

Neural Networks / Neural Computation (807G5)

Assignment 1

2nd February 2005

1. *Using Single Layer Networks for Classification*

Question 1.1

With a 2 dimensional Boolean array, there are 4 different possible input patterns (0,0), (0,1), (1,0) and (1,1).

With a single Boolean (2 state) output, there are 16 different possible configurations for the network. For example one possible network configuration would be, the classic logical AND and NAND would be represented as the following network:-

X_1	X_2	Y
0	0	0
0	1	0
1	0	0
1	1	1

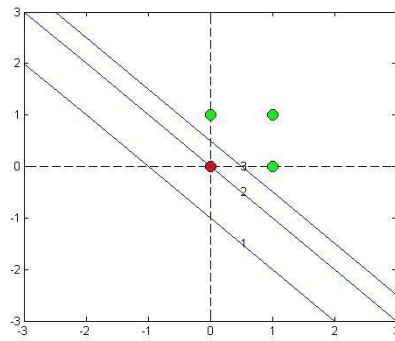
(figure 1.1.1 – Training of an AND gate)

X_1	X_2	Y
0	0	1
0	1	1
1	0	1
1	1	1

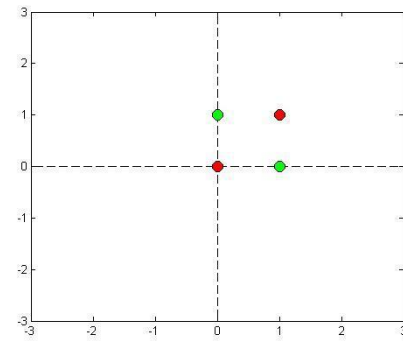
(figure 1.1.2 – Training of an NAND gate)

It should be noted however, that a number of network combinations can be considered to be “negated equivalents” of one another (in other words, from the purpose of classification, whether a class is denoted as -1 or as +1 is arbitrary).

Below is the output of a successful perceptron training. This is for a logical OR configuration.



(figure 1.1.3 – Training of an OR gate)



(figure 1.1.4 – Un-trainable XOR)

As you can see, the training process took 3 iterations (each blue line is the boundary classification denoted by the weights after each iteration).

The boundary classification is found by considering the output of the perceptron to be zero (halfway between the two classifications used of +1 and -1). This results in the below formula.

$$w_0 + w_1x_1 + w_2x_2 = 0 \quad \text{(formula 1.1)}$$

$$x_2 = \frac{-w_0 - w_1x_1}{w_2}$$

Of the 16 combinations, 14 were successfully trained in 6 iterations or less.

The remaining 2 were not trainable no matter what training rate or the no of iterations. These are the “negated equivalents” of the classic logical XOR (figure 1.1.4).

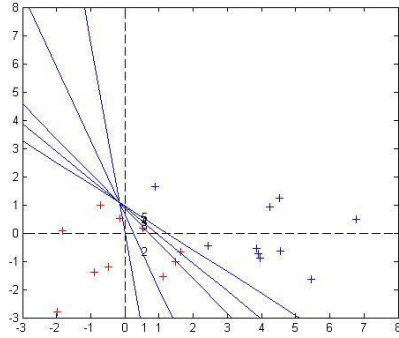
The illustration helps explain why the logical XOR is not trainable within a single perceptron. This is because the points are not linearly separable. In other words, there is no way of drawing a straight line classification boundary.

Question 1.2

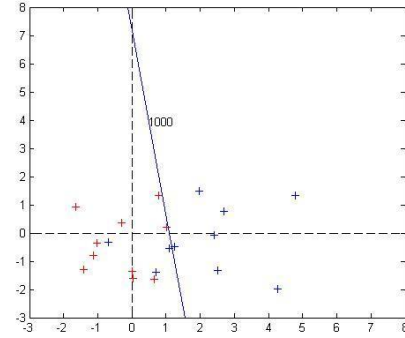
A single matrix was generated containing two distinct Gaussian distributions sets of 10 points each (plus an addition dimension of ones, for the bias).

This matrix was used as a set of inputs for the training of the perceptron, along with a “training” matrix requiring that the perceptron classify each set with either a +1 or -1 output.

Due to the nature of the Gaussian distributions used, the training data was linearly separable and as such the perceptron was able to train its weights to specify a classification boundary. On average only 6 training iterations were required (for a training rate of 0.5) to perform the necessary classification. Figure 1.2.1 shows the training steps towards a successful classification learning.



