

Perceptrons

Introduction (p. 3-2)	Introduces the chapter, and provides information on additional resources
Neuron Model (p. 3-3)	Provides a model of a perceptron neuron
Perceptron Architecture (p. 3-5)	Graphically displays perceptron architecture
Creating a Perceptron (newp) (p. 3-6)	Describes how to create a perceptron in Neural Network Toolbox
Learning Rules (p. 3-13)	Introduces network learning rules
Perceptron Learning Rule (learnp) (p. 3-14)	Discusses the perceptron learning rule learnp
Training (train) (p. 3-17)	Discusses the training function train
Limitations and Cautions (p. 3-22)	Describes the limitations of perceptron networks
Graphical User Interface (p. 3-24)	Discusses the Network/Data Manager GUI

Introduction

This chapter has a number of objectives. First it introduces you to learning rules, methods of deriving the next changes that might be made in a network, and training, a procedure whereby a network is actually adjusted to do a particular job. Along the way are described a toolbox function to create a simple perceptron network and functions to initialize and simulate such networks. The perceptron is used as a vehicle for tying these concepts together.

Rosenblatt [Rose61] created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptrons are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

This chapter defines what is meant by a learning rule, explains the perceptron network and its learning rule, and tells you how to initialize and simulate perceptron networks.

The discussion of perceptrons in this chapter is necessarily brief. For a more thorough discussion, see Chapter 4, “Perceptron Learning Rule,” of [HDB1996], which discusses the use of multiple layers of perceptrons to solve more difficult problems beyond the capability of one layer.

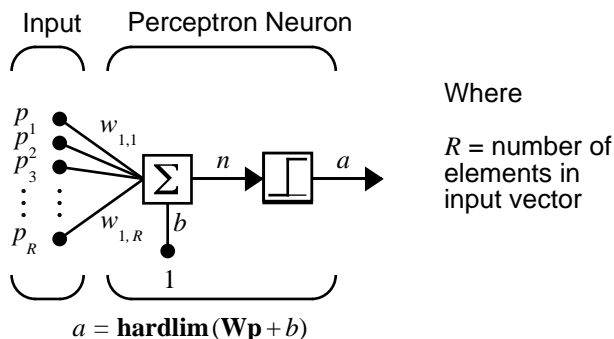
You might also want to refer to the original book on the perceptron, Rosenblatt, F., *Principles of Neurodynamics*, Washington D.C., Spartan Press, 1961 [Rose61].

Important Perceptron Functions

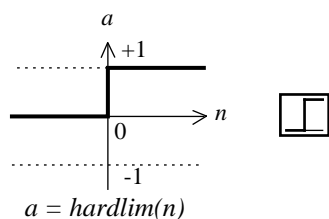
You can create perceptron networks with the function `newp`. These networks can be initialized, simulated, and trained with `init`, `sim`, and `train`. “Neuron Model” on page 3-3 describes how perceptrons work and introduces these functions.

Neuron Model

A perceptron neuron, which uses the hard-limit transfer function `hardlim`, is shown below.



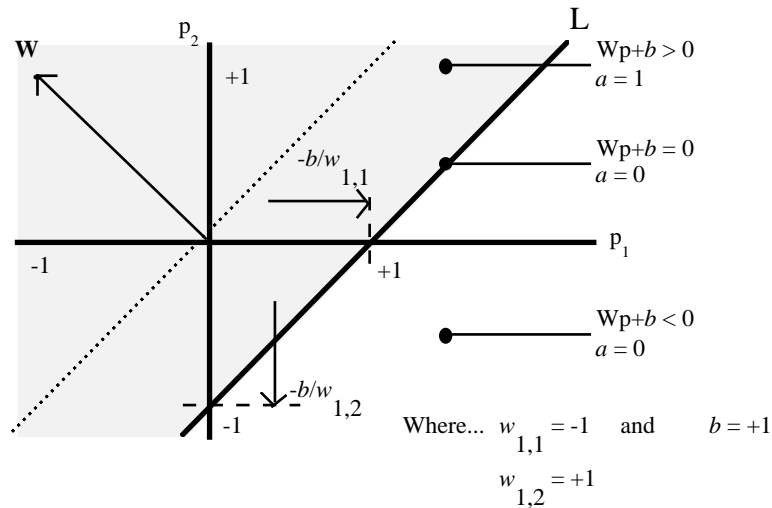
Each external input is weighted with an appropriate weight w_{1j} , and the sum of the weighted inputs is sent to the hard-limit transfer function, which also has an input of 1 transmitted to it through the bias. The hard-limit transfer function, which returns a 0 or a 1, is shown below.



