**QUESTION 1**

1. **You are asked to classify the actual data for 1 neuron, for that you will need to generate 2 classes of random data. You can do this in 2 ways**

****

1. You can specify 2 close centers and randomly generate elements from these 2 different classes (but creating margin won’t be that easy).

OR

1. 2 parallel lines by drawing margin between the top and bottom of these lines of data

**Solution to Part I**

This problem is about creating two sets of random samples and with given coordinate values find the decision boundaries between two data sets. This can be done first creating two groups of data samples and then using the line equation, corresponding points can be classified to be either in group one or group second.

**Explanation:**

While dealing with first part of the problem, using Matlab functionalities we generate random samples that belong to say feature 1 and to generate a boundary, we use line Equation.

where m is the slope of the line with .

and with the help of given two coordinate values , and ,through which the line passes, we generate a line that divides the points which are above and below the generated sample data .

Now if the generated sample is over the line, then the sample is accepted ,otherwise we try to find the value of y until the point is over the line.

**Code Snippet1:** This code snippet generates the samples that belong to Feature 1

The first point on which the line passes

x0 = -20.55;

y0 = -0.2771;

The second point on which the line passes

x1 = 0.6928;

y1 = 10.35;

|  |
| --- |
| samples = zeros(2,noSamples);  for index = 1:noSamples  x = width\*(2\*rand()-1);  y = width\*(2\*rand()-1);  yhat = (y1-y0)/(x1-x0)\*(x-x0)+y0;  while y < yhat  % The point is not over the line so we generate y again.  y = width\*(2\*rand()-1);  end  samples(1,index) = x;  samples(2,index) = y;  end |

Similarly, while generating the samples that belong to feature 2, we again use equation of line and with given two coordinate values , and ; . through which the line passes, we generate a line that divides the points which are above and below the generated sample data.

If the generated sample is lower the line then the sample is accepted. otherwise, we will try to find the value of y untill the point is lower the line.

**Code Snippet2:** This code snippet generates the samples that belong to Feature 2

The first point on which the line passes

x0 = -0.692;

y0 = -10.35;

The second point on which the line passes

x1 = 20.55;

y1 = 0.2771;

|  |
| --- |
| samples = zeros(2,noSamples);  for index = 1:noSamples  x = width\*(2\*rand()-1);  y = width\*(2\*rand()-1);  yhat = (y1-y0)/(x1-x0)\*(x-x0)+y0;  while y > yhat  % The point is not lower the line so we generate y again.  y = width\*(2\*rand()-1);  end  samples(1,index) = x;  samples(2,index) = y;  end |

Using the above code , all sample point poistions being in feature 1 or feature 2 can be determined easily.

After testing all the generated samples in both the features, feature 1 and feature 2 through the test condition, two lines can be generated. **Parallel Lıne 1** that passes through points **,** and **,** represents a boundary for samples that belong to feature 1 and are above the line. Similarly **Parallel Lıne 2** that passes through points **,** and **,**  represents a boundary for samples that belong to feature 2 and are below the line.

