Neural Networks
CC524

Basic Neuron Model and the Linear Perceptron

The Basic Neuron Model

- We will consider the features of a single neuron and how we can model it. The basic function of a biological neuron is to add up its inputs (with certain weights) and to produce an output if this sum is greater than some value, known as the threshold value.

- The inputs to the neuron arrive along the dendrites, which are connected to the outputs from other neurons by the synapses. These synapses control the weights by which the signals are transmitted; some synapses are effective and pass a large signal across: (large weights), whilst others are very poor, and

allow very little through: (small weights). The neuron receives all these inputs and fires an output if the total input exceeds the threshold value.

- Our neuron model must capture the above features, summarized as follows:

1. The output from a neuron is either on (fires) or off.
2. The output depends on a weighted sum of the inputs.

The weights correspond to the strengths of the synapses, and sometimes they are called synaptic weights.

So, now we have our basic model of the neuron, figure(2): it performs a *weighted sum* of its inputs, compares this to some internal *threshold* level, and turns on only if this level is exceeded, if not, it stays off.

Because the inputs are passed through the model neuron to produce the output (without any feedback connections) the system is called ***feedforward.***

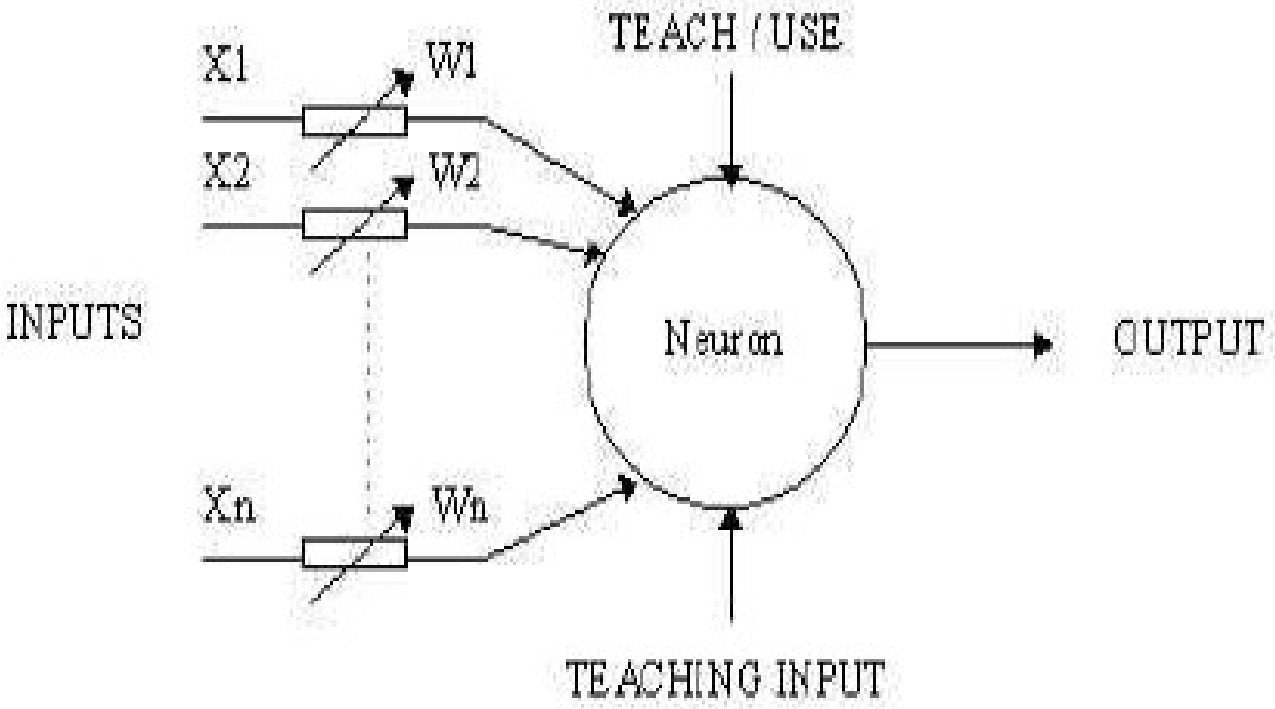
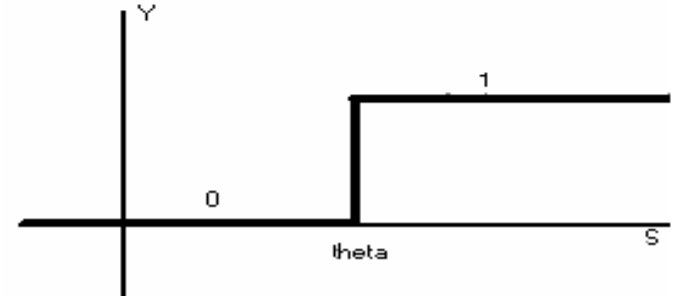


Fig.2: Single neuron:

total input $= x_1.w_1 + x_2.w_2 + \ldots\ldots\ldots + x_n.w_n$

- The sum "s" is compared to a threshold value in the neuron "θ". This thresholding process is accomplished by comparison; if the sum is greater than θ, then the output y is 1, if less, the output is 0.
- This can be seen graphically in fig.3, where the x-axis represents the total input sum, and the y-axis is the output.