Hard-Limit Transfer Function

The perceptron neuron produces a 1 if the net input into the transfer function is equal to or greater than 0; otherwise it produces a 0.

The hard-limit transfer function gives a perceptron the ability to classify input vectors by dividing the input space into two regions. Specifically, outputs will be 0 if the net input n is less than 0, or 1 if the net input n is 0 or greater. The input space of a two-input hard limit neuron with the weights $w_{1,1} = -1$, $w_{1,2} = 1$ and a bias $b = 1$ is shown below.



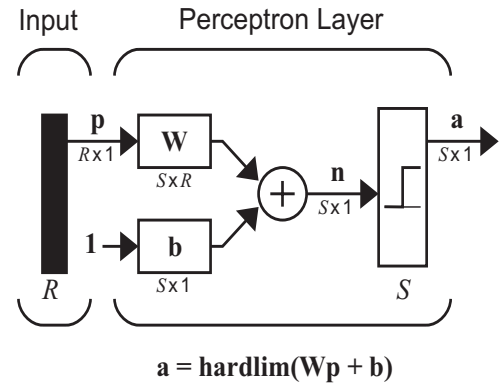
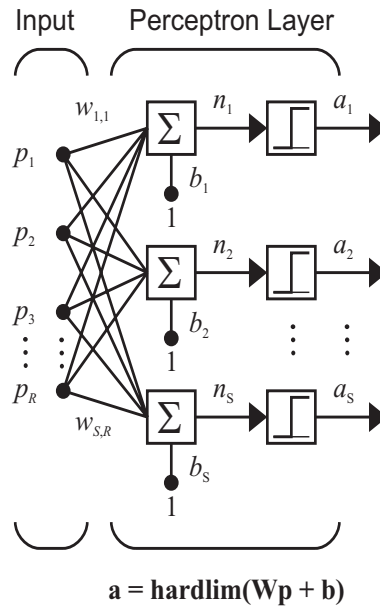
Two classification regions are formed by the *decision boundary* line L at $\mathbf{W}\mathbf{p} + b = 0$. This line is perpendicular to the weight matrix \mathbf{W} and shifted according to the bias b . Input vectors above and to the left of the line L will result in a net input greater than 0 and, therefore, cause the hard-limit neuron to output a 1. Input vectors below and to the right of the line L cause the neuron to output 0. You can pick weight and bias values to orient and move the dividing line so as to classify the input space as desired.

Hard-limit neurons without a bias will always have a classification line going through the origin. Adding a bias allows the neuron to solve problems where the two sets of input vectors are not located on different sides of the origin. The bias allows the decision boundary to be shifted away from the origin, as shown in the plot above.

You might want to run the demonstration program `nnd4db`. With it you can move a decision boundary around, pick new inputs to classify, and see how the repeated application of the learning rule yields a network that does classify the input vectors properly.

Perceptron Architecture

The perceptron network consists of a single layer of S perceptron neurons connected to R inputs through a set of weights w_{ij} , as shown below in two forms. As before, the network indices i and j indicate that w_{ij} is the strength of the connection from the j th input to the i th neuron.



Where

R = number of elements in input

S = number of neurons in layer

The perceptron learning rule described shortly is capable of training only a single layer. Thus only one-layer networks are considered here. This restriction places limitations on the computation a perceptron can perform. The types of problems that perceptrons are capable of solving are discussed in "Limitations and Cautions" on page 3-22.

Creating a Perceptron (newp)

A perceptron can be created with the function `newp`,

```
net = newp(P,T)
```

where input arguments are as follows:

- `P` is an R -by- Q matrix of Q input vectors of R elements each.
- `T` is an S -by- Q matrix of Q target vectors of S elements each.

Commonly the `hardlim` function is used in perceptrons, so it is the default.

The code below creates a perceptron network with a single one-element input vector with the values 0 and 2, and one neuron with outputs that can be either 0 or 1.

```
P = [0 2];  
T = [0 1];  
net = newp(P,T);
```

You can see what network has been created by executing the following code:

```
inputweights = net.inputweights{1,1}
```

which yields

```
inputweights =  
    delays: 0  
    initFcn: 'initzero'  
    learn: 1  
    learnFcn: 'learnp'  
    learnParam: []  
    size: [1 1]  
    userdata: [1x1 struct]  
    weightFcn: 'dotprod'  
    weightParam: [1x1 struct]
```

The default learning function is `learnp`, which is discussed in “Perceptron Learning Rule (`learnp`)” on page 3-14. The net input to the `hardlim` transfer function is `dotprod`, which generates the product of the input vector and weight matrix and adds the bias to compute the net input.