(figure 1.2.1 – Training a linearly separable data set)



(figure 1.2.2 – Training a non linearly separable data set)

A number of alternatives are available to the gradient ascent algorithms that were used in the perceptron training. For this particular classification, the weights for the entire training set were trained at once (using the standard equation below).

$$w_i(t+1) = w_i(t) - \eta \frac{\partial E(\mathbf{w}, t)}{\partial w_i} \quad (\text{formula 1.2})$$

Alternative methods could have been to step through each training example in the training set and adjust the weights after each parse; or to include “momentum”. However because of the simplicity of this data and the fact that it is a classification problem, this sort of approach was thought to be unnecessary.

Two different activation functions were examined. The simplest of these is the **sign** function (formula 1.3) which simply returns either +1 or -1 according to the sign of the sum of the weighted inputs. This provided quick and effective training results.

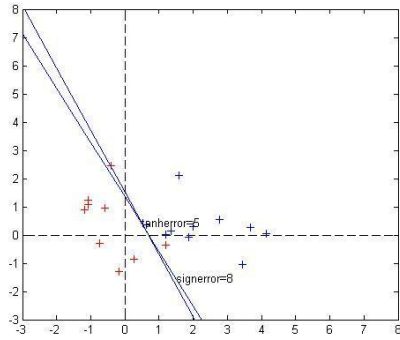
$$\text{sign}[y(\mathbf{x}, \mathbf{w})] = \text{sign}\left[\sum_{i=0}^m w_i x_i\right] \quad (\text{formula 1.3})$$

$$\tanh[y(\mathbf{x}, \mathbf{w})] = \tanh\left[\sum_{i=0}^m w_i x_i\right]$$

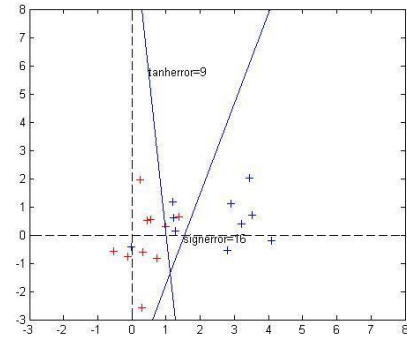
The second activation function examined was the **tanh** function. This provided very similar results with regards to no of iterations that it took in finding the linearly separable data.

However, when the two datasets were brought “closer together” to a point where “overlap” occurred and they were no longer linearly separable, the tanh function produced what appeared to be better results. This was especially true with lower training rates (see figures 1.2.3 and 1.2.4 for an example).

Because of the non-linearity of the sigmoidal function used, it allowed for a better “near match” classification (outputs would tend towards either +1 or -1 but not quite find the perfect match).



(figure 1.2.3 – Difference in results of a tanh and sign when trying to separate non-linearly separable sets –training rate = 0.5)



(figure 1.2.4 – Difference in results of a tanh and sign when trying to separate non-linearly separable sets –training rate = 0.1)

As can be seen in the graphical examples, when our perceptron was faced with a classification problem for a non linearly separable set of data it produces a closed approximation of the boundary.

2. *Using Single Layer Networks for Regression*

Question 2.1

A training set of 100 inputs was generated from integers between 1 and 200 (increasing in increments of 2). A corresponding training data was generated using the function shown below. With Δ equalling uniformly distributed values between -10 and 10 (acting as “noise” to the base function).

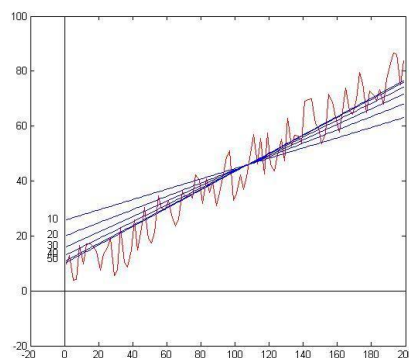
$$y = 0.4x + 3 + \Delta \quad \text{(formula 2.1)}$$

A single perceptron with 1 weighted input (and a bias) was then trained to find as close an approximation to this training data as possible.

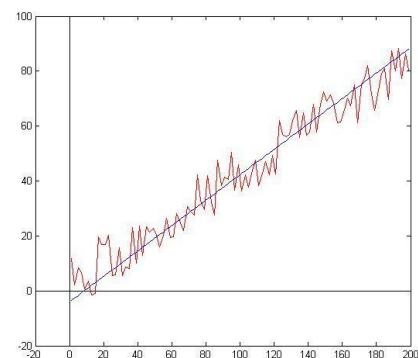
The data passed to the perceptron had to undergo certain “pre-processing” in order for the training to produce the desired results. Primarily, the input data was normalised to be between 0 and 1. This was for two reasons.