- The output is given by:

$$y = f_h \left(\sum_{i=1,n} x_i(t) \, . \, w_i(t) - \theta \right) \qquad (1)$$

Where f_h is a step function (known as the Heaviside function), and

$$f_h(s) = 1 \quad s > 0 \qquad (2)$$
$$f_h(s) = 0 \quad s \leq 0$$

- If we use the convention of biasing the neuron, we can define an extra input, always=1, with a weight that represents the threshold (bias) of the neuron. The equation of the neuron's output thus becomes:

$$y = f_h \left(\sum_{i=0,n} x_i(t) \cdot w_i(t) \right) \qquad (3)$$

- This model of the neuron was proposed in 1943 by McCulloch and Pitts. Rosenblatt, in 1962, gave the name "perceptrons" to neurons connected up in a simple fashion in his pioneering book "principles of neurodynamics".
- He stated that perceptrons are not attempting to build computer brains, rather, they are models to discover the properties and ideas of learning in a very simplified form.

- Now, we need a mechanism for achieving learning in our neuron model. Connecting these neurons together may produce networks that can do something but we need to be able to train them in order for them to do anything useful.
- We also want to find the simplest learning rule that we can, in order to keep our model understandable.
- Inspired by real neural systems, young children are praised for saying names correctly, and are scolded for trying to touch hot cups, this goes on until they learn!
- In general, good behavior is reinforced, whilst bad behavior is punished or discouraged. We can transfer this idea to our network. We must try to reinforce behavior (outputs) that we want (that are correct) and discourage behavior that is wrong.

- For example, if we have 2 groups of objects; one with several differently written A's and the other of B's. We want our neuron to tell the A's from the B's, as in fig.4.
  (of course we'll discuss later how to really do that !!)
- We want our neuron to output a 1 when A is presented and 0 when it sees a B. The guiding principle is to let our neuron to learn from its mistakes; if it produces the correct answer then we do nothing ☺ (or give it a bonus). If it produces the incorrect output, then we want to reduce the chances that this happens again ☹.
- We set up our neuron with ***random weights*** on its input at the beginning; corresponding to an ignorance state (it knows nothing in the beginning), then we present an A.

- The neuron performs a weighted sum of the inputs, compare it to the threshold, if it exceeds the threshold, it'll output 1 (correct), if not, it'll output 0 (wrong). At the beginning, the chance is 50:50.
- The Hebbian rule of learning (after Hebb 1949) says that; if it gets the correct answer, we do nothing. If it gets it wrong, we want to increase the weighted sum so that the next time, it exceeds the threshold. We would do this by increasing the weights. (to reinforce the chances of getting a 1). And vice versa if the input is a B and the output exceeds the threshold. Then, we'd want to decrease the weighted sum, by decreasing the weights, so that the weighted sum is less than the threshold the next time.

- This goes on for all the examples we have for the A's and B's (all the training examples). Then, we repeat all the training examples again and again until we get all of them correctly (hopefully, if this can actually be done).

- This process is formally called the "perceptron learning algorithm", and is summarized in the following steps:

Perceptron Learning Algorithm:

1. Define $w_i(t)$ $(0 < i < n)$ to be the weight from input i at time t , where $w_0 = -\theta$ is the bias term, and the corresponding input $x_0=1$ always. Set all the weights to small random values.
2. Present inputs ($x_0(t)$, $x_1(t)$, $x_2(t)$, ..., $x_n(t)$) to the neuron, where the desired output is d(t). For example, if the input is class-A d(t)=1 and if input is class-B d(t)=0.
3. Calculate the sum $S= \sum wi(t).xi(t)$, add the bias, and then the final output after the step function y(t), which will be either 0 or 1.
4. Adapt weights according to the perceptron rule:

If output is correct; $wi(t+1)=wi(t)$

If output is wrong:

a- if output is y(t)=0 and d(t)=1 (class-A): (4)

$wi(t+1)=wi(t) + \eta.xi(t)$

b- if output is y(t)=1 and d(t)=0 (class-B):

$wi(t+1)=wi(t) - \eta.xi(t)$

where η is a positive coefficient between 0 and 1, called the adaptation rate or learning gain, and is used to control the amount of adaptation for the weights.

It is clear from the perceptron learning rule that if the perceptron (neuron) is doing a good job, we leave it as it is.

