supervised learning, unsupervised learning and reinforcement (or graded) learning.

Supervised Learning
Training Set

In *supervised learning*, the learning rule is provided with a set of examples (the *training set*) of proper network behavior:

$$\{\mathbf{p}_1,\mathbf{t}_1\}, \{\mathbf{p}_2,\mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q,\mathbf{t}_Q\}, \tag{4.1}$$

Target

where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding correct (*target*) output. As the inputs are applied to the network, the network outputs are compared to the targets. The learning rule is then used to adjust the weights and biases of the network in order to move the network outputs closer to the targets. The perceptron learning rule falls in this supervised learning category. We will also investigate supervised learning algorithms in Chapters 7–12.

Reinforcement Learning

Reinforcement learning is similar to supervised learning, except that, instead of being provided with the correct output for each network input, the algorithm is only given a grade. The grade (or score) is a measure of the network performance over some sequence of inputs. This type of learning is currently much less common than supervised learning. It appears to be most suited to control system applications (see [BaSu83], [WhSo92]).

Unsupervised Learning

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. At first glance this might seem to be impractical. How can you train a network if you don't know what it is supposed to do? Most of these algorithms perform some kind of clustering operation. They learn to categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization. We will see in Chapters 13–16 that there are a number of unsupervised learning algorithms.

Perceptron Architecture

Before we present the perceptron learning rule, let's expand our investigation of the perceptron network, which we began in Chapter 3. The general perceptron network is shown in Figure 4.1.

The output of the network is given by

$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp} + \mathbf{b}) . \tag{4.2}$$

(Note that in Chapter 3 we used the *hardlims* transfer function, instead of *hardlim*. This does not affect the capabilities of the network. See Exercise E4.6.)

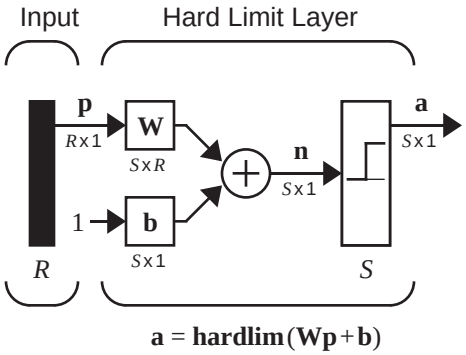


Figure 4.1 Perceptron Network

It will be useful in our development of the perceptron learning rule to be able to conveniently reference individual elements of the network output. Let's see how this can be done. First, consider the network weight matrix:

$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,R} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,R} \\ \vdots & \vdots & & \vdots \\ w_{S,1} & w_{S,2} & \cdots & w_{S,R} \end{bmatrix}. \tag{4.3}$$

We will define a vector composed of the elements of the ith row of $\mathbf{W}$:

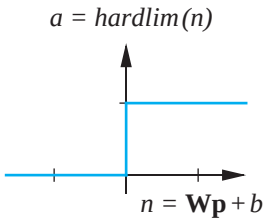
$${}_i\mathbf{w} = \begin{bmatrix} w_{i,1} \\ w_{i,2} \\ \vdots \\ w_{i,R} \end{bmatrix}. \tag{4.4}$$

Now we can partition the weight matrix:

$$\mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}. \tag{4.5}$$

This allows us to write the ith element of the network output vector as

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i) \ . \tag{4.6}$$



Recall that the *hardlim* transfer function (shown at left) is defined as:

$$a = hardlim(n) = \begin{cases} 1 \ \ if \ \ n \geq 0 \\ 0 \ \ otherwise. \end{cases} \tag{4.7}$$

Therefore, if the inner product of the ith row of the weight matrix with the input vector is greater than or equal to $-b_i$, the output will be 1, otherwise the output will be 0. *Thus each neuron in the network divides the input space into two regions*. It is useful to investigate the boundaries between these regions. We will begin with the simple case of a single-neuron perceptron with two inputs.

4

Single-Neuron Perceptron

Let's consider a two-input perceptron with one neuron, as shown in Figure 4.2.

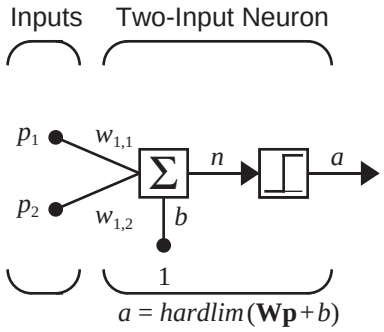


Figure 4.2 Two-Input/Single-Output Perceptron

The output of this network is determined by

$$\begin{aligned} a &= hardlim(n) = hardlim(\mathbf{W}\mathbf{p} + b) \\ &= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim(w_{1,1}p_1 + w_{1,2}p_2 + b) \end{aligned} \tag{4.8}$$

Decision Boundary

The *decision boundary* is determined by the input vectors for which the net input n is zero:

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = 0 \, . \tag{4.9}$$

To make the example more concrete, let's assign the following values for the weights and bias:

$$w_{1,1} = 1,\ w_{1,2} = 1,\ b = -1. \tag{4.10}$$

The decision boundary is then

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = p_1 + p_2 - 1 = 0. \tag{4.11}$$

This defines a line in the input space. On one side of the line the network output will be 0; on the line and on the other side of the line the output will be 1. To draw the line, we can find the points where it intersects the p_1 and p_2 axes. To find the p_2 intercept set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{-1}{1} = 1 \quad \text{if } p_1 = 0. \tag{4.12}$$

To find the p_1 intercept, set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{-1}{1} = 1 \quad \text{if } p_2 = 0. \tag{4.13}$$

The resulting decision boundary is illustrated in Figure 4.3.

To find out which side of the boundary corresponds to an output of 1, we just need to test one point. For the input $\mathbf{p} = \begin{bmatrix} 2 & 0 \end{bmatrix}^T$, the network output will be

$$a = hardlim({}_1\mathbf{w}^T\mathbf{p} + b) = hardlim\left(\begin{bmatrix} 1 & 1 \end{bmatrix}\begin{bmatrix} 2 \\ 0 \end{bmatrix} - 1\right) = 1. \tag{4.14}$$

Therefore, the network output will be 1 for the region above and to the right of the decision boundary. This region is indicated by the shaded area in Figure 4.3.

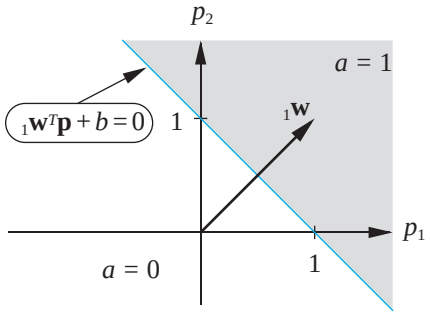
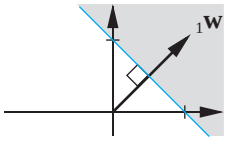
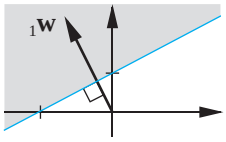


Figure 4.3 Decision Boundary for Two-Input Perceptron



We can also find the decision boundary graphically. The first step is to note that the boundary is always orthogonal to ${}_1\mathbf{w}$, as illustrated in the adjacent figures. The boundary is defined by

$${}_1\mathbf{w}^T\mathbf{p} + b = 0. \tag{4.15}$$

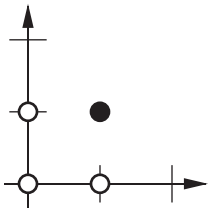


For all points on the boundary, the inner product of the input vector with the weight vector is the same. This implies that these input vectors will all have the same projection onto the weight vector, so they must lie on a line orthogonal to the weight vector. (These concepts will be covered in more detail in Chapter 5.) In addition, any vector in the shaded region of Figure 4.3 will have an inner product greater than $-b$, and vectors in the unshaded region will have inner products less than $-b$. Therefore the weight vector ${}_1\mathbf{w}$ will always point toward the region where the neuron output is 1.

After we have selected a weight vector with the correct angular orientation, the bias value can be computed by selecting a point on the boundary and satisfying Eq. (4.15).

Let's apply some of these concepts to the design of a perceptron network to implement a simple logic function: the AND gate. The input/target pairs for the AND gate are

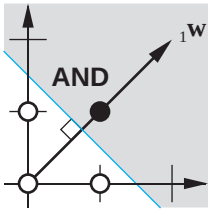
$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 0\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 1\right\}.$$



The figure to the left illustrates the problem graphically. It displays the input space, with each input vector labeled according to its target. The dark circles ● indicate that the target is 1, and the light circles ○ indicate that the target is 0.