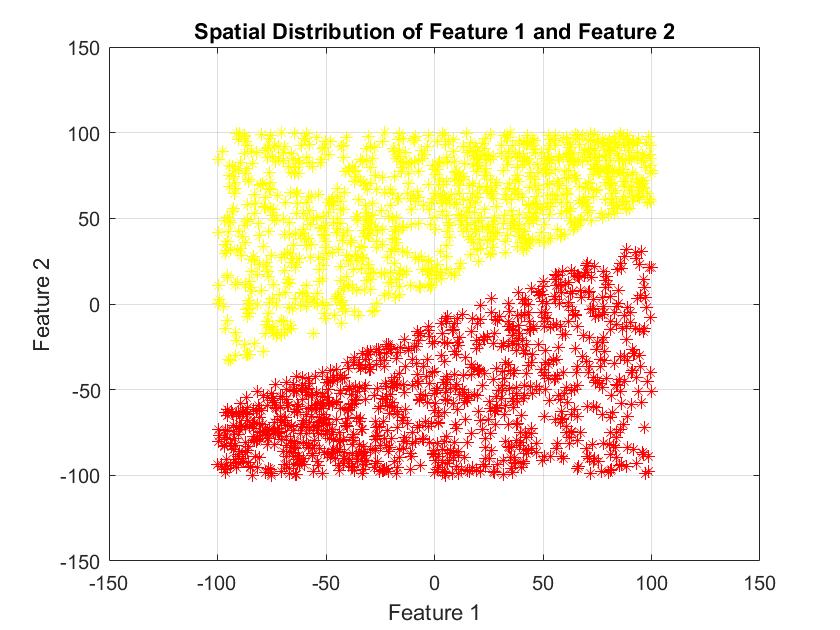


Figure 1: Two datasets with

1. **Gıven two generated datasets , you are supposed to classify the samples using Single Neuron Perceptron. You can take your training set as 75% of whole set, 15% for validation and 15% for test data**.

**Answer to Part II**

Using Sıngle Neuron Perceptron and initialising the weights , we train the perceptron with 75% of data in hand (feature1 and feature 2 samples). İn the following code, we generate a sample of size 1000 wherein we chose samples randomly from feature 1 or feature 2 classes .Simultaneously, we create an array that represents the class of sample. Further, we implemented two functions, that is

* generateFeature1Samples()
* generateFeature2Samples()

where each function generates samples for its specific class (feature1 or feature2).How we generate the samples is explained in part 1. We also implement updatePerceptronWeights()

function in which we compute the weights of the perceptron we generated for the sample sequence expleined above. Finally, we plot the seperator line using the weights we obtain. More can be seen in the snippet code

|  |
| --- |
| noSamples = 1000;  samples = zeros(2,noSamples);  targets = zeros(1,noSamples);  for i = 1:noSamples  if rand() < 0.5  samples(:,i) = generateFeature1Samples(1);  targets(i) = 1;  else  samples(:,i) = generateFeature2Samples(1);  targets(i) = 2;  end  end    %generateFeature1Samples and generateFeature2Samples are the functions %which was used in part 1 to generate the samples.  figure(2);  clf;  plot(samples(1,targets==1),samples(2,targets==1), 'y\*');  hold on;  plot(samples(1,targets==2),samples(2,targets==2), 'r\*');  xlabel('Feature 1');  ylabel('Feature 2');  title('Spatial Distribution of Feature 1 and Feature 2');  axis([-150 150 -150 150])  grid on;    W = [1;1;1];  W = updatePerceptronWeights(samples, targets, W);  % plot the perceptron line using the perceptron weights.  x = -150:0.1:150;  y = -(W(1) + W(2) \* x) / W(3);  plot(x, y, '-b'); |

In the following figure, we ilusturate the samples and the seperator line we obtain. Herein, it is important to note that the euqation of the seperator line depends on the randomness of samples. Thus, the number of possible seperator lines is infinite. We have just illistrated one of them below. Further, from the figure, we notice that the two data sets (yellow: Feature 1 and red: Feature 2) are seperable.