The default initialization function `initzero` is used to set the initial values of the weights to zero.

Similarly,

```
biases = net.biases{1}
```

gives

```
biases =  
    initFcn: 'initzero'  
    learn: 1  
    learnFcn: 'learnp'  
    learnParam: []  
    size: 1  
    userdata: [1x1 struct]
```

You can see that the default initialization for the bias is also 0.

Simulation (sim)

This section shows how `sim` works using a simple problem.

Suppose that you take a perceptron with a single two-element input vector, such as discussed in the decision boundary figure on page 3-4. This perceptron outputs the values 0 and 1.

Define the network using the following commands:

```
P = [-2 2; -2 2];  
T = [0 1];  
net = newp(P,[0 1]);
```

The network includes zero weights and biases. If you want weights and biases with values other than zero, you have to create them.

Set the two weights and the one bias to -1, 1, and 1, as they were in the decision boundary figure using the following commands:

```
net.IW{1,1}= [-1 1];  
net.b{1} = [1];
```


To make sure that these parameters were set correctly, check them with

```
net.IW{1,1}
ans =
    -1     1
net.b{1}
ans =
     1
```

Now see if the network responds to two signals, one on each side of the perceptron boundary.

```
p1 = [1;1];
a1 = sim(net,p1)
a1 =

    1
```

and for

```
p2 = [1;-1];
a2 = sim(net,p2)
a2 =

    0
```

Sure enough, the perceptron classified the two inputs correctly.

You could present the two inputs in a sequence and get the outputs in a sequence as well.

```
p3 = {[1;1] [1;-1]};
a3 = sim(net,p3)
a3 =

    [1]    [0]
```

Initialization (init)

You can use the function `init` to reset the network weights and biases to their original values. Suppose, for instance, that you start with the network

```
net = newp([-2 2;-2 2],[0 1]);
```

Now check the weights using the following command:

```
wts = net.IW{1,1}
```

which gives, as expected,

```
wts =

    0    0
```

In the same way, you can verify that the bias is 0 with

```
bias = net.b{1}
```

which gives

```
bias =
      0
```

Now set the weights to the values 3 and 4 and the bias to the value 5 with

```
net.IW{1,1} = [3,4];
net.b{1} = 5;
```

Recheck the weights and bias as shown above to verify that the change has been made. Sure enough,

```
wts =
      3      4
bias =
      5
```

Now use `init` to reset the weights and bias to their original values.

```
net = init(net);
```

You can check as shown above to verify that.

```
wts =
      0      0
bias =
      0
```

You can change the way that a perceptron is initialized with `init`. For instance, you can redefine the network input weights and bias `initFcns` as `rands`, and then apply `init` as shown below.

```
net.inputweights{1,1}.initFcn = 'rands';
net.biases{1}.initFcn = 'rands';
net = init(net);
```

Now check the weights and bias.

```
wt =  
  -0.2371 -0.8976  
bias =  
  0.0775
```

You can see that the weights and bias are assigned random numbers.

Learning Rules

A *learning rule* is defined as a procedure for modifying the weights and biases of a network. (This procedure can also be referred to as a training algorithm.) The learning rule is applied to train the network to perform some particular task. Learning rules in this toolbox fall into two broad categories: supervised learning, and unsupervised learning.

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

$$\{\mathbf{p}_1, \mathbf{t}_1\}, \{\mathbf{p}_2, \mathbf{t}_2\}, \dots, \{\mathbf{p}_Q, \mathbf{t}_Q\}$$

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.

In *unsupervised learning*, the weights and biases are modified in response to network inputs only. There are no target outputs available. Most of these algorithms perform clustering operations. They categorize the input patterns into a finite number of classes. This is especially useful in such applications as vector quantization.