The first step of the design is to select a decision boundary. We want to have a line that separates the dark circles and the light circles. There are an infinite number of solutions to this problem. It seems reasonable to choose the line that falls "halfway" between the two categories of inputs, as shown in the adjacent figure.

Next we want to choose a weight vector that is orthogonal to the decision boundary. The weight vector can be any length, so there are infinite possibilities. One choice is



$${}_1\mathbf{w} = \begin{bmatrix}2\\2\end{bmatrix}, \tag{4.16}$$

as displayed in the figure to the left.

Finally, we need to find the bias, b. We can do this by picking a point on the decision boundary and satisfying Eq. (4.15). If we use $\mathbf{p} = \begin{bmatrix} 1.5 & 0 \end{bmatrix}^T$ we find

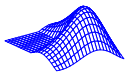
$$ {}_1\mathbf{w}^T\mathbf{p} + b = \begin{bmatrix} 2 & 2 \end{bmatrix}\begin{bmatrix} 1.5 \\ 0 \end{bmatrix} + b = 3 + b = 0 \quad \Rightarrow \quad b = -3. \tag{4.17} $$

We can now test the network on one of the input/target pairs. If we apply $\mathbf{p}_2$ to the network, the output will be

$$ a = hardlim({}_1\mathbf{w}^T\mathbf{p}_2 + b) = hardlim\left(\begin{bmatrix} 2 & 2 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} - 3 \right) \tag{4.18} $$

$$ a = hardlim(-1) = 0, $$

which is equal to the target output t_2. Verify for yourself that all inputs are correctly classified.



*To experiment with decision boundaries, use the Neural Network Design Demonstration Decision Boundaries (*nnd4db*).*

Multiple-Neuron Perceptron

Note that for perceptrons with multiple neurons, as in Figure 4.1, there will be one decision boundary for each neuron. The decision boundary for neuron i will be defined by

$$ {}_i\mathbf{w}^T\mathbf{p} + b_i = 0. \tag{4.19} $$

A single-neuron perceptron can classify input vectors into two categories, since its output can be either 0 or 1. A multiple-neuron perceptron can classify inputs into many categories. Each category is represented by a different output vector. Since each element of the output vector can be either 0 or 1, there are a total of 2^S possible categories, where S is the number of neurons.

Perceptron Learning Rule

Now that we have examined the performance of perceptron networks, we are in a position to introduce the perceptron learning rule. This learning rule is an example of supervised training, in which the learning rule is provided with a set of examples of proper network behavior:

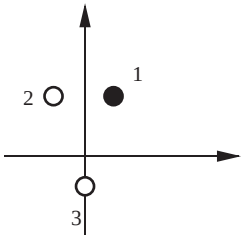
$$ \{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \ldots, \{\mathbf{p}_Q, \mathbf{t}_Q\}, \tag{4.20} $$

where $\mathbf{p}_q$ is an input to the network and $\mathbf{t}_q$ is the corresponding target output. As each input is applied to the network, the network output is compared to the target. The learning rule then adjusts the weights and biases of the network in order to move the network output closer to the target.

Test Problem

In our presentation of the perceptron learning rule we will begin with a simple test problem and will experiment with possible rules to develop some intuition about how the rule should work. The input/target pairs for our test problem are

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\2\end{bmatrix}, t_1 = 1\right\}\left\{\mathbf{p}_2 = \begin{bmatrix}-1\\2\end{bmatrix}, t_2 = 0\right\}\left\{\mathbf{p}_3 = \begin{bmatrix}0\\-1\end{bmatrix}, t_3 = 0\right\}.$$



The problem is displayed graphically in the adjacent figure, where the two input vectors whose target is 0 are represented with a light circle ○, and the vector whose target is 1 is represented with a dark circle ●. This is a very simple problem, and we could almost obtain a solution by inspection. This simplicity will help us gain some intuitive understanding of the basic concepts of the perceptron learning rule.

The network for this problem should have two-inputs and one output. To simplify our development of the learning rule, we will begin with a network without a bias. The network will then have just two parameters, $w_{1,1}$ and $w_{1,2}$, as shown in Figure 4.4.

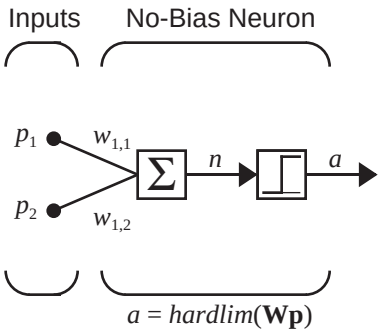
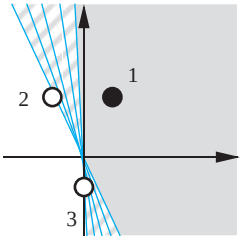
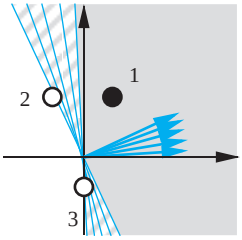


Figure 4.4 Test Problem Network



By removing the bias we are left with a network whose decision boundary must pass through the origin. We need to be sure that this network is still able to solve the test problem. There must be an allowable decision boundary that can separate the vectors $\mathbf{p}_2$ and $\mathbf{p}_3$ from the vector $\mathbf{p}_1$. The figure to the left illustrates that there are indeed an infinite number of such boundaries.



The adjacent figure shows the weight vectors that correspond to the allowable decision boundaries. (Recall that the weight vector is orthogonal to the decision boundary.) We would like a learning rule that will find a weight vector that points in one of these directions. Remember that the length of the weight vector does not matter; only its direction is important.

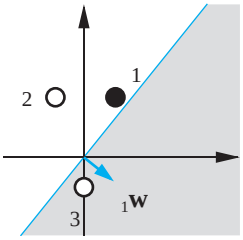
Constructing Learning Rules

Training begins by assigning some initial values for the network parameters. In this case we are training a two-input/single-output network without a bias, so we only have to initialize its two weights. Here we set the elements of the weight vector, $_1\mathbf{w}$, to the following randomly generated values:

$$_1\mathbf{w}^T = \begin{bmatrix} 1.0 & -0.8 \end{bmatrix}. \tag{4.21}$$

We will now begin presenting the input vectors to the network. We begin with $\mathbf{p}_1$:

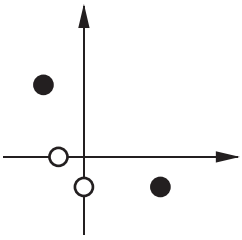
$$a = hardlim(_1\mathbf{w}^T\mathbf{p}_1) = hardlim\left(\begin{bmatrix} 1.0 & -0.8 \end{bmatrix}\begin{bmatrix} 1 \\ 2 \end{bmatrix}\right)$$
$$a = hardlim(-0.6) = 0. \tag{4.22}$$



The network has not returned the correct value. The network output is 0, while the target response, t_1, is 1.

We can see what happened by looking at the adjacent diagram. The initial weight vector results in a decision boundary that incorrectly classifies the vector $\mathbf{p}_1$. We need to alter the weight vector so that it points more toward $\mathbf{p}_1$, so that in the future it has a better chance of classifying it correctly.

One approach would be to set $_1\mathbf{w}$ equal to $\mathbf{p}_1$. This is simple and would ensure that $\mathbf{p}_1$ was classified properly in the future. Unfortunately, it is easy to construct a problem for which this rule cannot find a solution. The diagram to the lower left shows a problem that cannot be solved with the weight vector pointing directly at either of the two class 1 vectors. If we apply the rule $_1\mathbf{w} = \mathbf{p}$ every time one of these vectors is misclassified, the network's weights will simply oscillate back and forth and will never find a solution.

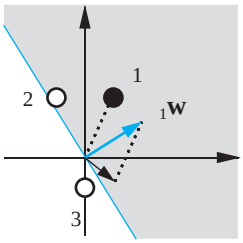


Another possibility would be to add $\mathbf{p}_1$ to $_1\mathbf{w}$. Adding $\mathbf{p}_1$ to $_1\mathbf{w}$ would make $_1\mathbf{w}$ point more in the direction of $\mathbf{p}_1$. Repeated presentations of $\mathbf{p}_1$ would cause the direction of $_1\mathbf{w}$ to asymptotically approach the direction of $\mathbf{p}_1$. This rule can be stated:

$$\text{If } t = 1 \text{ and } a = 0, \text{ then } _1\mathbf{w}^{new} = _1\mathbf{w}^{old} + \mathbf{p}. \tag{4.23}$$

Applying this rule to our test problem results in new values for ${}_1\mathbf{w}$:

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}_1 = \begin{bmatrix} 1.0 \\ -0.8 \end{bmatrix} + \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix}. \tag{4.24} $$



This operation is illustrated in the adjacent figure.