On the other hand, if the weighted sum is lower than it should be, resulting on a 0 output while it should be 1, then we try to increase the weights by an amount corresponding to the inputs connected to them. This makes sense, since larger inputs contribute more to the sum, thus their weights should be corrected more significantly.

Similarly, if the weighted sum is higher than it should, resulting on a 1 output, while it should be 0, then we try to decrease the weights in a way proportional to their respective inputs. The η value determines how strong we do that.

- If η is too big the learning may keep oscillating. If η is too small, the learning may be too slow. In practice, η is chosen by trial and error, or it is made adaptive.

- The above steps are repeated for the whole training examples (epoch). Then, the epochs are repeated until we get zero misclassifications, or until we can't improve the results any more.
    - The trained perceptron should now be able to classify A from B, even for examples it never seen before.

For a very interesting and simple reference on the ANN introduction, please take a look at:
www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html

- The perceptron learning procedure has another different form, which summarizes the 3 previous weight update equations in one: The weight update equation becomes:

$$w_i(t+1)=w_i(t) + \eta \, . \, \mathbf{\Delta(t)} \, . \, x_i(t) \qquad (5)$$

where $\Delta(t) = d(t) - y(t)$ (6)

It is the error between the desired output d(t) and the actual output after the "hardlim" function y(t).

For Class-A, d(t)=1, for Class-B d(t)=0.

If the y(t)=d(t), no change occurs in the weights.

If d(t)=1, and y(t)=0, $\Delta(t)=1$, and we get the first update eqn.

If d(t)=0, and y(t)=1, $\Delta(t)=-1$, and we get the second eqn.

- The above delta-based rule is due to Rosenblatt, 1959. By following that procedure, and choosing a learning rate between 0 and 1, Minsky and Papert, In their book "Perceptrons, 1969" proved that the weight update will reduce the error at each step (iteration) and will converge to the best solution available (see book, pp.53-57).
- Widrow and Hoff, in 1960, "later re-established by Duda and Hart in 1973", have proposed a learning algorithm for the perceptron which they called

adaptive linear combiner "ADALINE", which has a linear output (no hardlim). This learning algorithm is considered one of the most important adaptation algorithms in history, and is based on minimizing the error between the actual output and the desired response. It is called the least-mean square error algorithm "LMS" and is a gradient-descent technique.

Widrow Hoff Least mean square algorithm "LMS":

The linear sum at the neuron's input is $S(t)=\sum_{i=1,n} x_i(t) . w_i(t)$, and the error between that output and the desired output d(t) is $e(t)=d(t) - S(t)$. Assuming a threshold of 0.5, we want the S(t) to be 1 for class-A, and 0 for class-B, such that for **A,** S(t) is higher than 0.5, and for **B,** S(t) is smaller than 0.5.

In optimization, a very popular error function is the squared error $E(t) = [d(t) - S(t)]^2$. Thus, we want to minimize this squared error with respect to the weights to get the optimal values of those weights which minimize the error between the desired output and the actual output.

The approach to do this is using a gradient-descent technique, where the weights are updated by:

$$w_i(t+1) = w_i(t) - \eta \, \partial E/\partial w_i \qquad (7)$$

differentiating the squared error with respect to the weight w_i we get: $\partial E/\partial w_i = - 2 (d(t) - S(t)) . x_i(t)$ (8)

substituting (8) in (7), we get the LMS learning equation:

$$w_i(t+1) = w_i(t) + \eta\ (d(t) - S(t))\ .\ x_i(t) \qquad (9)$$

and since $e(t) = d(t) - S(t)$ thus (9) can be written as:

$$w_i(t+1) = w_i(t) + \eta\ .\ \mathbf{e(t)}\ .\ x_i(t) \qquad (10)$$

which is very similar to the form in (5). However:

- In this case, and since the output is linear "***purelin***" **e(t)** can take any value between −1 and 1 , unlike **Δ(t)** which takes only the values −1, 0, and 1.
- This means that in the Widrow-Hoff LMS delta rule, the weights keep changing all the time, even if the classification is correct as long as e(t) is not yet zero.
- Also, the advantage of this technique is that e(t) corresponds to the real error between the desired and the actual. Thus, in cases when this difference is large, the weight update is large. When this difference is small, the weight update is small.

- Notice that the differentiation (gradient-descent method) requires that the E(t) be differentiable with respect to the weights. That's why we can't use this proof if the output is taken after a ***hardlim*** unit step, since it is not differentiable (***logsig*** and ***tansig*** are commonly used).
- We can also assume that the threshold is unknown, and include the $x_0=1$ (bias) and w_0 in our linear output S'. The desired response d(t) in this case may be put 1 for class-A and –1 for class-B, such that, when S' is greater than 0 we decide for class-A, and when S' is smaller than 0 we decide for class-B.