Perceptron Learning Rule (learnp)

Perceptrons are trained on examples of desired behavior. The desired behavior can be summarized by a set of input, output pairs

$$\mathbf{p}_1 \mathbf{t}_1, \mathbf{p}_2 \mathbf{t}_1, \dots, \mathbf{p}_Q \mathbf{t}_Q$$

where \mathbf{p} is an input to the network and \mathbf{t} is the corresponding correct (target) output. The objective is to reduce the error \mathbf{e} , which is the difference $\mathbf{t} - \mathbf{a}$ between the neuron response \mathbf{a} and the target vector \mathbf{t} . The perceptron learning rule `learnp` calculates desired changes to the perceptron's weights and biases, given an input vector \mathbf{p} and the associated error \mathbf{e} . The target vector \mathbf{t} must contain values of either 0 or 1, because perceptrons (with `hardlim` transfer functions) can only output these values.

Each time `learnp` is executed, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.

If a bias is not used, `learnp` works to find a solution by altering only the weight vector \mathbf{w} to point toward input vectors to be classified as 1 and away from vectors to be classified as 0. This results in a decision boundary that is perpendicular to \mathbf{w} and that properly classifies the input vectors.

There are three conditions that can occur for a single neuron once an input vector \mathbf{p} is presented and the network's response \mathbf{a} is calculated:

CASE 1. If an input vector is presented and the output of the neuron is correct ($\mathbf{a} = \mathbf{t}$ and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 0$), then the weight vector \mathbf{w} is not altered.

CASE 2. If the neuron output is 0 and should have been 1 ($\mathbf{a} = 0$ and $\mathbf{t} = 1$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = 1$), the input vector \mathbf{p} is added to the weight vector \mathbf{w} . This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a 1 in the future.

CASE 3. If the neuron output is 1 and should have been 0 ($\mathbf{a} = 1$ and $\mathbf{t} = 0$, and $\mathbf{e} = \mathbf{t} - \mathbf{a} = -1$), the input vector \mathbf{p} is subtracted from the weight vector \mathbf{w} . This makes the weight vector point farther away from the input vector, increasing the chance that the input vector will be classified as a 0 in the future.

The perceptron learning rule can be written more succinctly in terms of the error $\mathbf{e} = \mathbf{t} - \mathbf{a}$ and the change to be made to the weight vector $\Delta\mathbf{w}$:

CASE 1. If $\mathbf{e} = 0$, then make a change $\Delta\mathbf{w}$ equal to 0.

CASE 2. If $\mathbf{e} = 1$, then make a change $\Delta\mathbf{w}$ equal to \mathbf{p}^T .

CASE 3. If $\mathbf{e} = -1$, then make a change $\Delta\mathbf{w}$ equal to $-\mathbf{p}^T$.

All three cases can then be written with a single expression:

$$\Delta\mathbf{w} = (t - a)\mathbf{p}^T = e\mathbf{p}^T$$

You can get the expression for changes in a neuron's bias by noting that the bias is simply a weight that always has an input of 1:

$$\Delta b = (t - a)(1) = e$$

For the case of a layer of neurons you have

$$\Delta\mathbf{W} = (\mathbf{t} - \mathbf{a})(\mathbf{p})^T = \mathbf{e}(\mathbf{p})^T$$

and

$$\Delta\mathbf{b} = (\mathbf{t} - \mathbf{a}) = \mathbf{e}$$

The perceptron learning rule can be summarized as follows:

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

and

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

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

Now try a simple example. Start with a single neuron having an input vector with just two elements. Here are input vectors with the values -2 and 2, and outputs with values 0 and 1.

```
net = newp([-2 2;-2 2],[0 1]);
```

To simplify matters, set the bias equal to 0 and the weights to 1 and -0.8.

```
net.b{1} = [0];
```

```
w = [1 -0.8];  
net.IW{1,1} = w;
```

The input target pair is given by

```
p = [1; 2];  
t = [1];
```

You can compute the output and error with

```
a = sim(net,p)  
a =  
    0  
e = t-a  
e =  
    1
```

and use the function `learnp` to find the change in the weights.

```
dw = learnp(w,p,[],[],[],[],e,[],[],[])  
dw =  
    1    2
```

The new weights, then, are obtained as

```
w = w + dw  
w =  
    2.0000    1.2000
```

The process of finding new weights (and biases) can be repeated until there are no errors. Recall that the perceptron learning rule is guaranteed to converge in a finite number of steps for all problems that can be solved by a perceptron. These include all classification problems that are linearly separable. The objects to be classified in such cases can be separated by a single line.

You might want to try `demo nnd4pr`. It allows you to pick new input vectors and apply the learning rule to classify them.

