EMI Assignment 6:
Name: Harshit Vijayvargia
UFID: 19355645

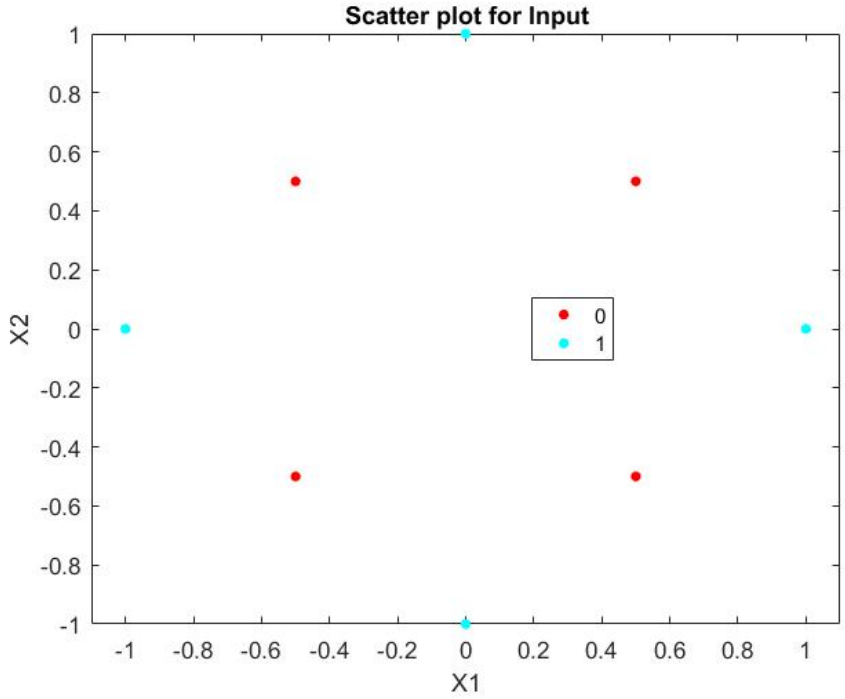
In this homework, we are implementing a neural network which can learn the pattern from the input and classify them into two classes (Binary Classification). Since our decision boundary can be nonlinear so neural networks will be the best way to perform this logistic regression.

Selecting parameters of the network before training:

1) Network Structure:

Initially I analyzed the input and try to guess the structure of my neural network.
The input given is:

X1	X2	Class
1	0	1
0	1	1
-1	0	1
0	-1	1
0.5	0.5	0
-0.5	0.5	0
0.5	-0.5	0
-0.5	-0.5	0



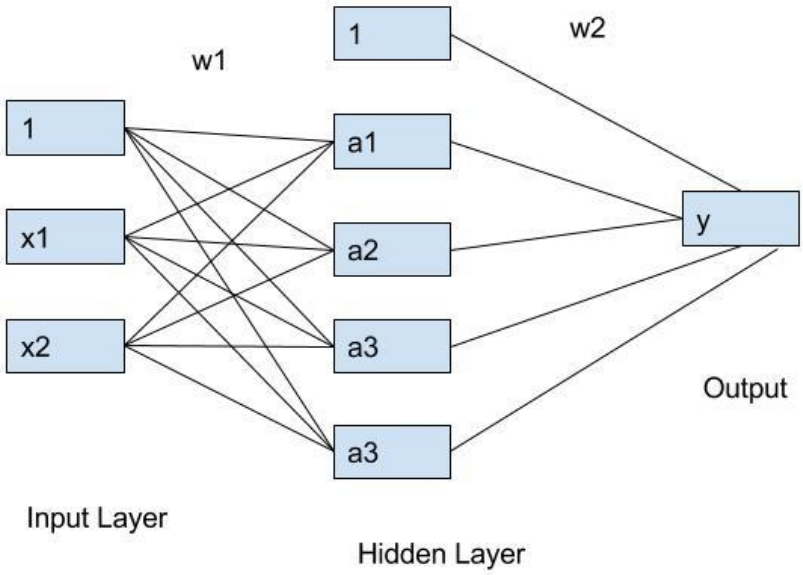
From the scatter plot of input, it can be observed that there are total 8 points, 4 at the periphery of square (Class :1) and 4 other at the midpoint of the sides of square (Side Length=2). The center of the square is origin.
There is no assured method to determine the number of hidden elements and hidden layers in the network but seeing the plot it can be observed that the decision boundary is non-linear. So there are non-linear variables involved. I will start with 2 hidden elements initially and will increase them if I don't see good performance. I will take one hidden layer initially with 2 hidden elements and will evaluate the performance of my neural network on the given data.