Firstly, without this normalisation the input data would be such that the training algorithm used (formula 1.2 above) would fail to converge the weights. The details would simply grow to infinity (and beyond).

Secondly, normalising the input data (and also therefore the training data) would “bring in line” the bias input. Without this, the bias term will never train at the same



(figure 2.1.1 training of formula 2.1)



(figure 2.1.2 result of training formula 2.1 after 100 iterations)

The “desired” formula is shown in red in figure 2.1.1. The output of the perceptron can also be seen, in blue (every 10 steps of the training algorithm is displayed – showing the convergence towards the desired results).

Figure 2.1.2 shows the output of a trained perceptron after 100 iterations. At this point the value of the weight was 0.389, and the value of the bias was 0.0167 (factored this equals 3.34).

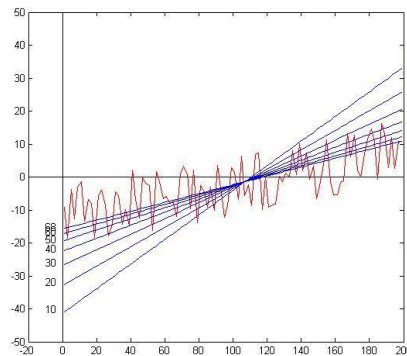
These values very closely match the numerical values found within the original formula (0.4 and 3). This can be understood further by examining the mechanism of the perceptron.

Simply put, the output (y) of the perceptron is the value of the weighted input ($0.4 * x$) plus the bias value (3) and the value Δ can be thought to represent “noise” around this function. So, the perceptron has been trained to “find” a linear equation within a set of data points containing small, distributed (Gaussian) noise.

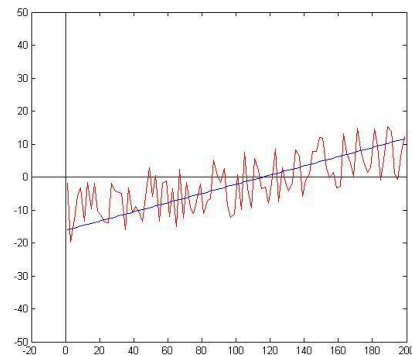
Interestingly, as the amount of noise was altered (for example to +25 -25), the perceptron still managed to provide a good estimated of the underlying formula. However, the bias value was notably less accurate, and therefore the basin of the error function appeared further from zero.

$$y = \frac{0.2x}{2} - 12 + \Delta \quad (\text{formula 2.2})$$

A second formula was also trained using the perceptron (see above). The results were similar. After 100 iterations, the perceptron produced a good approximation of the base formula.



(figure 2.1.3 training of formula 2.2)



(figure 2.1.4 result of training formula 2.2 after 100 iterations)

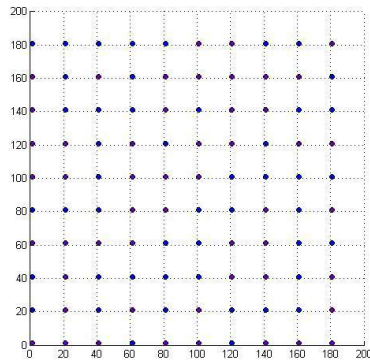
Question 2.2

A 2 dimensional matrix of 100 training inputs between 1 and 200 was generated along with training outputs corresponding to the formula below.

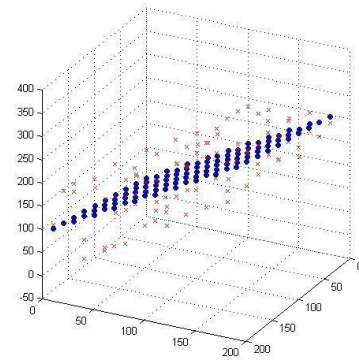
$$y = 0.4x_1 + 1.4x_2 + \Delta \quad (\text{formula 2.3})$$

This, as in question 2.1, included a set of uniformly distributed noise (Δ) but this time in the range of -100 to +100.