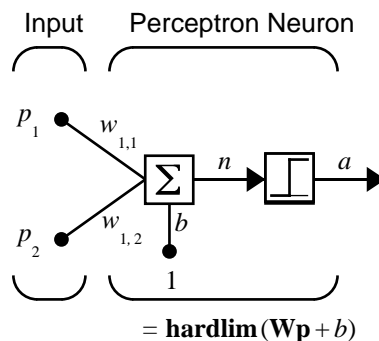
Training (train)

If `sim` and `learnp` are used repeatedly to present inputs to a perceptron, and to change the perceptron weights and biases according to the error, the perceptron will eventually find weight and bias values that solve the problem, given that the perceptron *can* solve it. Each traversal through all the training input and target vectors is called a *pass*.

The function `train` carries out such a loop of calculation. In each pass the function `train` proceeds through the specified sequence of inputs, calculating the output, error, and network adjustment for each input vector in the sequence as the inputs are presented.

Note that `train` does not guarantee that the resulting network does its job. You must check the new values of \mathbf{W} and \mathbf{b} by computing the network output for each input vector to see if all targets are reached. If a network does not perform successfully you can train it further by calling `train` again with the new weights and biases for more training passes, or you can analyze the problem to see if it is a suitable problem for the perceptron. Problems that cannot be solved by the perceptron network are discussed in “Limitations and Cautions” on page 3-22.

To illustrate the training procedure, work through a simple problem. Consider a one-neuron perceptron with a single vector input having two elements:



This network, and the problem you are about to consider, are simple enough that you can follow through what is done with hand calculations if you want. The problem discussed below follows that found in [HDB1996].

Suppose you have the following classification problem and would like to solve it with a single vector input, two-element perceptron network.

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

Use the initial weights and bias. Denote the variables at each step of this calculation by using a number in parentheses after the variable. Thus, above, the initial values are $\mathbf{W}(0)$ and $b(0)$.

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

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

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

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

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

You can calculate the new weights and bias using the perceptron update rules.

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

Now present the next input vector, \mathbf{p}_2 . The output is calculated below.

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

On this occasion, the target is 1, so the error is zero. Thus there are no changes in weights or bias, so $\mathbf{W}(2) = \mathbf{W}(1) = \begin{bmatrix} -2 & -2 \end{bmatrix}$ and $p(2) = p(1) = -1$.

You can continue in this fashion, presenting \mathbf{p}_3 next, calculating an output and the error, and making changes in the weights and bias, etc. After making one pass through all of the four inputs, you get the values $\mathbf{W}(4) = \begin{bmatrix} -3 & -1 \end{bmatrix}$ and $b(4) = 0$. To determine whether a satisfactory solution is obtained, make one pass through all input vectors to see if they all produce the desired target values. This is not true for the fourth input, but the algorithm does converge on the sixth presentation of an input. The final values are

$$\mathbf{W}(6) = \begin{bmatrix} -2 & -3 \end{bmatrix} \text{ and } b(6) = 1$$

This concludes the hand calculation. Now, how can you do this using the train function?

The following code defines a perceptron like that shown in the previous figure, with initial weights and bias values of 0.

```
net = newp([-2 2;-2 2],[0 1]);
```

Consider the application of a single input.

```
p = [2; 2];
```

having the target

```
t = [0];
```

Set epochs to 1, so that train goes through the input vectors (only one here) just one time.

```
net.trainParam.epochs = 1;
net = train(net,p,t);
```

The new weights and bias are

```
w =
    -2    -2
b =
    -1
```

Thus, the initial weights and bias are 0, and after training on only the first vector, they have the values [-2 -2] and -1, just as you hand calculated.

Now apply the second input vector \mathbf{p}_2 . The output is 1, as it will be until the weights and bias are changed, but now the target is 1, the error will be 0, and the change will be zero. You could proceed in this way, starting from the previous result and applying a new input vector time after time. But you can do this job automatically with `train`.