2) **Parameters for training**: There are other parameters too which will be employed while training the neural network such as:
   a) Weight initialization: I took random weights as it is observed that if we assign weights as zeros then hidden layer becomes redundant as parameters don't get updated during training.
   b) Step size: It should not be too small while not too large. I took 0.01 initially.
   c) Cost function: I chose cross entropy as my cost function. MSE (Mean Square error) was fine in case when we were doing linear regression but in our case output function is non-linear so if we plug in mean square error as our cost function which is a non–convex function then it might be a case that it does not converges to global minima.

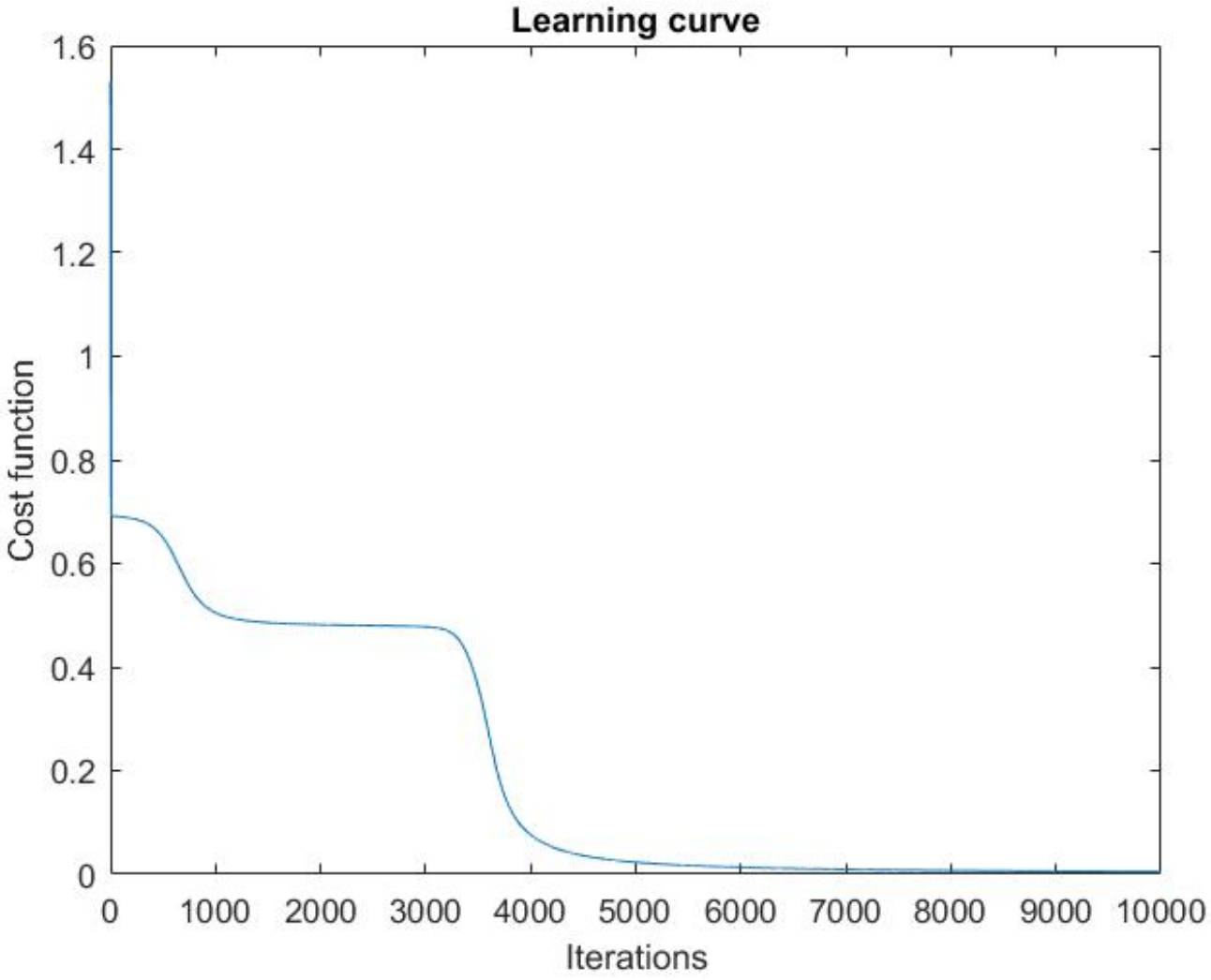
      Training :