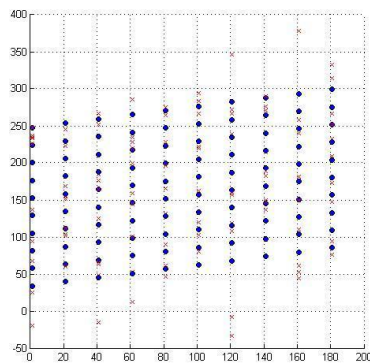
Training was performed using the same learning algorithm as has previously been used. (formula 1.2). Figures 2.2.1 to 2.2.4 show the resulting outputs after 100 iterations of learning. As there are now 2 inputs and 1 output to the perceptron, the graph is shown in 3 dimensions. As can be seen, the output of the perceptron can be represented as a hyperplane within the 3 dimensional space of the whole system.



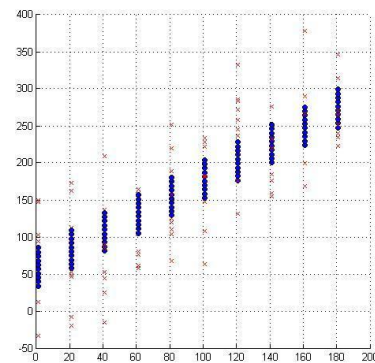
(figure 2.2.1 training of formula 2.3
 x_1 by x_2)



(figure 2.2.2 training of formula 2.3
3-dimensional view)



(figure 2.2.3 training of formula 2.3
 x_1 by y)



(figure 2.2.1 training of formula 2.3
 x_2 by y)

The final weights and bias values at the end of the training were:

$$\begin{array}{ll} w_1 & 0.3851 \\ w_2 & 1.4574 \\ \text{bias} & 0.0044 \end{array}$$

These weights (following the lessons learnt in question 2) can be said to correspond with the input factors of the underlying formula. This is formulated below (where w_0 is the bias):-

$$y = w_1 x_1 + w_2 x_2 + w_0 \quad (\text{formula 2.4})$$

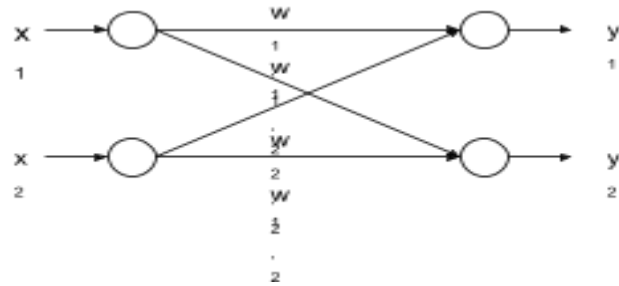
This is, of course, the standard formula for the output of a single perceptron and so it therefore becomes trivially obvious how such a formula as 2.3 can be trained within it.

This has been checked with similar linear functions such as formulas 2.5 and 2.6 and each time, the perceptron was trained to the same close mapping.

$$y = x_1 - 2.4x_2 + \Delta \quad (\text{formula 2.5})$$

$$y = 2x_1 + 0.9x_2 + 3 + \Delta \quad (\text{formula 2.6})$$

Finally, below is a network diagram for a 2 output single perceptron network. From the conclusions above it can be seen that 2 different linear functions (of the form specified in formula 2.4) could be learnt by this network. In 3 dimensional space this can be pictured as 2 separate hyper-planes. Additional output nodes would add the capacity for additional hyper-planes to be found within the space.



A network of this type could be used for separating two distinct functions from a “noisy” set of data points within large multi-dimensional space. Useful in, for example, processes such as sound or visual recognition.