The Perceptron: A vectorial perspective:

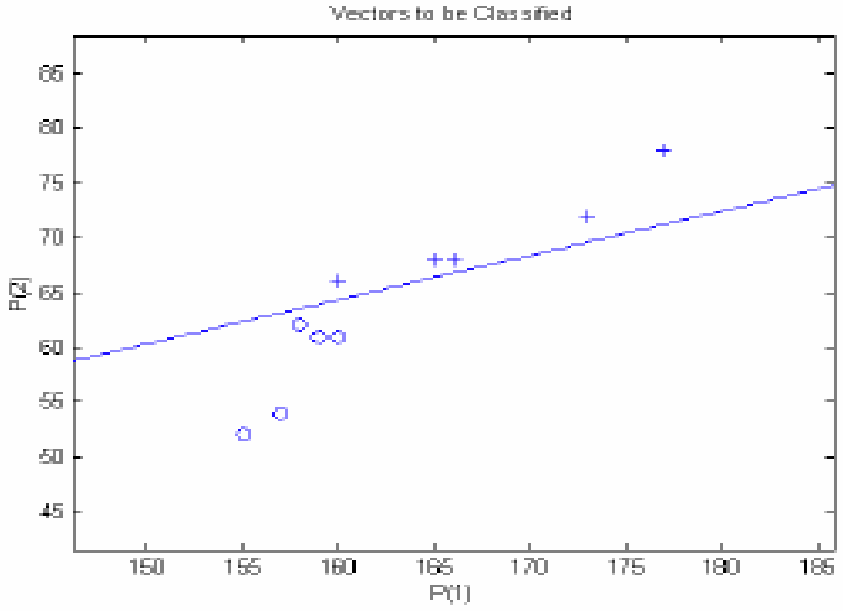
Let's assume that the data from classes A&B are represented by 2-dimensional vectors (x_1,x_2). As an example, let the problem be to distinguish between males and females based on 2 features: x_1=height x_2=weight. Let's assume we have 5 male students and 5 female students with (x_1,x_2) as:

Males: student1:(160,66), student2:(166,68), student3(165,68),

student4(177,78) and student5(173,72).

Females: student1(158,62), student2(155,52), student3(157,54), student4(159,61) and student5(160,61).

Each (x_1,x_2) pair is called a feature vector, or a pattern in a 2-dimensional space. (we can have more features if we have more information about the students or in general, about the problem).The 2-dimensional space for the student features is shown:



Let's use a perceptron to recognize males from females based on the given features (x_1,x_2). The perceptron will have 2 inputs x_1 and x_2, connected to the weights w_1 and w_2.

Also, we have a bias=x_0=1 connected to a weight w0, where the threshold $\theta = - w_0 . x_0$

The linear output of the perceptron:

$$S = w_1.x_1 + w_2.x_2 - \theta \qquad (11)$$

When this output is > zero, the y=1 and we decide class-A

when this output is < zero, the y=0 and we decide class-B.

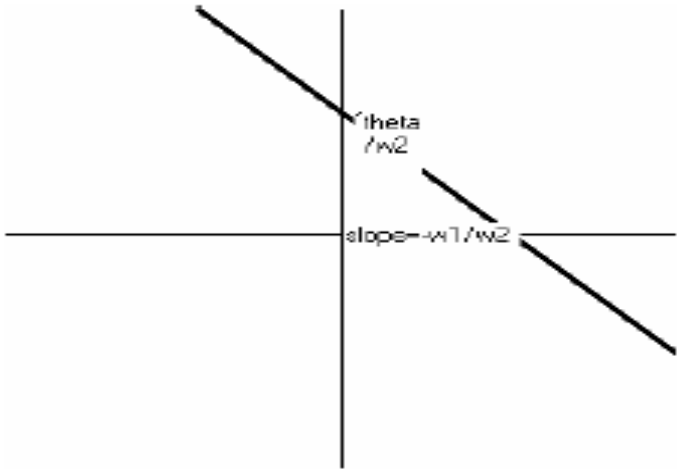
Thus, the perceptron **decision boundary**, which divides the x_1-x_2 space between class-A and class-B is when $S'=0$:

$$w_1.x_1 + w_2.x_2 - \theta = 0 \qquad (12)$$

(remember that the straight line eqn is y=mx+c), putting (12) on the same form, where the vertical axis is x_2 and the horizontal is x_1, we get:

$$x_2 = -w_1/w_2 \, . \, x_1 + \theta/w_2 \qquad (12)'$$

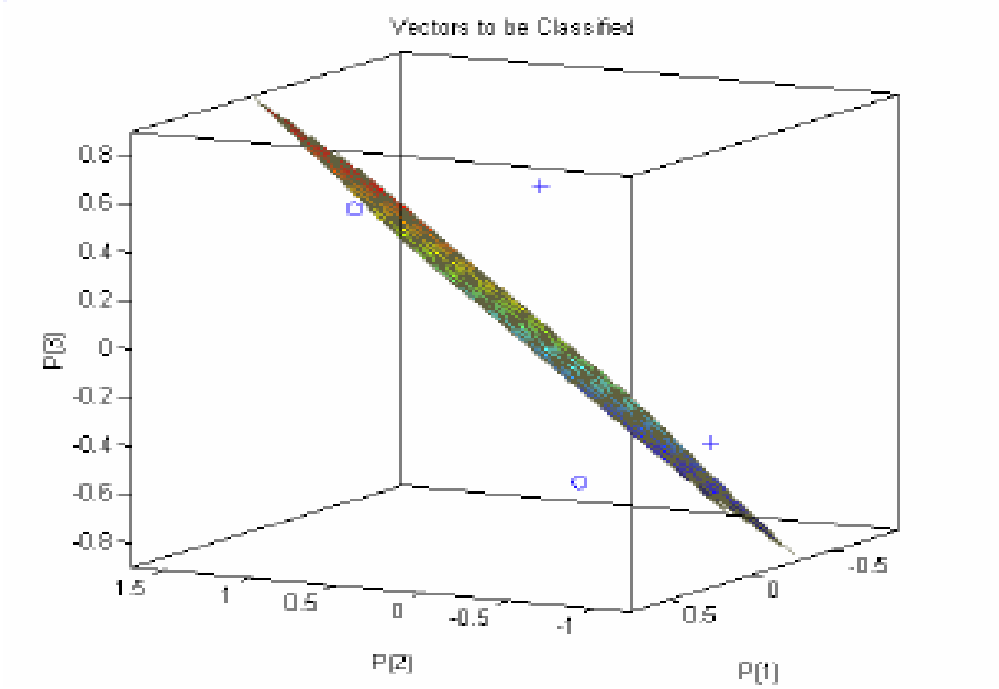
which is a straight line in the x_1-x_2 space, with a slope= - w_1/w_2 and intersection of θ/w_2 .



Thus, any given w_1 , w_2 and θ will give a new line which splits the space into 2 parts; one corresponds to $S' > 0$, and the other for $S' < 0$: i.e., one for class-A and one for class-B.

- Hence, for each set of weights and bias, the perceptron produces a decision boundary in the space. In a 2-dimensional space, this boundary is a straight line as seen before. In 3 or more dimensional spaces, this becomes a plane, then a hyperplane respectively.
- Points (patterns) above the line (hyperplane) can be regarded as representing patterns from class-A, whilst those below the line are from class-B.
- This line is what we want the perceptron to discover for itself by the learning algorithm, using the available training data, by adjusting the values of its weights.

- This is achieved, graphically, by moving the line a finite number of times until it separates the space between the data from the class-A and class-B.



An example of a 3-dimensional space with 2 points for each class. The decision boundary is ***a plane*** in this case.

A *Matlab* example:

In this example, The input vectors (Patterns) used for training are generated:

P=[-0.5 -0.5 0.3 0.1; -0.5 0.5 -0.5 1.0]; This defines 4, 2-dimensional vectors; a1=[-0.5 –0.5], a2=[-0.5 0.5] and b1=[0.3 -0.5], b2=[0.1 1.0]

The target vector T is defined as:

T=[1 1 0 0]; The desired response (target data)

The initial set of weights is put at random w1= -1 , w2= -1.3 and the bias is fixed at x_0=1 & w_0=1, i.e. θ= -1. This is done as follows:

minmax(P) This gives the ranges of x1 and x2

net = newp(minmax(P),1); This creates a new perceptron with 1 o/p.

net.iw{1,1} = [-1.0 -1.3]; This initializes the 2 weights

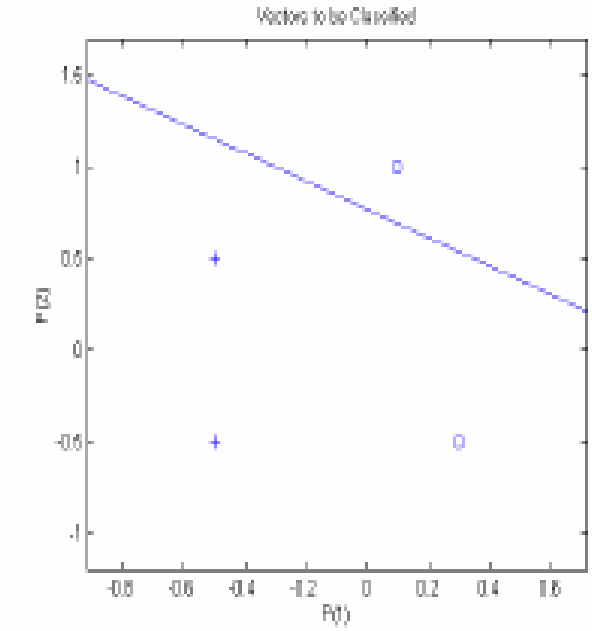
net.b{1} = 1; This init. the bias (b equals the w_0 while x_0 is fixed to 1).