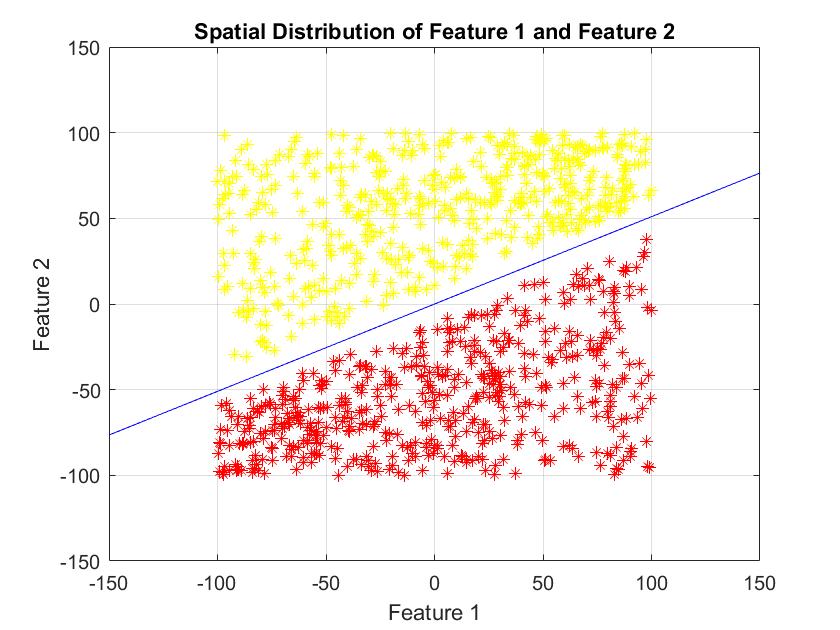


Figure 2: Linearly separable datasets

1. **Using the already generated data samples, add noise to sample points with noise level varying from zero to some particular point (user defined value).**

**This can be done by changing the mean and variance (in terms of standard deviation) of the new generated data points. (As given in the question) and add these points to create noisy data.**

**Answer to Part III**

Noise can be described as an unwanted or unrelated data which can reduce the signal or data quality. In this interesting part of the homework, we are supposed to classify or draw a perceptron on the data with different levels of noise and simultaneously with different values of learning rate and Epoch values, calculate the decision error (i.e. difference between target and observed value) and perform the classification.

While creating random samples which belong to feature 1 and feature 2, usually samples are created with given mean and variance. Similarly while creating another signal (noisy) and adding it to already presenting signal, we create it with different mean and variance. We change the mean and variance values. While doing this work, we created arrays of means and variances and while creating noisy signals, we looped over the mean and variance matrices to add different levels of noise to two different sample sets. Here again while training 75% of data was data,15% for validation and 15% for testing (which is expressed in numerical values e,g with total of 1000 samples, 700 samples are used for training,150 for testing and 150 for validation).

While testing with varying learning rates and epoch values, we checked for the accuracy of the model. Accuracy is given in terms of Decision error values.

Weights are updated continuously for the given input sample values

W = updatePerceptronWeights(traningSamples, traningTargets, W);

where the implementation code of updatePerceptronWeights() is given below

|  |
| --- |
| %%  % This function finds the values of perceptron weights from feature groups.  %  function [W] = updatePerceptronWeights(mixedFeatures, targets, prevW)  if size(mixedFeatures,2) ~= numel(targets)  error('The size of the targets and the number of samples does not matches!');  end  W = prevW;  for i = 1:numel(targets)  S = [1; mixedFeatures(1,i); mixedFeatures(2,i)];  target = targets(i);  if target == 1  % For the first group  if sign(W'\*S) >= 0  % Update the weights using Feature 1.  W = W - S;  end  elseif target == 2  % For the second group  if sign(W'\*S) <= 0  % Update the weights using Feature 2.  W = W + S;  end  else  error('Unknown target!')  end  end  end |

At different levels of noise, with different learning rate and epoch values, decision errors were calculated and the separability between noisy feature samples was done.

**Snippet code**: Add noise to training samples

|  |
| --- |
| %% TRANING SAMPLES  traningNoSamples = 7000; % NUMBER OF TRANING SAMPLES  traningSamples = zeros(2,traningNoSamples);  traningTargets = zeros(1,traningNoSamples);  for i = 1:traningNoSamples  if rand() < 0.5  traningSamples(:,i) = generateFeature1Samples(1) + ...  noiseSTD1 \* randn(2,1) + noiseMean1;  traningTargets(i) = 1;  else  traningSamples(:,i) = generateFeature2Samples(1) + ...  noiseSTD2 \* randn(2,1) + noiseMean2;  traningTargets(i) = 2;  end  end |