We now move on to the next input vector and will continue making changes to the weights and cycling through the inputs until they are all classified correctly.

The next input vector is $\mathbf{p}_2$. When it is presented to the network we find:

$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_2) = hardlim\left(\begin{bmatrix} 2.0 & 1.2 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} \right) \\ &= hardlim(0.4) = 1 . \end{aligned} \tag{4.25} $$

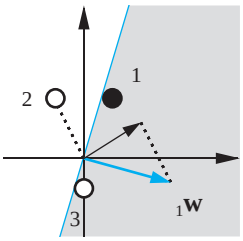
The target t_2 associated with $\mathbf{p}_2$ is 0 and the output a is 1. A class 0 vector was misclassified as a 1.

Since we would now like to move the weight vector ${}_1\mathbf{w}$ away from the input, we can simply change the addition in Eq. (4.23) to subtraction:

$$ \text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \tag{4.26} $$

If we apply this to the test problem we find:

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_2 = \begin{bmatrix} 2.0 \\ 1.2 \end{bmatrix} - \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix}, \tag{4.27} $$



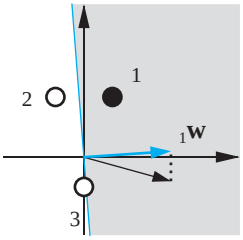
which is illustrated in the adjacent figure.

Now we present the third vector $\mathbf{p}_3$:

$$ \begin{aligned} a &= hardlim({}_1\mathbf{w}^T\mathbf{p}_3) = hardlim\left(\begin{bmatrix} 3.0 & -0.8 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \end{bmatrix} \right) \\ &= hardlim(0.8) = 1 . \end{aligned} \tag{4.28} $$

The current ${}_1\mathbf{w}$ results in a decision boundary that misclassifies $\mathbf{p}_3$. This is a situation for which we already have a rule, so ${}_1\mathbf{w}$ will be updated again, according to Eq. (4.26):

$$ {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}_3 = \begin{bmatrix} 3.0 \\ -0.8 \end{bmatrix} - \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 3.0 \\ 0.2 \end{bmatrix}. \tag{4.29} $$



The diagram to the left shows that the perceptron has finally learned to classify the three vectors properly. If we present any of the input vectors to the neuron, it will output the correct class for that input vector.

This brings us to our third and final rule: if it works, don't fix it.

$$\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \tag{4.30}$$

Here are the three rules, which cover all possible combinations of output and target values:

$$\begin{aligned} &\text{If } t = 1 \text{ and } a = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}. \\ &\text{If } t = 0 \text{ and } a = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \\ &\text{If } t = a, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \end{aligned} \tag{4.31}$$

Unified Learning Rule

The three rules in Eq. (4.31) can be rewritten as a single expression. First we will define a new variable, the perceptron error e:

$$e = t - a. \tag{4.32}$$

We can now rewrite the three rules of Eq. (4.31) as:

$$\begin{aligned} &\text{If } e = 1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + \mathbf{p}. \\ &\text{If } e = -1, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} - \mathbf{p}. \\ &\text{If } e = 0, \text{ then } {}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old}. \end{aligned} \tag{4.33}$$

Looking carefully at the first two rules in Eq. (4.33) we can see that the sign of $\mathbf{p}$ is the same as the sign on the error, e. Furthermore, the absence of $\mathbf{p}$ in the third rule corresponds to an e of 0. Thus, we can unify the three rules into a single expression:

$${}_1\mathbf{w}^{new} = {}_1\mathbf{w}^{old} + e\mathbf{p} = {}_1\mathbf{w}^{old} + (t - a)\mathbf{p}. \tag{4.34}$$

This rule can be extended to train the bias by noting that a bias is simply a weight whose input is always 1. We can thus replace the input $\mathbf{p}$ in Eq. (4.34) with the input to the bias, which is 1. The result is the perceptron rule for a bias:

$$b^{new} = b^{old} + e. \tag{4.35}$$

Training Multiple-Neuron Perceptrons

The perceptron rule, as given by Eq. (4.34) and Eq. (4.35), updates the weight vector of a single neuron perceptron. We can generalize this rule for the multiple-neuron perceptron of Figure 4.1 as follows. To update the ith row of the weight matrix use:

$$ {}_i\mathbf{w}^{new} = {}_i\mathbf{w}^{old} + e_i\mathbf{p} . \tag{4.36}$$

To update the ith element of the bias vector use:

$$ b_i^{new} = b_i^{old} + e_i . \tag{4.37}$$

Perceptron Rule

The *perceptron rule* can be written conveniently in matrix notation:

$$ \mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T , \tag{4.38}$$

and

$$ \mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e} . \tag{4.39}$$

To test the perceptron learning rule, consider again the apple/orange recognition problem of Chapter 3. The input/output prototype vectors will be

$$ \left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, t_1 = \begin{bmatrix} 0 \end{bmatrix} \right\} \qquad \left\{ \mathbf{p}_2 = \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix}, t_2 = \begin{bmatrix} 1 \end{bmatrix} \right\} . \tag{4.40}$$

(Note that we are using 0 as the target output for the orange pattern, $\mathbf{p}_1$, instead of -1, as was used in Chapter 3. This is because we are using the *hardlim* transfer function, instead of *hardlims*.)

Typically the weights and biases are initialized to small random numbers. Suppose that here we start with the initial weight matrix and bias:

$$ \mathbf{W} = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} , \; b = 0.5 . \tag{4.41}$$

The first step is to apply the first input vector, $\mathbf{p}_1$, to the network:

$$ \begin{aligned} a &= hardlim(\mathbf{W}\mathbf{p}_1 + b) = hardlim\left(\begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5 \right) \\ &= hardlim(2.5) = 1 \end{aligned} \tag{4.42}$$

Then we calculate the error:

$$e = t_1 - a = 0 - 1 = -1 . \tag{4.43}$$

The weight update is

$$\begin{aligned} \mathbf{W}^{new} &= \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & -1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} . \end{aligned} \tag{4.44}$$

The bias update is

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 . \tag{4.45}$$

This completes the first iteration.

The second iteration of the perceptron rule is:

$$\begin{aligned} a &= hardlim\,(\mathbf{W}\mathbf{p}_2 + b) = hardlim\,(\begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} + (-0.5)) \\ &= hardlim\,(-0.5) = 0 \end{aligned} \tag{4.46}$$

$$e = t_2 - a = 1 - 0 = 1 \tag{4.47}$$

$$\mathbf{W}^{new} = \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} -0.5 & 0 & 0.5 \end{bmatrix} + 1\begin{bmatrix} 1 & 1 & -1 \end{bmatrix} = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} \tag{4.48}$$

$$b^{new} = b^{old} + e = -0.5 + 1 = 0.5 \tag{4.49}$$

The third iteration begins again with the first input vector:

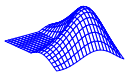
$$\begin{aligned} a &= hardlim\,(\mathbf{W}\mathbf{p}_1 + b) = hardlim\,(\begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix}\begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix} + 0.5) \\ &= hardlim\,(0.5) = 1 \end{aligned} \tag{4.50}$$

$$e = t_1 - a = 0 - 1 = -1 \tag{4.51}$$

$$\begin{aligned} \mathbf{W}^{new} &= \mathbf{W}^{old} + e\mathbf{p}^T = \begin{bmatrix} 0.5 & 1 & -0.5 \end{bmatrix} + (-1)\begin{bmatrix} 1 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} -0.5 & 2 & 0.5 \end{bmatrix} \end{aligned} \tag{4.52}$$

$$b^{new} = b^{old} + e = 0.5 + (-1) = -0.5 . \tag{4.53}$$

If you continue with the iterations you will find that both input vectors will now be correctly classified. The algorithm has converged to a solution. Note that the final decision boundary is not the same as the one we developed in Chapter 3, although both boundaries correctly classify the two input vectors.



To experiment with the perceptron learning rule, use the Neural Network Design Demonstration Perceptron Rule (nnd4pr).

Proof of Convergence

Although the perceptron learning rule is simple, it is quite powerful. In fact, it can be shown that the rule will always converge to weights that accomplish the desired classification (assuming that such weights exist). In this section we will present a proof of convergence for the perceptron learning rule for the single-neuron perceptron shown in Figure 4.5.