Now we want to plot the patterns and the initial decision line produced by the perceptron:

plotpv(P,T); This simply plots the training data

linehandle=plotpc(net.iw{1,1},net.b{1});

This plots the initial decision boundary between the classes before training. The result is shown in this figure, which is not doing the correct separation between the classes.



e=1; just an initialization

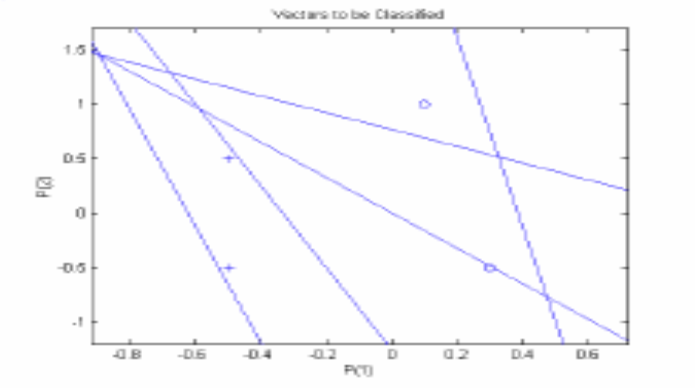
while (sse(e)) This continues until e=0, where e is the error

[net,y,e] = adapt(net,P,T);

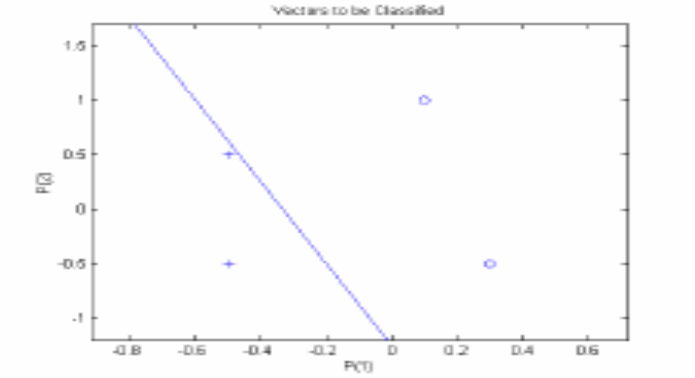
This is the adaptation done on epoch basis, and
produces the adapted net, the output y, and the error e.

plotpc(net.iw{1,1},net.b{1})

This plots all the decision boundaries over one another



linehandle=plotpc(net.iw{1,1},net.b{1},linehandle);
drawnow;
This plots the decision boundary at each iteration, and removes previous ones.
end



This shows the final decision which separates the 2 classes perfectly. Final weights are {-3 -0.8} and bias weight = -1.

The *train* function:

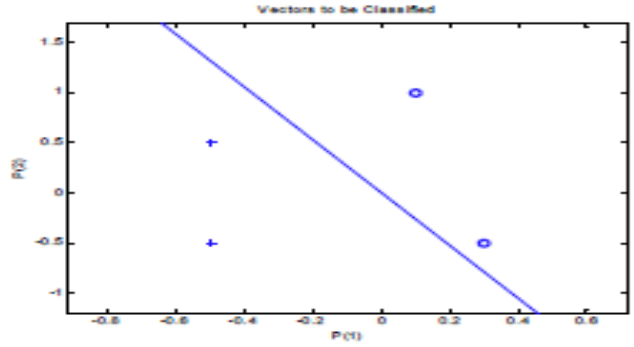
The same example is repeated using the function *train* instead of *adapt*. *train* updates the weights after each pattern is presented, while *adapt* accumulates the updates for each epoch. The *train* function corresponds to the perceptron learning we have studied.

net.trainParam.goal = 0.0; this means train until error=0.
[net,y,e] = train(net,P,T); This is the adaptation done on pattern by pattern basis, and produces the adapted net, the output y, and the error e.

```
net.iw{1,1}
net.b{1}
plotpv(P,T)
plotpc(net.iw{1,1},net.b{1})
```

The output from the program gives:

weights = -2.1 -0.8
bias weight = 0
(Exactly as we obtained using the mathematical calculations).
In this case, only 3 updates were needed, and the final decision boundary is:



Now, let's repeat the same example by hand calculations:

Initial perceptron weights are: w_1=-1 , w_2=-1.3 , w_0=1 x_0=1.
Initial decision line:
$S' = w_1.x_1 + w_2.x_2 + w_0.x_0 = 0$
$- x_1 - 1.3x_2 + 1 = 0$ (line with slope –0.77 and intersection at 0.77).