      For 2 neurons the performance was not good. So, I increased the number of neurons in hidden layer to 3. It was still not good so I increased it further to 4. My cost function converged when I used 4 neurons and one hidden layer in 10000 iterations with a step size = 0.5. There was no need to further increase number of neurons or layers as it would have increased computation time also since the training set has 8 samples taking more neurons in hidden layer would might create a possibility of overfitting.
      The final architecture I used is shown below with the learning curve and decision boundary. The ones (1) are the bias added at each step to do regularization. Regularization is done to avoid data overfitting of data.

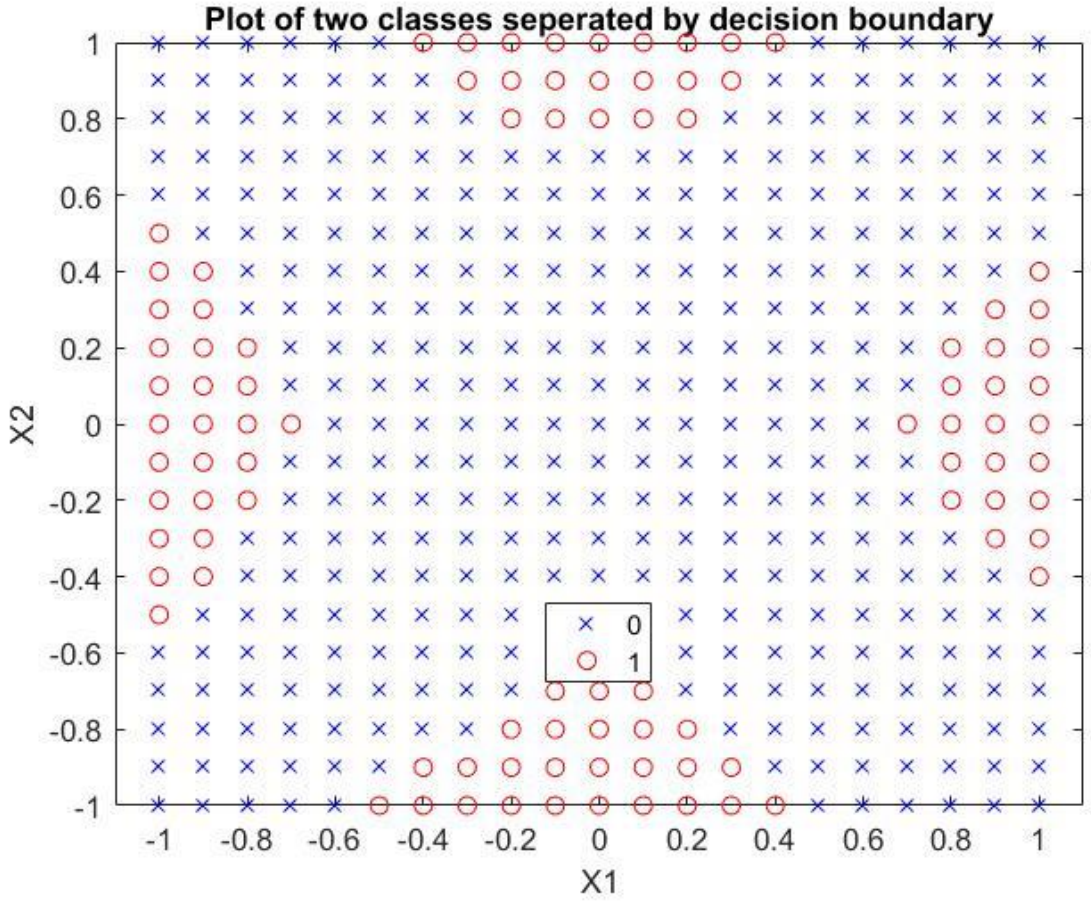
Network Architecture:



The **Learning curve** is shown below which is monotonically decreasing:

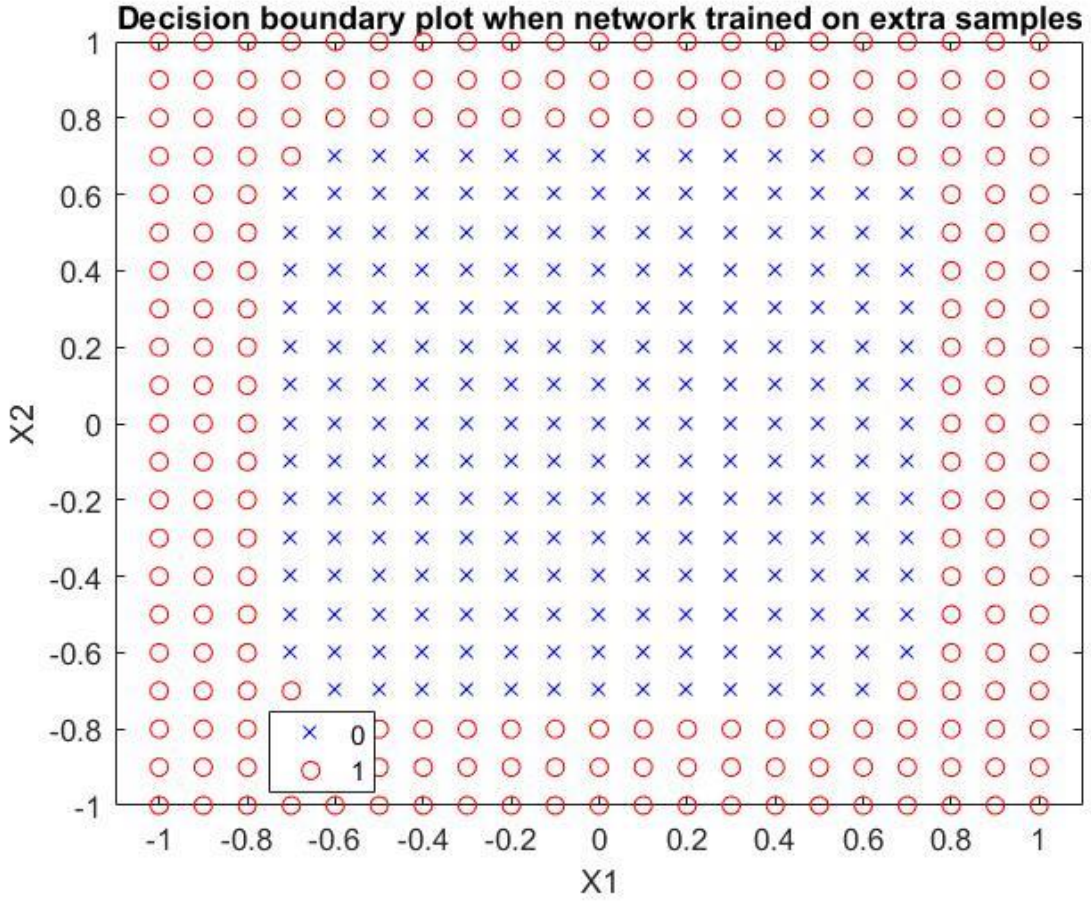


After training, neural network on given 8 training samples. The decision boundary obtained is: It represents two classes 0 and 1 with a clear distinction between them.



I expected the decision boundary to be a square when I started but since training data has 8 samples so neural network will try to fit data as per given samples and the decision boundary above is also consistent with the data given.

If I would have given more samples (1,1), (-1,1) , (1,-1) , (-1,-1) then my neural network is giving the following decision boundary because it learned a different pattern now. This was the pattern I expected as it is given that points closer to axes fall in one class. To conclude, we can make a guess what decision boundary may look like from input data but can't be sure unless training samples are good enough to fix one type of decision boundary.