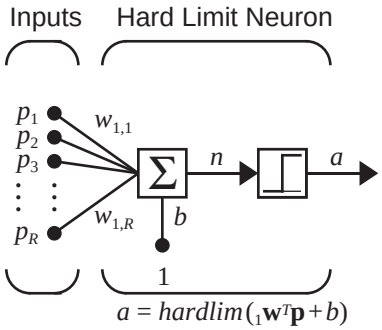


Figure 4.5 Single-Neuron Perceptron

The output of this perceptron is obtained from

$$a = hardlim(_1\mathbf{w}^T\mathbf{p} + b) . \tag{4.54}$$

The network is provided with the following examples of proper network behavior:

$$\{\mathbf{p}_1, t_1\}, \{\mathbf{p}_2, t_2\}, \ldots, \{\mathbf{p}_Q, t_Q\} . \tag{4.55}$$

where each target output, t_q, is either 0 or 1.

Notation

To conveniently present the proof we will first introduce some new notation. We will combine the weight matrix and the bias into a single vector:

$$\mathbf{x} = \begin{bmatrix} {}_1\mathbf{w} \\ b \end{bmatrix}. \tag{4.56}$$

We will also augment the input vectors with a 1, corresponding to the bias input:

$$\mathbf{z}_q = \begin{bmatrix} \mathbf{p}_q \\ 1 \end{bmatrix}. \tag{4.57}$$

Now we can express the net input to the neuron as follows:

$$n = {}_1\mathbf{w}^T\mathbf{p} + b = \mathbf{x}^T\mathbf{z}. \tag{4.58}$$

The perceptron learning rule for a single-neuron perceptron (Eq. (4.34) and Eq. (4.35)) can now be written

$$\mathbf{x}^{new} = \mathbf{x}^{old} + e\mathbf{z}. \tag{4.59}$$

The error e can be either 1, -1 or 0. If $e = 0$, then no change is made to the weights. If $e = 1$, then the input vector is added to the weight vector. If $e = -1$, then the negative of the input vector is added to the weight vector. If we count only those iterations for which the weight vector is changed, the learning rule becomes

$$\mathbf{x}(k) = \mathbf{x}(k-1) + \mathbf{z}'(k-1) \quad , \tag{4.60}$$

where $\mathbf{z}'(k-1)$ is the appropriate member of the set

$$\{\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_Q, -\mathbf{z}_1, -\mathbf{z}_2, \ldots, -\mathbf{z}_Q\}. \tag{4.61}$$

We will assume that a weight vector exists that can correctly categorize all Q input vectors. This solution will be denoted $\mathbf{x}^*$. For this weight vector we will assume that

$$\mathbf{x}^{*T}\mathbf{z}_q > \delta > 0 \text{ if } t_q = 1, \tag{4.62}$$

and

$$\mathbf{x}^{*T}\mathbf{z}_q < -\delta < 0 \text{ if } t_q = 0. \tag{4.63}$$

Proof

We are now ready to begin the proof of the perceptron convergence theorem. The objective of the proof is to find upper and lower bounds on the length of the weight vector at each stage of the algorithm.

Assume that the algorithm is initialized with the zero weight vector: $\mathbf{x}(0) = \mathbf{0}$. (This does not affect the generality of our argument.) Then, after k iterations (changes to the weight vector), we find from Eq. (4.60):

$$\mathbf{x}(k) = \mathbf{z}'(0) + \mathbf{z}'(1) + \cdots + \mathbf{z}'(k-1) \quad . \tag{4.64}$$

If we take the inner product of the solution weight vector with the weight vector at iteration k we obtain

$$\mathbf{x}^{*T}\mathbf{x}(k) = \mathbf{x}^{*T}\mathbf{z}'(0) + \mathbf{x}^{*T}\mathbf{z}'(1) + \cdots + \mathbf{x}^{*T}\mathbf{z}'(k-1) \quad . \tag{4.65}$$

From Eq. (4.61)–Eq. (4.63) we can show that

$$\mathbf{x}^{*T}\mathbf{z}'(i) > \delta \; . \tag{4.66}$$

Therefore

$$\mathbf{x}^{*T}\mathbf{x}(k) > k\delta \; . \tag{4.67}$$

From the Cauchy-Schwartz inequality (see [Brog91])

$$(\mathbf{x}^{*T}\mathbf{x}(k))^2 \le \|\mathbf{x}^*\|^2 \|\mathbf{x}(k)\|^2 \quad , \tag{4.68}$$

where

$$\|\mathbf{x}\|^2 = \mathbf{x}^T\mathbf{x} \, . \tag{4.69}$$

If we combine Eq. (4.67) and Eq. (4.68) we can put a lower bound on the squared length of the weight vector at iteration k:

$$\|\mathbf{x}(k)\|^2 \ge \frac{(\mathbf{x}^{*T}\mathbf{x}(k))^2}{\|\mathbf{x}^*\|^2} > \frac{(k\delta)^2}{\|\mathbf{x}^*\|^2} \quad . \tag{4.70}$$

Next we want to find an upper bound for the length of the weight vector. We begin by finding the change in the length at iteration k:

$$\begin{aligned} \|\mathbf{x}(k)\|^2 &= \mathbf{x}^T(k)\mathbf{x}(k) \\ &= [\mathbf{x}(k-1) + \mathbf{z}'(k-1)]^T[\mathbf{x}(k-1) + \mathbf{z}'(k-1)] \\ &= \mathbf{x}^T(k-1)\mathbf{x}(k-1) + 2\mathbf{x}^T(k-1)\mathbf{z}'(k-1) \\ &\quad + \mathbf{z}'^T(k-1)\mathbf{z}'(k-1) \end{aligned} \tag{4.71}$$

Note that

$$\mathbf{x}^T(k-1)\mathbf{z}'(k-1) \leq 0 \ , \tag{4.72}$$

since the weights would not be updated unless the previous input vector had been misclassified. Now Eq. (4.71) can be simplified to

$$\|\mathbf{x}(k)\|^2 \leq \|\mathbf{x}(k-1)\|^2 + \|\mathbf{z}'(k-1)\|^2 \ . \tag{4.73}$$

We can repeat this process for $\|\mathbf{x}(k-1)\|^2$, $\|\mathbf{x}(k-2)\|^2$, etc., to obtain

$$\|\mathbf{x}(k)\|^2 \leq \|\mathbf{z}'(0)\|^2 + \cdots + \|\mathbf{z}'(k-1)\|^2 \ . \tag{4.74}$$

If $\Pi = max\{\|\mathbf{z}'(i)\|^2\}$, this upper bound can be simplified to

$$\|\mathbf{x}(k)\|^2 \leq k\Pi \ . \tag{4.75}$$

We now have an upper bound (Eq. (4.75)) and a lower bound (Eq. (4.70)) on the squared length of the weight vector at iteration k . If we combine the two inequalities we find

$$k\Pi \geq \|\mathbf{x}(k)\|^2 > \frac{(k\delta)^2}{\|\mathbf{x}^*\|^2} \quad \text{or} \quad k < \frac{\Pi\|\mathbf{x}^*\|^2}{\delta^2} \ . \tag{4.76}$$

Because k has an upper bound, this means that the weights will only be changed a finite number of times. Therefore, the perceptron learning rule will converge in a finite number of iterations.

The maximum number of iterations (changes to the weight vector) is inversely related to the square of δ . This parameter is a measure of how close the solution decision boundary is to the input patterns. This means that if the input classes are difficult to separate (are close to the decision boundary) it will take many iterations for the algorithm to converge.

Note that there are only three key assumptions required for the proof:

1. A solution to the problem exists, so that Eq. (4.66) is satisfied.
2. The weights are only updated when the input vector is misclassified, therefore Eq. (4.72) is satisfied.
3. An upper bound, Π , exists for the length of the input vectors.

Because of the generality of the proof, there are many variations of the perceptron learning rule that can also be shown to converge. (See Exercise E4.9.)

Limitations

The perceptron learning rule is guaranteed to converge to a solution in a finite number of steps, so long as a solution exists. This brings us to an im-

portant question. What problems can a perceptron solve? Recall that a single-neuron perceptron is able to divide the input space into two regions. The boundary between the regions is defined by the equation

$$_1\mathbf{w}^T\mathbf{p} + b = 0. \tag{4.77}$$

Linear Separability

This is a linear boundary (hyperplane). The perceptron can be used to classify input vectors that can be separated by a linear boundary. We call such vectors *linearly separable*. The logical AND gate example on page 4-7 illustrates a two-dimensional example of a linearly separable problem. The apple/orange recognition problem of Chapter 3 was a three-dimensional example.

Unfortunately, many problems are not linearly separable. The classic example is the XOR gate. The input/target pairs for the XOR gate are

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\0\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\1\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\0\end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\1\end{bmatrix}, t_4 = 0\right\}.$$

This problem is illustrated graphically on the left side of Figure 4.6, which also shows two other linearly inseparable problems. Try drawing a straight line between the vectors with targets of 1 and those with targets of 0 in any of the diagrams of Figure 4.6.