1st Epoch:

1st pattern; a1[-0.5 -0.5], T=1
S' = (-0.5*-1) + (-0.5*-1.3) + 1=2.3 +ve correct, no change

2nd pattern; a2[-0.5 0.5], T=1
S' = (-0.5*-1) + (0.5*-1.3) + 1=0.85 +ve correct, no change

3rd pattern; b1[0.3 –0.5], T=0
S' = (0.3*-1) + (-0.5*-1.3) + 1=1.35 +ve wrong: weight update:
w0= 1 – 1 = 0
w1= -1 - 0.3 = -1.3
w2= -1.3 + 0.5 = -0.8

4th pattern; b2[0.1 1.0], T=0
S' = (0.1*-1.3) + (1.0*-0.8) + 0 = -0.93 -ve correct, no change.
Finished 1st Epoch.

2nd Epoch

1st pattern; a1[-0.5 -0.5], T=1
S' = (-0.5*-1.3) + (-0.5*-0.8) + 0= +ve correct, no change

2nd pattern; a2[-0.5 0.5], T=1
S' = (-0.5*-1.3) + (0.5*-0.8) + 0= +ve correct, no change

3rd pattern; b1[0.3 -0.5], T=0
S' = (0.3*-1.3) + (-0.5*-0.8) + 0= 0.01 +ve wrong, weight update:
w0=0 – 1 = -1
w1= -1.3 – 0.3 = -1.6
w2= -0.8 + 0.5 = -0.3

4th pattern; b2[0.1 1.0], T=0
S' = (0.1*-1.6) + (1.0*-0.3) - 1 = -1.46 -ve correct, no change.
Finished 2nd Epoch.

3rd Epoch

1st pattern; a1[-0.5 -0.5], T=1
S' = (-0.5*-1.6) + (-0.5*-0.3) - 1 = -0.05 -ve, wrong, weight update:
w0 = -1 + 1 = 0
w1= -1.6 –0.5 = -2.1
w2= -0.3 –0.5 = -0.8

2nd pattern; a2[-0.5 0.5], T=1
S' = (-0.5*-2.1) + (0.5*-0.8) + 0 = +ve correct, no change

3rd pattern; b1[0.3 -0.5], T=0
S' = (0.3*-2.1) + (-0.5*-0.8) + 0 = -ve correct, no change

4th pattern; b2[0.1 1.0], T=0
S' = (0.1*-2.1) + (1.0*-0.8) + 0 = -ve correct, no change

Finished 3rd Epoch.

4th Epoch

1st pattern; a1[-0.5 -0.5], T=1
S' = (-0.5*-2.1) + (-0.5*-0.8) + 0 = +ve correct, no change
(That was the one which was wrong before the last weight update), and all the other three were correct after that last update. Thus, All the patterns are correctly classified now, i.e., the error=0, and the **learning is done**.

Final weights:
w_0=**0**
w_1=**-2.1**
w_2=**-0.8**

which are exactly what we have obtained using the ***train*** function, as shown earlier.

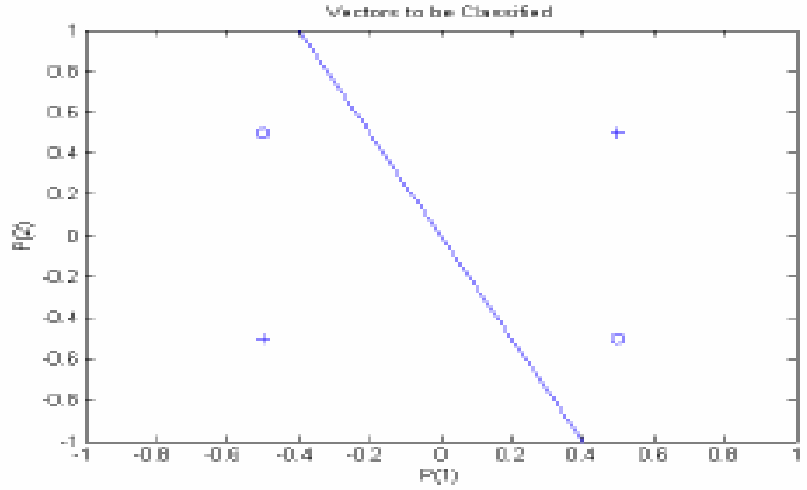
Limitations of the Perceptron:

(for interactive perceptron, see:)
http://www.hitl.washington.edu/people/mbrown/Perceptron.html

- The perceptron produces a straight line decision boundary in 2-dimensional space, i.e., when we have only 2 features (attributes) for the classes. In 3-dimensional spaces, i.e., 3 features, it produces a plane. In general, when we have n-features patterns, it produces (n – 1)-dimensional hyperplanes.
- This is all very good and sufficient to solve classification problems where the patterns for the classes can be separated by the line, or (hyper) plane, as we have seen in the previous examples. ***However, life is not always that nice!.***
- Minsky and Papert, in their book "perceptrons", in 1969, have shown that fact, and hence criticized the simple neuron perceptron severely. That was the main reason the neural networks field stayed very calm for about 15 years.