2. *Appendix – MATLAB Source Code*

```
% NEURAL NETWORKS - ASSIGNMENT 1
%
% For easy readability each question
% was created within its own function.
% As a result, this source code is not
% a fully normalised low-redundancy set
% of functions.

% Main Function - Please un-comment the desired question
function[] = nnAs1(training)
    result = questionA1(training)
    %result = questionA2()
    %result = questionB1()
    %result = questionB2()
end

% QUESTION A.1 (parameter - choose which set of training outputs
% you wish to view - e.g. 7 = XOR)
function[result] = questionA1(training)
    % Define Constants
    noOfInputs = 2           % No of perceptron inputs
    noOfTrains = 4           % No of sets of training data
    trainingRate = 0.5       % Perceptron training rate
    err = 20                 % The current error value
    noOfIters = 0            % Current no of training iterations
    maxIters = 100          % No of training iterations before "quit"

    X = [0 0 1; 0 1 1; 1 0 1; 1 1 1]' % The INPUTS used for training
    W = zeros(1,noOfInputs+1)         % Initialise the WEIGHTS used for training

    % Set up the training TARGET outputs
    switch training
    case 1
        T = [-1, -1, -1, -1]
    case 2
        T = [-1, -1, -1, 1]
    case 3
        T = [-1, -1, 1, -1]
    case 4
        T = [-1, -1, 1, 1]
    case 5
        T = [-1, 1, -1, -1]
    case 6
        T = [-1, 1, -1, 1]
    case 7
        T = [-1, 1, 1, -1]
    case 8
        T = [-1, 1, 1, 1]
    case 9
        T = [1, -1, -1, -1]
    case 10
        T = [1, -1, -1, 1]
    case 11
        T = [1, -1, 1, -1]
    case 12
        T = [1, -1, 1, 1]
    case 13
        T = [1, 1, -1, -1]
    case 14
        T = [1, 1, -1, 1]
    case 15
        T = [1, 1, 1, -1]
    case 16
        T = [1, 1, 1, 1]
    end
```

```

% Prepare the graph
hold off
plot(0,0)
minAxes = -3
maxAxes = 3
axis([minAxes maxAxes minAxes maxAxes])
hold on
line([minAxes maxAxes],[0 0],'LineStyle','--','Color','k')
line([0 0],[minAxes maxAxes],'LineStyle','--','Color','k')

% Plot the data points
for i = 1:noOfTrains
    if (T(1,i) == -1)
        plot(X(1,i),X(2,i),'ro', 'MarkerEdgeColor','k', 'MarkerFaceColor','r', 'MarkerSize',10)
    else
        plot(X(1,i),X(2,i),'ro', 'MarkerEdgeColor','k', 'MarkerFaceColor','g', 'MarkerSize',10)
    end
end

% Perceptron training
Y = sign(W * X)
err = sumOfSqError(Y,T)
while ((err > 0) && (noOfIters < maxIters))
    noOfIters = noOfIters + 1
    W = W - ( trainingRate * (Y-T)*X' )
    Y = sign(W * X)
    err = sumOfSqError(Y,T)
end

% Draw classification boundary line
xbit = minAxes:1:maxAxes
ybit = (-W(1,3)-xbit*(W(1,1)))/W(1,2)
plot(xbit,ybit,'-b')
text(0.5,((-W(1,3)-(0.5*W(1,1))) / W(1,2)),num2str(noOfIters,'%1u'))

% End
result = noOfIters
hold off
end

```

```

% QUESTION A.2
function[result] = questionA2()
% Define Constants
noOfInputs = 2           % No of perceptron inputs
noOfTrains = 20          % No of sets of training data
trainingRate = 10        % Perceptron training rate
err = 20                 % The current error value
noOfIters = 0            % Current no of training iterations
maxIters = 1000          % No of training iterations before "quit"

% Set up training data
class1u = [0 0]
class1E = [1 0; 0 1]
class2u = [4 0]
class2E = [1 0; 0 1]

X = ones(noOfInputs+1,noOfTrains)
X(1:2,1:10) = mvnrnd(class1u,class1E,noOfTrains/2)'
X(1:2,11:20) = mvnrnd(class2u,class2E,noOfTrains/2)'
T = [ones(1,noOfTrains/2)*1 ones(1,noOfTrains/2)*-1]
W = zeros(1,noOfInputs+1)
% Initialise second set of weights to test sigmoidal activation
W2 = zeros(1,noOfInputs+1)

% Prepare Graph
hold off
plot(0,0)
minAxes = -3
maxAxes = 8
axis([minAxes maxAxes minAxes maxAxes])
hold on
line([minAxes maxAxes],[0 0],'LineStyle','--','Color','k')
line([0 0],[minAxes maxAxes],'LineStyle','--','Color','k')

% Plot data points
plot(X(1,1:10),X(2,1:10),'+r')
plot(X(1,11:20),X(2,11:20),'+b')

% Train Weights (for both sets of training data
Y = sign(W * X)
Y2 = tanh(W2 * X)          % for testing sigmoidal activation
err = sumOfSqError(Y,T)
err2 = sumOfSqError(Y2,T)  % for testing sigmoidal activation
while ((err > 0) && (noOfIters < maxIters))
    noOfIters = noOfIters + 1

    W = W - (trainingRate * (Y-T)*X')
    W2 = W2 - (trainingRate * (Y2-T)*X')

    Y = sign(W * X)
    Y2 = tanh(W2 * X)
    err = sumOfSqError(Y,T)
    err2 = sumOfSqError(Y2,T)
end

% Draw Classification Boundary
xbit = minAxes:1:maxAxes
ybit = (-W(1,3)-xbit*(W(1,1)))/W(1,2)
text(1.5,((-W(1,3)-(1.5*W(1,1)))/ W(1,2)),num2str(err,'sign error = %5.0f'))
plot(xbit,ybit,'-b')
% For testing sigmoidal activation
xbit = minAxes:1:maxAxes
ybit = (-W2(1,3)-xbit*(W2(1,1)))/W2(1,2)
text(0.5,((-W2(1,3)-(0.5*W2(1,1)))/ W2(1,2)),num2str(err2,'tanh error = %5.0f'))
plot(xbit,ybit,'-b')

% End
result = noOfIters
hold off
end