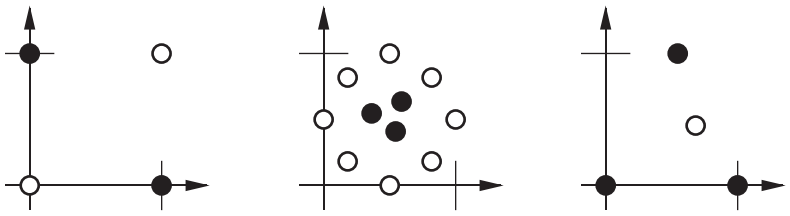
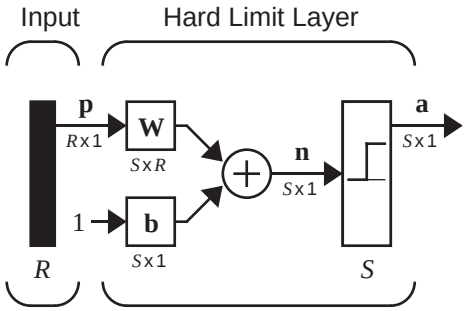


Figure 4.6 Linearly Inseparable Problems

It was the inability of the basic perceptron to solve such simple problems that led, in part, to a reduction in interest in neural network research during the 1970s. Rosenblatt had investigated more complex networks, which he felt would overcome the limitations of the basic perceptron, but he was never able to effectively extend the perceptron rule to such networks. In Chapter 11 we will introduce multilayer perceptrons, which can solve arbitrary classification problems, and will describe the backpropagation algorithm, which can be used to train them.

Summary of Results

Perceptron Architecture



$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp}+\mathbf{b})$$

$$\mathbf{a} = \mathbf{hardlim}(\mathbf{Wp} + \mathbf{b}) \qquad \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \\ \vdots \\ {}_S\mathbf{w}^T \end{bmatrix}$$

$$a_i = hardlim(n_i) = hardlim({}_i\mathbf{w}^T\mathbf{p} + b_i)$$

Decision Boundary

$${}_i\mathbf{w}^T\mathbf{p} + b_i = 0 .$$

The decision boundary is always orthogonal to the weight vector.

Single-layer perceptrons can only classify linearly separable vectors.

Perceptron Learning Rule

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \mathbf{e}$$

where $\mathbf{e} = \mathbf{t} - \mathbf{a}$.

Solved Problems

P4.1 Solve the three simple classification problems shown in Figure P4.1 by drawing a decision boundary. Find weight and bias values that result in single-neuron perceptrons with the chosen decision boundaries.

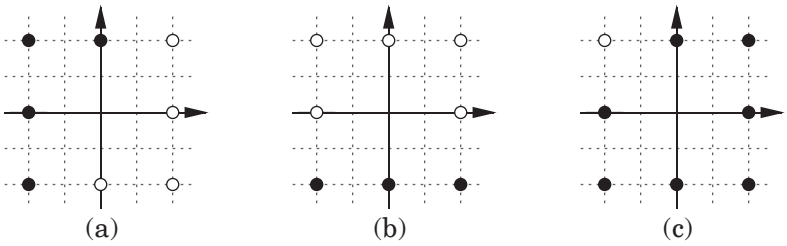
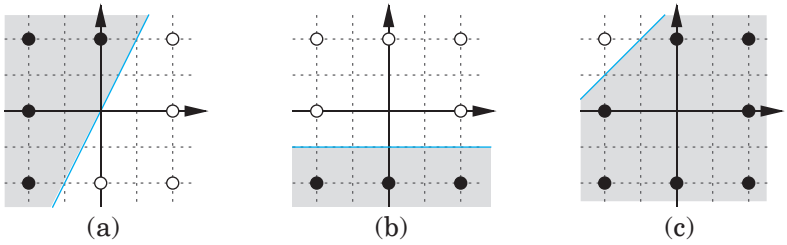
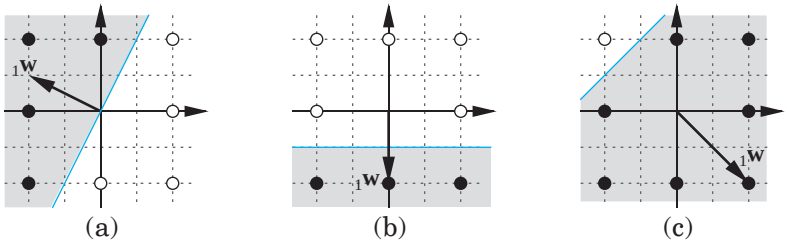


Figure P4.1 Simple Classification Problems

First we draw a line between each set of dark and light data points.



The next step is to find the weights and biases. The weight vectors must be orthogonal to the decision boundaries, and pointing in the direction of points to be classified as 1 (the dark points). The weight vectors can have any length we like.



Here is one set of choices for the weight vectors:

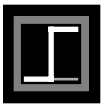
$$\text{(a) } {}_1\mathbf{w}^T = \begin{bmatrix} -2 & 1 \end{bmatrix}, \quad \text{(b) } {}_1\mathbf{w}^T = \begin{bmatrix} 0 & -2 \end{bmatrix}, \quad \text{(c) } {}_1\mathbf{w}^T = \begin{bmatrix} 2 & -2 \end{bmatrix}.$$

Now we find the bias values for each perceptron by picking a point on the decision boundary and satisfying Eq. (4.15).

$$
\begin{aligned}
{}_1\mathbf{w}^T\mathbf{p} + b &= 0 \\
b &= -{}_1\mathbf{w}^T\mathbf{p}
\end{aligned}
$$

This gives us the following three biases:

$$(a)\ b = -\begin{bmatrix}-2 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} = 0, (b)\ b = -\begin{bmatrix}0 & -2\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix} = -2, (c)\ b = -\begin{bmatrix}2 & -2\end{bmatrix}\begin{bmatrix}-2\\1\end{bmatrix} = 6$$



We can now check our solution against the original points. Here we test the first network on the input vector $\mathbf{p} = \begin{bmatrix}-2 & 2\end{bmatrix}^T$.

$$
\begin{aligned}
a &= hardlim({}_1\mathbf{w}^T\mathbf{p} + b) \\
&= hardlim\left(\begin{bmatrix}-2 & 1\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} + 0\right) \\
&= hardlim(6) \\
&= 1
\end{aligned}
$$

We can use MATLAB to automate the testing process and to try new points. Here the first network is used to classify a point that was not in the original problem.

```
w=[-2 1]; b = 0;
a = hardlim(w*[1;1]+b)
a =
       0
```

P4.2 Convert the classification problem defined below into an equivalent problem definition consisting of inequalities constraining weight and bias values.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}0\\2\end{bmatrix}, t_1 = 1\right\}\ \left\{\mathbf{p}_2 = \begin{bmatrix}1\\0\end{bmatrix}, t_2 = 1\right\}\ \left\{\mathbf{p}_3 = \begin{bmatrix}0\\-2\end{bmatrix}, t_3 = 0\right\}\ \left\{\mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}, t_4 = 0\right\}$$

Each target t_i indicates whether or not the net input in response to $\mathbf{p}_i$ must be less than 0, or greater than or equal to 0. For example, since t_1 is 1, we

know that the net input corresponding to $\mathbf{p}_1$ must be greater than or equal to 0. Thus we get the following inequality:

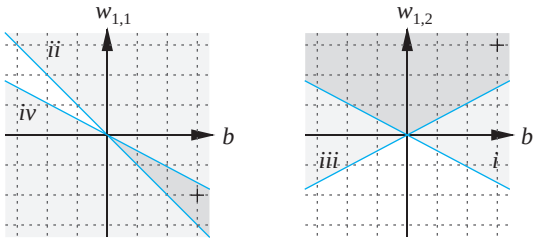
$$\begin{aligned} \mathbf{W}\mathbf{p}_1 + b &\geq 0 \\ 0w_{1,1} + 2w_{1,2} + b &\geq 0 \\ 2w_{1,2} + b &\geq 0. \end{aligned}$$

Applying the same procedure to the input/target pairs for $\{\mathbf{p}_2, t_2\}$, $\{\mathbf{p}_3, t_3\}$ and $\{\mathbf{p}_4, t_4\}$ results in the following set of inequalities.

$$\begin{aligned} 2w_{1,2} + b &\geq 0 \quad (i) \\ w_{1,1} + b &\geq 0 \quad (ii) \\ -2w_{1,2} + b &< 0 \quad (iii) \\ 2w_{1,1} + b &< 0 \quad (iv) \end{aligned}$$

Solving a set of inequalities is more difficult than solving a set of equalities. One added complexity is that there are often an infinite number of solutions (just as there are often an infinite number of linear decision boundaries that can solve a linearly separable classification problem).