Improving generalization accuracy : Generalization is done to avoid overfitting of data. For better performance, we desire that whenever there is a small change in input, this change should not affect our output. To improve generalization, we can use **double back propagation**. It is done during training where during backpropagation we add an additional term which is a function of jacobian. It is observed that double backpropagation creates smaller weights which causes output of the neuron to spend more time in linear region. There are other methods too like : **Increasing the size of neural networks** , as with large network , number of variables increases which avoid data overfitting. We can also use **regularization and cross validation** . It is also observed if training set is much smaller then network size then overfitting problem can be avoided.

Code of algorithm :

```
clear all;
load('EMIHW6.mat');   %Loading file with input
X = EMIHW6;

%Setting Network structure parameters

L1 = 2,L2=4,L3=1;     % L1,L2,L3 : size of input,hidden and output layer
without bias

total_size= L2*(L1 +1 ) + L3* ( L2 + 1);     % Defining parameter vector size
theta = rand(total_size,1);             % Initialising random weights
Y = X(:,3);                             % output
Parameter1 = reshape(theta((1:L2*(L1+1)),:),[L2 L1+1]);                     %
Generating weight matrices
Parameter2 = reshape(theta(L2*(L1+1)+1:L2*(L1+1)+L3*(L2+1),:),[L3 L2+1]);
for iteration =1:10000
    output_mat = zeros(size(Y,1),1);         %Matrix for storing output
    Accum_par1 =zeros(L2,L1+1);              % Storing accumulated values
after each batch is passed
    Accum_par2 = zeros(L3,L2+1);

    for i =1:8                 % 8 samples in training batch
        Lone = X(i,(1:2))';       % Lone, Ltwo, Lthree layers without bias
        Ltwo = zeros(L2,1) ;
        Lthree = zeros(L3,L3);

        % Forward Propogation ,A1,A2,A3 layers with bias

        [A1,A2,A3,output] = forwardprop(Lone,Parameter1,Parameter2);  %Lone
and Parameter1 and Parameter2 passed as arguement for

        output_mat(i,:) = output;      % output stored
        Ltwo = A2(2:size(A2,1),:);
        Lthree = A3;

        %Backward Propogation

        e3 = output - Y(i) ;
        [e2] = backward_prop(e3,Parameter2,A2);

        Accum_par1 = Accum_par1 + e2(2:size(e2, 1))*A1';
        Accum_par2 = Accum_par2 + e3*A2';

    end

    % calculating D1 & D2 which are partial dervatives w.r.t cost function

    D1 = Accum_par1*(1/size(X,1));
    D2 = Accum_par2*(1/size(X,1));


       step_size = 0.5;                      %setting step_size

       %Updating parameters
    Parameter1 = Parameter1 - step_size.*D1;
    Parameter2 = Parameter2 - step_size.*D2;
```

```
%Forward propagation function


function [A1,A2,A3,output] = forwardprop(Lone,Parameter1,Parameter2)
 A1 = [1;Lone];
 Z2 = Parameter1*A1;
 SZ2 =  sigmg(Z2);
 A2 = [1;SZ2];
 Z3 = Parameter2*A2;
 A3 =  sigmg(Z3);
 output = A3;
end

%Backward propagation function


function [e2] = backward_prop(e3,Parameter2,A2)
e2 = Parameter2'*e3.*(A2.*(1- A2));
end

%sigmoid function

function sigmx = sigmg(X)
sigmx = 1./(1+exp(-X));
end


%Generating Decision boundary

  X1 = randi([-1,1]*10^1,[100000,2])/10^1;
  output_mat = X1;
for i =1:size(X1,1)
 Lone = X1(i,(1:2))';       % Lone, Ltwo, Lthree layers without bias
 Ltwo = zeros(L2,1) ;
 Lthree = zeros(L3,L3);

[A1,A2,A3,output] = forwardprop(Lone,Parameter1,Parameter2);
output_mat(i,3) = output;
if output_mat(i,3) >= 0.5
    output_mat(i,4)=1;
else
    output_mat(i,4)=0;
end
end
gscatter(output_mat(:,1),output_mat(:,2),output_mat(:,4),'br','xo');
```