Apply `train` for one epoch, a single pass through the sequence of all four input vectors. Start with the network definition.

```
net = newp([-2 2;-2 2],[0 1]);
net.trainParam.epochs = 1;
```

The input vectors and targets are

```
p = [[2;2] [1;-2] [-2;2] [-1;1]]
t = [0 1 0 1]
```

Now train the network with

```
net = train(net,p,t);
```

The new weights and bias are

```
w =
    -3    -1
b =
     0
```

Note that this is the same result as you got previously by hand. Finally, simulate the trained network for each of the inputs.

```
a = sim(net,p)
a =
     0     0     1     1
```

The outputs do not yet equal the targets, so you need to train the network for more than one pass. Try more epochs. This run gives the following results:

```
TRAINC, Epoch 0/20
TRAINC, Epoch 2/20
TRAINC, Performance goal met.
```

Thus, the network was trained by the time the inputs were presented on the third epoch. (As you know from hand calculation, the network converges on the presentation of the sixth input vector. This occurs in the middle of the second epoch, but it takes the third epoch to detect the network convergence.) The final weights and bias are

```
w =
    -2    -3
b =
     1
```

The simulated output and errors for the various inputs are

```
a =
     0     1     0     1
error = a-t
error =
     0     0     0     0
```

You confirm that the training procedure is successful. The network converges and produces the correct target outputs for the four input vectors.

The default training function for networks created with `newp` is `trainc`. (You can find this by executing `net.trainFcn`.) This training function applies the perceptron learning rule in its pure form, in that individual input vectors are applied individually, in sequence, and corrections to the weights and bias are made after each presentation of an input vector. Thus, perceptron training with `train` will converge in a finite number of steps unless the problem presented cannot be solved with a simple perceptron.

The function `train` can be used in various ways by other networks as well. Type `help train` to read more about this basic function.

You might want to try various demonstration programs. For instance, `demop1` illustrates classification and training of a simple perceptron.

Limitations and Cautions

Perceptron networks should be trained with `adapt`, which presents the input vectors to the network one at a time and makes corrections to the network based on the results of each presentation. Use of `adapt` in this way guarantees that any linearly separable problem is solved in a finite number of training presentations.

As noted in the previous pages, perceptrons can also be trained with the function `train`, which is discussed in detail in the next chapter. Commonly when `train` is used for perceptrons, it presents the inputs to the network in batches, and makes corrections to the network based on the sum of all the individual corrections. Unfortunately, there is no proof that such a training algorithm converges for perceptrons. On that account the use of `train` for perceptrons is not recommended.

Perceptron networks have several limitations. First, the output values of a perceptron can take on only one of two values (0 or 1) because of the hard-limit transfer function. Second, perceptrons can only classify linearly separable sets of vectors. If a straight line or a plane can be drawn to separate the input vectors into their correct categories, the input vectors are linearly separable. If the vectors are not linearly separable, learning will never reach a point where all vectors are classified properly. However, it has been proven that if the vectors are linearly separable, perceptrons trained adaptively will always find a solution in finite time. You might want to try `demo6`. It shows the difficulty of trying to classify input vectors that are not linearly separable.

It is only fair, however, to point out that networks with more than one perceptron can be used to solve more difficult problems. For instance, suppose that you have a set of four vectors that you would like to classify into distinct groups, and that two lines can be drawn to separate them. A two-neuron network can be found such that its two decision boundaries classify the inputs into four categories. For additional discussion about perceptrons and to examine more complex perceptron problems, see [HDB1996].

Outliers and the Normalized Perceptron Rule

Long training times can be caused by the presence of an *outlier* input vector whose length is much larger or smaller than the other input vectors. Applying the perceptron learning rule involves adding and subtracting input vectors from the current weights and biases in response to error. Thus, an input vector

with large elements can lead to changes in the weights and biases that take a long time for a much smaller input vector to overcome. You might want to try demop4 to see how an outlier affects the training.

By changing the perceptron learning rule slightly, you can make training times insensitive to extremely large or small outlier input vectors.

Here is the original rule for updating weights:

$$\Delta \mathbf{w} = (t - a) \mathbf{p}^T = e \mathbf{p}^T$$

As shown above, the larger an input vector \mathbf{p} , the larger its effect on the weight vector \mathbf{w} . Thus, if an input vector is much larger than other input vectors, the smaller input vectors must be presented many times to have an effect.

The solution is to normalize the rule so that the effect of each input vector on the weights is of the same magnitude:

$$\Delta \mathbf{w} = (t - a) \frac{\mathbf{p}^T}{\|\mathbf{p}\|} = e \frac{\mathbf{p}^T}{\|\mathbf{p}\|}$$

The normalized perceptron rule is implemented with the function `learnpn`, which is called exactly like `learnp`. The normalized perceptron rule function `learnpn` takes slightly more time to execute, but reduces the number of epochs considerably if there are outlier input vectors. You might try demop5 to see how this normalized training rule works.