However, because of the simplicity of this problem, we can solve it by graphing the solution spaces defined by the inequalities. Note that $w_{1,1}$ only appears in inequalities (*ii*) and (*iv*), and $w_{1,2}$ only appears in inequalities (*i*) and (*iii*). We can plot each pair of inequalities with two graphs.



Any weight and bias values that fall in both dark gray regions will solve the classification problem.

Here is one such solution:

$$\mathbf{W} = \begin{bmatrix} -2 & 3 \end{bmatrix} \qquad b = 3.$$

P4.3 We have a classification problem with four classes of input vector. The four classes are

$$\text{class 1:} \left\{ \mathbf{p}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{p}_2 = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \right\}, \text{class 2:} \left\{ \mathbf{p}_3 = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \mathbf{p}_4 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \right\},$$

$$\text{class 3:} \left\{ \mathbf{p}_5 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \mathbf{p}_6 = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \right\}, \text{class 4:} \left\{ \mathbf{p}_7 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \mathbf{p}_8 = \begin{bmatrix} -2 \\ -2 \end{bmatrix} \right\}.$$

Design a perceptron network to solve this problem.

To solve a problem with four classes of input vector we will need a perceptron with at least two neurons, since an S-neuron perceptron can categorize 2^S classes. The two-neuron perceptron is shown in Figure P4.2.

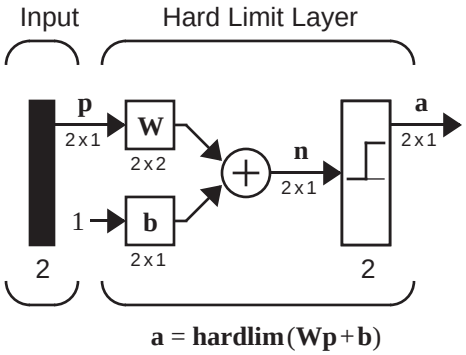


Figure P4.2 Two-Neuron Perceptron

Let's begin by displaying the input vectors, as in Figure P4.3. The light circles ○ indicate class 1 vectors, the light squares □ indicate class 2 vectors, the dark circles ● indicate class 3 vectors, and the dark squares ■ indicate class 4 vectors.

A two-neuron perceptron creates two decision boundaries. Therefore, to divide the input space into the four categories, we need to have one decision boundary divide the four classes into two sets of two. The remaining boundary must then isolate each class. Two such boundaries are illustrated in Figure P4.4. We now know that our patterns are linearly separable.

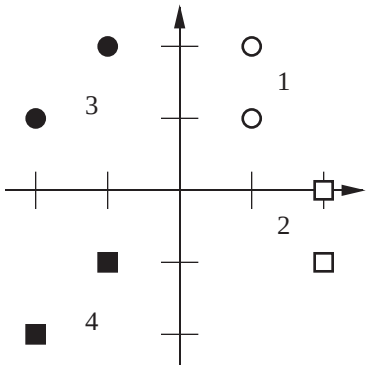


Figure P4.3 Input Vectors for Problem P4.3

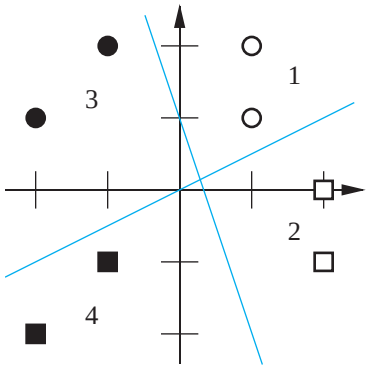


Figure P4.4 Tentative Decision Boundaries for Problem P4.3

The weight vectors should be orthogonal to the decision boundaries and should point toward the regions where the neuron outputs are 1. The next step is to decide which side of each boundary should produce a 1. One choice is illustrated in Figure P4.5, where the shaded areas represent outputs of 1. The darkest shading indicates that both neuron outputs are 1. Note that this solution corresponds to target values of

$$\text{class 1:} \left\{ \mathbf{t}_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \right\}, \text{ class 2: } \left\{ \mathbf{t}_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\},$$

$$\text{class 3:} \left\{ \mathbf{t}_5 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \text{ class 4: } \left\{ \mathbf{t}_7 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}.$$

We can now select the weight vectors:

$$ {}_1\mathbf{w} = \begin{bmatrix} -3 \\ -1 \end{bmatrix} \text{ and } {}_2\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}. $$

Note that the lengths of the weight vectors is not important, only their directions. They must be orthogonal to the decision boundaries. Now we can calculate the bias by picking a point on a boundary and satisfying Eq. (4.15):

$$ b_1 = -{}_1\mathbf{w}^T\mathbf{p} = -\begin{bmatrix} -3 & -1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1, $$

$$ b_2 = -{}_2\mathbf{w}^T\mathbf{p} = -\begin{bmatrix} 1 & -2 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix} = 0. $$

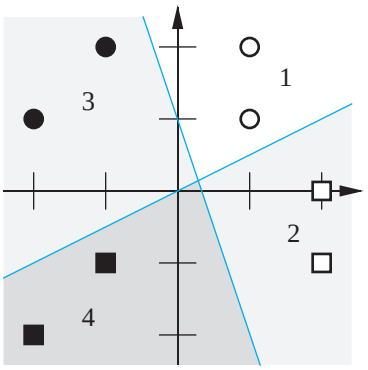


Figure P4.5 Decision Regions for Problem P4.3

In matrix form we have

$$ \mathbf{W} = \begin{bmatrix} {}_1\mathbf{w}^T \\ {}_2\mathbf{w}^T \end{bmatrix} = \begin{bmatrix} -3 & -1 \\ 1 & -2 \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, $$

which completes our design.

P4.4 Solve the following classification problem with the perceptron rule. Apply each input vector in order, for as many repetitions as it takes to ensure that the problem is solved. Draw a graph of the problem only after you have found a solution.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}2\\2\end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\-2\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}-2\\2\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}-1\\1\end{bmatrix}, t_4 = 1\right\}$$

Use the initial weights and bias:

$$\mathbf{W}(0) = \begin{bmatrix}0 & 0\end{bmatrix} \qquad b(0) = 0 \ .$$

We start by calculating the perceptron's output a for the first input vector $\mathbf{p}_1$, using the initial weights and bias.

$$a = hardlim(\mathbf{W}(0)\mathbf{p}_1 + b(0))$$

$$= hardlim\left(\begin{bmatrix}0 & 0\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(0) = 1$$

The output a does not equal the target value t_1, so we use the perceptron rule to find new weights and biases based on the error.

$$e = t_1 - a = 0 - 1 = -1$$
$$\mathbf{W}(1) = \mathbf{W}(0) + e\mathbf{p}_1^T = \begin{bmatrix}0 & 0\end{bmatrix} + (-1)\begin{bmatrix}2 & 2\end{bmatrix} = \begin{bmatrix}-2 & -2\end{bmatrix}$$
$$b(1) = b(0) + e = 0 + (-1) = -1$$

We now apply the second input vector $\mathbf{p}_2$, using the updated weights and bias.

$$a = hardlim(\mathbf{W}(1)\mathbf{p}_2 + b(1))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} - 1\right) = hardlim(1) = 1$$

This time the output a is equal to the target t_2. Application of the perceptron rule will not result in any changes.

$$\mathbf{W}(2) = \mathbf{W}(1)$$
$$b(2) = b(1)$$

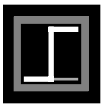
We now apply the third input vector.

$$a = hardlim(\mathbf{W}(2)\mathbf{p}_3 + b(2))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} - 1\right) = hardlim(-1) = 0$$

The output in response to input vector $\mathbf{p}_3$ is equal to the target t_3, so there will be no changes.

$$\mathbf{W}(3) = \mathbf{W}(2)$$
$$b(3) = b(2)$$



We now move on to the last input vector $\mathbf{p}_4$.

$$a = hardlim(\mathbf{W}(3)\mathbf{p}_4 + b(3))$$

$$= hardlim\left(\begin{bmatrix}-2 & -2\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix} - 1\right) = hardlim(-1) = 0$$

This time the output a does not equal the appropriate target t_4. The perceptron rule will result in a new set of values for $\mathbf{W}$ and b.

$$e = t_4 - a = 1 - 0 = 1$$
$$\mathbf{W}(4) = \mathbf{W}(3) + e\mathbf{p}_4^T = \begin{bmatrix}-2 & -2\end{bmatrix} + (1)\begin{bmatrix}-1 & 1\end{bmatrix} = \begin{bmatrix}-3 & -1\end{bmatrix}$$
$$b(4) = b(3) + e = -1 + 1 = 0$$

We now must check the first vector $\mathbf{p}_1$ again. This time the output a is equal to the associated target t_1.

$$a = hardlim(\mathbf{W}(4)\mathbf{p}_1 + b(4))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 0\right) = hardlim(-8) = 0$$

Therefore there are no changes.

$$\mathbf{W}(5) = \mathbf{W}(4)$$
$$b(5) = b(4)$$

The second presentation of $\mathbf{p}_2$ results in an error and therefore a new set of weight and bias values.

$$a = hardlim(\mathbf{W}(5)\mathbf{p}_2 + b(5))$$

$$= hardlim\left(\begin{bmatrix}-3 & -1\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} + 0\right) = hardlim(-1) = 0$$

Here are those new values:

$$e = t_2 - a = 1 - 0 = 1$$
$$\mathbf{W}(6) = \mathbf{W}(5) + e\mathbf{p}_2^T = \begin{bmatrix}-3 & -1\end{bmatrix} + (1)\begin{bmatrix}1 & -2\end{bmatrix} = \begin{bmatrix}-2 & -3\end{bmatrix}$$
$$b(6) = b(5) + e = 0 + 1 = 1.$$

Cycling through each input vector once more results in no errors.