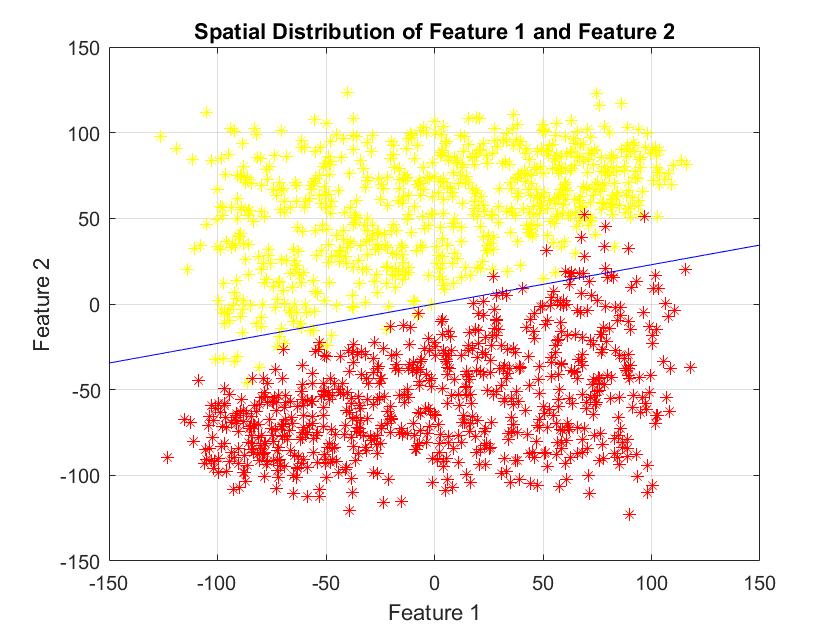


Figure 3: Noisy data classification

|  |
| --- |
| Variance = 0  Mean = [0; 0]  Epoch = 1  Total decision error = 0  Learning rate = 0.000000 |
| Variance = 0  Mean = [10; -10]  Epoch = 1  Total decision error = 10  Learning rate = 0.066667 |
| Variance = 10  Mean = [-10; -10]  Epoch = 1  Total decision error = 0  Learning rate = 0.000000 |
| Variance = 100  Mean = [-10; 10]  Epoch = 1  Total decision error = 19  Learning rate = 0.126667 |
| Variance = 100  Mean = [0; 0]  Epoch = 1  Total decision error = 7  Learning rate = 0.046667 |
| Variance = 100  Mean = [-10; 10]  Epoch = 1  Total decision error = 19  Learning rate = 0.126667 |

**CODE FILE**

clear all;

clc;

noSamples = 1000;

feature1 = generateFeature1Samples(noSamples);

feature2 = generateFeature2Samples(noSamples);

%% ------------------------------------------------------------------------

% FIGURE 1

% -------------------------------------------------------------------------

figure(1);

clf;

plot(feature1(1,:),feature1(2,:), 'y\*');

hold on;

plot(feature2(1,:),feature2(2,:), 'r\*');

xlabel('Feature 1');

ylabel('Feature 2');

title('Spatial Distribution of Feature 1 and Feature 2');

axis([-150 150 -150 150])

grid on;

%% ------------------------------------------------------------------------

% FIGURE 2

% -------------------------------------------------------------------------

noSamples = 1000;

samples = zeros(2,noSamples);

targets = zeros(1,noSamples);

for i = 1:noSamples

if rand() < 0.5

samples(:,i) = generateFeature1Samples(1);

targets(i) = 1;

else

samples(:,i) = generateFeature2Samples(1);

targets(i) = 2;

end

end

figure(2);

clf;

plot(samples(1,targets==1),samples(2,targets==1), 'y\*');

hold on;

plot(samples(1,targets==2),samples(2,targets==2), 'r\*');

xlabel('Feature 1');

ylabel('Feature 2');

title('Spatial Distribution of Feature 1 and Feature 2');

axis([-150 150 -150 150])

grid on;