- The problem, mathematically, is that the output which is compared to the threshold, and used to make the decision in the perceptron is a linear function of $\{x_n\}$, thus, only a linear boundary can be obtained: only ***linearly separable classes*** can be classified properly by the perceptron.
- In other words, the perceptron cannot classify nonlinearly separable classes. such as the one example shown here:

(clearly, the decision boundary couldn't separate classes).



- The most famous example for the limited capability of the perceptron was the ***XOR problem***, where the classes are:

A: [1 1] and [-1 -1] while B: [-1 1] and [1 -1]. That is similar to the above figure. i.e, data from the same class exist in opposite parts of the space.

- In general, any classification problem, where the classes are not linearly separable (nonlinearly separable) cannot be solved by the perceptron.
- Such problems are called linearly inseparable since no straight line can divide them successfully. Since we cannot divide them with a straight line, the perceptron will not be able to find any such line either; ***A single-layer perceptron cannot solve any problem that is linearly inseparable.***

- One solution is to make the output of the perceptron a nonlinear function of the inputs $\{x_n\}$. However, it is not always clear what kind of nonlinearity is needed for each problem. For example, in some problems, maybe introducing a term "$x_1.x_2.x_2$" can solve the problem, while in another problem another nonlinearity is needed.
- The ultimate solution for this dilemma is to make the neural network determines or learns the nonlinear expansion that it needs for each problem, from the training data given. To do that, the neural network must have nonlinear capabilities.
- In other words, the linear summation must be followed by nonlinear functions which produces nonlinear expansions for the inputs $\{x_n\}$ at the output. Thus, the decision boundary becomes a nonlinear function of $\{x_n\}$.

The Many-Perceptron Idea:

- To construct such a neural network, the first thought was to have many perceptrons, each is set up to identify a small, linearly separable section of the input. Then, to combine their outputs into another perceptron, which would produce a final indication of the class to which the input belongs.
- This works theoretically, for example, for the XOR problem, we could have percp1 separates between classes A1 [-1 1] , [1 1] , [-1 -1] and B1 [1 -1], with its output y1 equals 0 for A1 and 1 for B1.

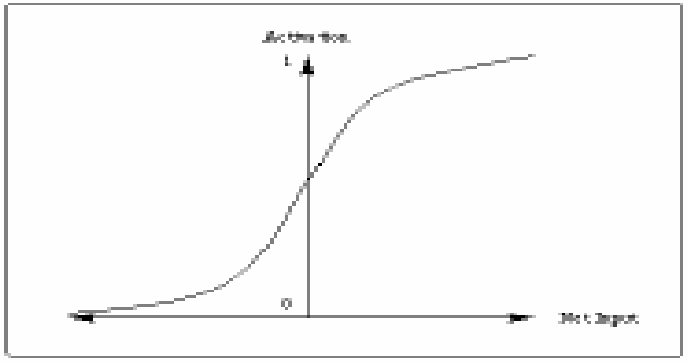
Then percp2 separates between classes A2 [-1 -1] ,[1 1] , [1 -1] and B2 [-1 1], with its output y2 equals 0 for A2 and 1 for B2.

- Thus, if the ***y1=1 or y2=1*** it is class-B, while iff ***y1=0 and y2=0*** then it is class-A.

This can be done by a third perceptron taking its inputs as y1 and y2, and doing a sum to get y3=y1+y2. Thus, if y3=1 we have class-B, while if y3=0 then it is class-A, and we can have a threshold=0.5 in this case on y3.

- However, we have solved this problem by common sense, and not by a true learning algorithm. In fact, Widrow's Many-Adalines (MaDalines) have used this approach, but uses some exhaustive trial and error technique to set the weights and thresholds for the perceptrons used.
- We can imagine how tedious this must be if we have a large number of inputs and a complex nonlinear decision boundary. Thus, the idea may be interesting, but it needs another formulation such that we can have an automatic algorithm to be able to find the weights and the thresholds.
- The correct approach appeared in 1986 with the "PDP" book. The approach relies on Widrow's "MaDaline", but with some modifications.
- The approach also relies on Widrow's LMS algorithm, which estimates the weights by minimizing a squared error criterion (as discussed earlier).
- The problem with the LMS approach is that the output has to be a differentiable function of the weights. This cannot be done with the hardlim functions.

- Thus, the final solution is to use the many-perceptron idea (MaDalines), but instead of using a hardlim function at the output of each neuron, we use an approximation of the hardlim, one that has a differentiation. For example, the logsig (logistic sigmoid), or the tansig (tanh function):



- The new neural network structure in this case is called the Multi-Layer Perceptron. Each layer is made of many perceptrons in parallel, each has a nonlinear, differentiable output. The next layer takes its inputs from the layer before, until the final perceptron output, which is required to be equal to the target value for each input.