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_3 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}-2\\2\end{bmatrix} + 1\right) = 0 = t_3$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_4 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}-1\\1\end{bmatrix} + 1\right) = 1 = t_4$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_1 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}2\\2\end{bmatrix} + 1\right) = 0 = t_1$$

$$a = hardlim(\mathbf{W}(6)\mathbf{p}_2 + b(6)) = hardlim\left(\begin{bmatrix}-2 & -3\end{bmatrix}\begin{bmatrix}1\\-2\end{bmatrix} + 1\right) = 1 = t_2$$

Therefore the algorithm has converged. The final solution is:

$$\mathbf{W} = \begin{bmatrix}-2 & -3\end{bmatrix} \qquad b = 1.$$

Now we can graph the training data and the decision boundary of the solution. The decision boundary is given by

$$n = \mathbf{W}\mathbf{p} + b = w_{1,1}p_1 + w_{1,2}p_2 + b = -2p_1 - 3p_2 + 1 = 0.$$

To find the p_2 intercept of the decision boundary, set $p_1 = 0$:

$$p_2 = -\frac{b}{w_{1,2}} = -\frac{1}{-3} = \frac{1}{3} \quad \text{if } p_1 = 0.$$

To find the p_1 intercept, set $p_2 = 0$:

$$p_1 = -\frac{b}{w_{1,1}} = -\frac{1}{-2} = \frac{1}{2} \quad \text{if } p_2 = 0.$$

The resulting decision boundary is illustrated in Figure P4.6.

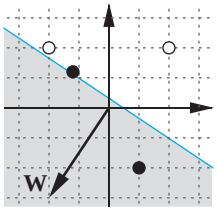


Figure P4.6 Decision Boundary for Problem P4.4

Note that the decision boundary falls across one of the training vectors. This is acceptable, given the problem definition, since the hard limit function returns 1 when given an input of 0, and the target for the vector in question is indeed 1.

P4.5 Consider again the four-class decision problem that we introduced in Problem P4.3. Train a perceptron network to solve this problem using the perceptron learning rule.

If we use the same target vectors that we introduced in Problem P4.3, the training set will be:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}1\\1\end{bmatrix}, \mathbf{t}_1 = \begin{bmatrix}0\\0\end{bmatrix}\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}1\\2\end{bmatrix}, \mathbf{t}_2 = \begin{bmatrix}0\\0\end{bmatrix}\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}2\\-1\end{bmatrix}, \mathbf{t}_3 = \begin{bmatrix}0\\1\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_4 = \begin{bmatrix}2\\0\end{bmatrix}, \mathbf{t}_4 = \begin{bmatrix}0\\1\end{bmatrix}\right\} \left\{\mathbf{p}_5 = \begin{bmatrix}-1\\2\end{bmatrix}, \mathbf{t}_5 = \begin{bmatrix}1\\0\end{bmatrix}\right\} \left\{\mathbf{p}_6 = \begin{bmatrix}-2\\1\end{bmatrix}, \mathbf{t}_6 = \begin{bmatrix}1\\0\end{bmatrix}\right\}$$

$$\left\{\mathbf{p}_7 = \begin{bmatrix}-1\\-1\end{bmatrix}, \mathbf{t}_7 = \begin{bmatrix}1\\1\end{bmatrix}\right\} \left\{\mathbf{p}_8 = \begin{bmatrix}-2\\-2\end{bmatrix}, \mathbf{t}_8 = \begin{bmatrix}1\\1\end{bmatrix}\right\}.$$

Let's begin the algorithm with the following initial weights and biases:

$$\mathbf{W}(0) = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}, \quad \mathbf{b}(0) = \begin{bmatrix}1\\1\end{bmatrix}.$$

The first iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(0)\mathbf{p}_1 + \mathbf{b}(0)) = hardlim\,(\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}1\\1\end{bmatrix}) = \begin{bmatrix}1\\1\end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix},$$

$$\mathbf{W}(1) = \mathbf{W}(0) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix},$$

$$\mathbf{b}(1) = \mathbf{b}(0) + \mathbf{e} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The second iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(1)\mathbf{p}_2 + \mathbf{b}(1)) = hardlim\,(\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_2 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\mathbf{W}(2) = \mathbf{W}(1) + \mathbf{e}\mathbf{p}_2^T = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix},$$

$$\mathbf{b}(2) = \mathbf{b}(1) + \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The third iteration is

$$\mathbf{a} = hardlim\,(\mathbf{W}(2)\mathbf{p}_3 + \mathbf{b}(2)) = hardlim\,(\begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix},$$

$$\mathbf{e} = \mathbf{t}_3 - \mathbf{a} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} - \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix},$$

$$\mathbf{W}(3) = \mathbf{W}(2) + \mathbf{e}\mathbf{p}_3^T = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} 2 & -1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix},$$

$$\mathbf{b}(3) = \mathbf{b}(2) + \mathbf{e} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} .$$

Iterations four through eight produce no changes in the weights.

$$\mathbf{W}(8) = \mathbf{W}(7) = \mathbf{W}(6) = \mathbf{W}(5) = \mathbf{W}(4) = \mathbf{W}(3)$$

$$\mathbf{b}(8) = \mathbf{b}(7) = \mathbf{b}(6) = \mathbf{b}(5) = \mathbf{b}(4) = \mathbf{b}(3)$$

The ninth iteration produces

$$\mathbf{a} = hardlim\,(\mathbf{W}(8)\mathbf{p}_1 + \mathbf{b}(8)) = hardlim\,(\begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \end{bmatrix}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} ,$$

$$\mathbf{e} = \mathbf{t}_1 - \mathbf{a} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\mathbf{W}(9) = \mathbf{W}(8) + \mathbf{e}\mathbf{p}_1^T = \begin{bmatrix} -2 & 0 \\ 1 & -1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix} ,$$

$$\mathbf{b}(9) = \mathbf{b}(8) + \mathbf{e} = \begin{bmatrix} -1 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ -1 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \end{bmatrix} .$$

At this point the algorithm has converged, since all input patterns will be correctly classified. The final decision boundaries are displayed in Figure P4.7. Compare this result with the network we designed in Problem P4.3.

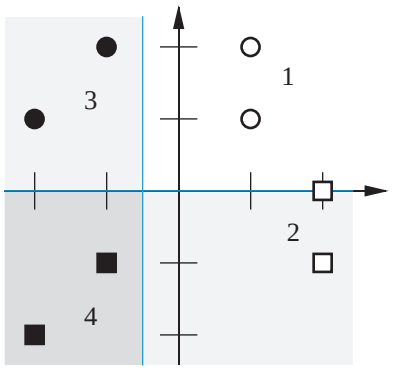


Figure P4.7 Final Decision Boundaries for Problem P4.5

Epilogue

In this chapter we have introduced our first learning rule — the perceptron learning rule. It is a type of learning called *supervised learning*, in which the learning rule is provided with a set of examples of proper network behavior. As each input is applied to the network, the learning rule adjusts the network parameters so that the network output will move closer to the target.

The perceptron learning rule is very simple, but it is also quite powerful. We have shown that the rule will always converge to a correct solution, if such a solution exists. The weakness of the perceptron network lies not with the learning rule, but with the structure of the network. The standard perceptron is only able to classify vectors that are linearly separable. We will see in Chapter 11 that the perceptron architecture can be generalized to mutlilayer perceptrons, which can solve arbitrary classification problems. The backpropagation learning rule, which is introduced in Chapter 11, can be used to train these networks.