```

```

% QUESTION B.1
function[result] = questionB1()
% Define Constants
noOfInputs = 1           % No of perceptron inputs
noOfTrains = 100         % No of sets of training data
trainingRate = 0.005     % Perceptron training rate
err = 20                 % The current error value
noOfIters = 0            % Current no of training iterations
maxIters = 100           % No of training iterations before "quit"
factor = 200             % Scaling factor used for normalisation

% Set up training data
X = ones(noOfInputs+1,noOfTrains)*factor
X(1,:) = 1:2:200
V = rand(1,noOfTrains)*20 - 10
T = 0.2*X(1,:)/2 - 12 + V
X = X/factor
T = T/factor
W = rand(1,noOfInputs+1)

% Prepare Graph
hold off
plot(0,0)
minAxesX = -20
maxAxesX = 200
minAxesY = -50
maxAxesY = 50
axis([minAxesX maxAxesX minAxesY maxAxesY])
hold on
line([minAxesX maxAxesX],[0 0],'LineStyle','-','Color','k')
line([0 0],[minAxesY maxAxesY],'LineStyle','-','Color','k')

% Plot data points (as a line)
plot(X(1,:)*factor,T*factor,'-r')

% Train Weights
Y = W*X
err = sumOfSqError(Y,T)
while ((err > 0.1) && (noOfIters < maxIters))
    noOfIters = noOfIters + 1
    W = W - (trainingRate * ((Y-T)*X'))
    Y = W*X
    err = sumOfSqError(Y,T)
end
% Plot trained perceptron
plot(X(1,:)*factor,Y(1,:)*factor,'-b')

% End
result = noOfIters
hold off
end

```

```

% QUESTION B.2
function[result] = questionB2()

% Define Constants
noOfInputs = 2           % No of perceptron inputs
noOfTrains = 100         % No of sets of training data
trainingRate = 0.005     % Perceptron training rate
err = 20                 % The current error value
noOfIters = 0            % Current no of training iterations
maxIters = 500           % No of training iterations before "quit"
factor = 200             % Scaling factor used for normalisation

% Set up training data
X = ones(noOfInputs+1,noOfTrains)*factor
X(1,:) = [1:20:200 1:20:200 1:20:200 1:20:200 1:20:200 1:20:200 1:20:200 1:20:200]
X(2,:) = [ones(1,10)*1 ones(1,10)*21 ones(1,10)*41 ones(1,10)*61 ones(1,10)*81 ones(1,10)*101 ones(1,10)*121
ones(1,10)*141 ones(1,10)*161 ones(1,10)*181]
V = rand(1,noOfTrains)*200 - 100
T = 2*X(1,:) + 0.9*X(2,:) + 3*factor + V
X = X/factor
T = T/factor
W = rand(1,noOfInputs+1)

% Prepare graph
cla reset
hold off
hold on
plot3(X(1,:)*factor, X(2,:)*factor, T*factor,'rx')
grid on
axis square

% Train Weights
Y = W*X
err = sumOfSqError(Y,T)
while ((err > 0.1) && (noOfIters < maxIters))
    noOfIters = noOfIters + 1
    W = W - (trainingRate * ((Y-T)*X'))
    Y = W*X
    err = sumOfSqError(Y,T)
end

% Plot trained perceptron output
plot3(X(1,:)*factor, X(2,:)*factor, Y*factor,'bo', 'MarkerEdgeColor','k', 'MarkerFaceColor','b', 'MarkerSize',5)

% End
result = noOfIters
hold off
end

% Sum Of Squares Function
% Returns the sum of the squares of the difference
% between the two parameters
function[result] = sumOfSqError(Y,T)
    result = sum(sum((T-Y).^2))
    if isinf(result)
        result = 1000
    end
end
end

```