In Chapters 3 and 4 we have used many concepts from the field of linear algebra, such as inner product, projection, distance (norm), etc. We will find in later chapters that a good foundation in linear algebra is essential to our understanding of all neural networks. In Chapters 5 and 6 we will review some of the key concepts from linear algebra that will be most important in our study of neural networks. Our objective will be to obtain a fundamental understanding of how neural networks work.

Further Reading

[BaSu83] A. Barto, R. Sutton and C. Anderson, "Neuron-like adaptive elements can solve difficult learning control problems," *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 13, No. 5, pp. 834–846, 1983.

A classic paper in which a reinforcement learning algorithm is used to train a neural network to balance an inverted pendulum.

[Brog91] W. L. Brogan, *Modern Control Theory*, 3rd Ed., Englewood Cliffs, NJ: Prentice-Hall, 1991.

A well-written book on the subject of linear systems. The first half of the book is devoted to linear algebra. It also has good sections on the solution of linear differential equations and the stability of linear and nonlinear systems. It has many worked problems.

[McPi43] W. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of Mathematical Biophysics*, Vol. 5, pp. 115–133, 1943.

This article introduces the first mathematical model of a neuron, in which a weighted sum of input signals is compared to a threshold to determine whether or not the neuron fires.

[MiPa69] M. Minsky and S. Papert, *Perceptrons*, Cambridge, MA: MIT Press, 1969.

A landmark book that contains the first rigorous study devoted to determining what a perceptron network is capable of learning. A formal treatment of the perceptron was needed both to explain the perceptron's limitations and to indicate directions for overcoming them. Unfortunately, the book pessimistically predicted that the limitations of perceptrons indicated that the field of neural networks was a dead end. Although this was not true, it temporarily cooled research and funding for research for several years.

[Rose58] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, Vol. 65, pp. 386–408, 1958.

This paper presents the first practical artificial neural network — the perceptron.

[Rose61] F. Rosenblatt, *Principles of Neurodynamics*, Washington DC: Spartan Press, 1961.

One of the first books on neurocomputing.

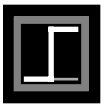
[WhSo92] D. White and D. Sofge (Eds.), *Handbook of Intelligent Control*, New York: Van Nostrand Reinhold, 1992.

Collection of articles describing current research and applications of neural networks and fuzzy logic to control systems.

Exercises

E4.1 Consider the classification problem defined below:

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = 1\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}0\\0\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}1\\-1\end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\0\end{bmatrix}, t_4 = 0\right\}$$
$$\left\{\mathbf{p}_5 = \begin{bmatrix}0\\1\end{bmatrix}, t_5 = 0\right\}.$$



i. Draw a diagram of the single-neuron perceptron you would use to solve this problem. How many inputs are required?

ii. Draw a graph of the data points, labeled according to their targets. Is this problem solvable with the network you defined in part (i)? Why or why not?

E4.2 Consider the classification problem defined below.

$$\left\{\mathbf{p}_1 = \begin{bmatrix}-1\\1\end{bmatrix}, t_1 = 1\right\} \left\{\mathbf{p}_2 = \begin{bmatrix}-1\\-1\end{bmatrix}, t_2 = 1\right\} \left\{\mathbf{p}_3 = \begin{bmatrix}0\\0\end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix}1\\0\end{bmatrix}, t_4 = 0\right\}.$$

i. Design a single-neuron perceptron to solve this problem. Design the network graphically, by choosing weight vectors that are orthogonal to the decision boundaries.

ii. Test your solution with all four input vectors.

iii. Classify the following input vectors with your solution. You can either perform the calculations manually or with MATLAB.

$$\mathbf{p}_5 = \begin{bmatrix}-2\\0\end{bmatrix} \qquad \mathbf{p}_6 = \begin{bmatrix}1\\1\end{bmatrix} \qquad \mathbf{p}_7 = \begin{bmatrix}0\\1\end{bmatrix} \qquad \mathbf{p}_8 = \begin{bmatrix}-1\\-2\end{bmatrix}$$

iv. Which of the vectors in part (iii) will always be classified the same way, regardless of the solution values for $\mathbf{W}$ and b? Which may vary depending on the solution? Why?

E4.3 Solve the classification problem in Exercise E4.2 by solving inequalities (as in Problem P4.2), and repeat parts (ii) and (iii) with the new solution. (The solution is more difficult than Problem P4.2, since you can't isolate the weights and biases in a pairwise manner.)

E4.4 Solve the classification problem in Exercise E4.2 by applying the perceptron rule to the following initial parameters, and repeat parts (ii) and (iii) with the new solution.

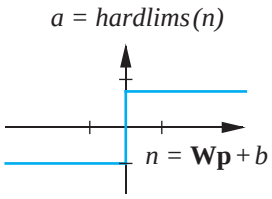
$$\mathbf{W}(0) = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad b(0) = 0$$

E4.5 Prove mathematically (not graphically) that the following problem is unsolvable for a two-input/single-neuron perceptron.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} -1 \\ 1 \end{bmatrix}, t_1 = 1\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, t_2 = 0\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, t_3 = 1\right\} \left\{\mathbf{p}_4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, t_4 = 0\right\}$$

(Hint: start by rewriting the input/target requirements as inequalities that constrain the weight and bias values.)

E4.6 The symmetric hard limit function is sometimes used in perceptron networks, instead of the hard limit function. Target values are then taken from the set [-1, 1] instead of [0, 1].



i. Write a simple expression that maps numbers in the ordered set [0, 1] into the ordered set [-1, 1]. Write the expression that performs the inverse mapping.

ii. Consider two single-neuron perceptrons with the same weight and bias values. The first network uses the hard limit function ([0, 1] values), and the second network uses the symmetric hard limit function. If the two networks are given the same input $\mathbf{p}$, and updated with the perceptron learning rule, will their weights continue to have the same value?

iii. If the changes to the weights of the two neurons are different, how do they differ? Why?

iv. Given initial weight and bias values for a standard hard limit perceptron, create a method for initializing a symmetric hard limit perceptron so that the two neurons will always respond identically when trained on identical data.

E4.7 The vectors in the ordered set defined below were obtained by measuring the weight and ear lengths of toy rabbits and bears in the Fuzzy Wuzzy Animal Factory. The target values indicate whether the respective input vector was taken from a rabbit (0) or a bear (1). The first element of the input vector is the weight of the toy, and the second element is the ear length.

$$\left\{\mathbf{p}_1 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, t_1 = 0\right\} \left\{\mathbf{p}_2 = \begin{bmatrix} 1 \\ 5 \end{bmatrix}, t_2 = 0\right\} \left\{\mathbf{p}_3 = \begin{bmatrix} 2 \\ 4 \end{bmatrix}, t_3 = 0\right\} \left\{\mathbf{p}_4 = \begin{bmatrix} 2 \\ 5 \end{bmatrix}, t_4 = 0\right\}$$

$$\left\{\mathbf{p}_5 = \begin{bmatrix}3\\1\end{bmatrix}, t_5 = 1\right\} \left\{\mathbf{p}_6 = \begin{bmatrix}3\\2\end{bmatrix}, t_6 = 1\right\} \left\{\mathbf{p}_7 = \begin{bmatrix}4\\1\end{bmatrix}, t_7 = 1\right\} \left\{\mathbf{p}_8 = \begin{bmatrix}4\\2\end{bmatrix}, t_8 = 1\right\}$$

i. Use MATLAB to initialize and train a network to solve this "practical" problem.

ii. Use MATLAB to test the resulting weight and bias values against the input vectors.

iii. Alter the input vectors to ensure that the decision boundary of any solution will not intersect one of the original input vectors (i.e., to ensure only robust solutions are found). Then retrain the network.

E4.8 Consider again the four-category classification problem described in Problems P4.3 and P4.5. Suppose that we change the input vector $\mathbf{p}_3$ to

$$\mathbf{p}_3 = \begin{bmatrix}2\\2\end{bmatrix}.$$

i. Is the problem still linearly separable? Demonstrate your answer graphically.

ii. Use MATLAB and to initialize and train a network to solve this problem. Explain your results.

iii. If $\mathbf{p}_3$ is changed to

$$\mathbf{p}_3 = \begin{bmatrix}2\\1.5\end{bmatrix}$$

is the problem linearly separable?

iv. With the $\mathbf{p}_3$ from (iii), use MATLAB to initialize and train a network to solve this problem. Explain your results.

E4.9 One variation of the perceptron learning rule is

$$\mathbf{W}^{new} = \mathbf{W}^{old} + \alpha\mathbf{e}\mathbf{p}^T$$

$$\mathbf{b}^{new} = \mathbf{b}^{old} + \alpha\mathbf{e}$$

where α is called the learning rate. Prove convergence of this algorithm. Does the proof require a limit on the learning rate? Explain.