W = [1;1;1];

W = updatePerceptronWeights(samples, targets, W);

% plot the perceptron line using the perceptron weights.

x = -150:0.1:150;

y = -(W(1) + W(2) \* x) / W(3);

plot(x, y, '-b');

%% ------------------------------------------------------------------------

% FIGURE 3

% -------------------------------------------------------------------------

%% SIMULATION PARAMETERS

variance = [0, 10, 100, 1000, 10000];

meanValues = [-10, -10, 0, 10, 10;

-10, 10, 0, -10, 10];

expectedLearningRate = 0.35;

for varIndex = 1:numel(variance)

for meanIndex = 1:size(meanValues,2)

noiseVariance1 = variance(varIndex);

noiseSTD1 = sqrt(noiseVariance1);

noiseMean1 = meanValues(:,meanIndex);

noiseVariance2 = variance(varIndex);

noiseSTD2 = sqrt(noiseVariance2);

noiseMean2 = meanValues(:,meanIndex);

% The initial value of weight. It is chosen randomly.

W = [1; 1; 1];

epochIndex = 0;

while(true)

epochIndex = epochIndex + 1;

%% TRANING SAMPLES

traningNoSamples = 7000; % NUMBER OF TRANING SAMPLES

traningSamples = zeros(2,traningNoSamples);

traningTargets = zeros(1,traningNoSamples);

for i = 1:traningNoSamples

if rand() < 0.5

traningSamples(:,i) = generateFeature1Samples(1) + ...

noiseSTD1 \* randn(2,1) + noiseMean1;

traningTargets(i) = 1;

else

traningSamples(:,i) = generateFeature2Samples(1) + ...

noiseSTD2 \* randn(2,1) + noiseMean2;

traningTargets(i) = 2;

end

end

%% VALIDATION SAMPLES

validationNoSamples = 150; % NUMBER OF VALIDATION SAMPLES

validationSamples = zeros(2,validationNoSamples);

validationTargets = zeros(1,validationNoSamples);

for i = 1:validationNoSamples

if rand() < 0.5

validationSamples(:,i) = generateFeature1Samples(1) + ...

noiseSTD1 \* randn(2,1) + noiseMean1;

validationTargets(i) = 1;

else

validationSamples(:,i) = generateFeature2Samples(1) + ...

noiseSTD2 \* randn(2,1) + noiseMean2;

validationTargets(i) = 2;

end

end

%% TEST SAMPLES

testNoSamples = 150; % NUMBER OF TEST SAMPLES

testSamples = zeros(2,testNoSamples);

testTargets = zeros(1,testNoSamples);

for i = 1:testNoSamples

if rand() < 0.5

testSamples(:,i) = generateFeature1Samples(1) + ...

noiseSTD1 \* randn(2,1) + noiseMean1;

testTargets(i) = 1;

else

testSamples(:,i) = generateFeature2Samples(1) + ...

noiseSTD2 \* randn(2,1) + noiseMean2;

testTargets(i) = 2;

end

end

%% UPDATE WEIGHTS

% Plotting the traning samples and finding the weights of the perceptron.

W = updatePerceptronWeights(traningSamples, traningTargets, W);

%% DECISIONS USING TEST SAMPLES

decision = zeros(1,testNoSamples);

for i = 1:testNoSamples

S = [1; testSamples(1,i); testSamples(2,i)];

predicate = sign(W'\*S);

if (predicate <= 0)

decision(i) = 1;

else

decision(i) = 2;

end

end

%% STATISTICS

totalDecisionError = sum(decision ~= testTargets);

learningRate = totalDecisionError/testNoSamples;

if learningRate > expectedLearningRate

continue

end

%% PLOT STATISTICS FOR INFORMATION

fprintf('------------------------------\n');

fprintf('Variance = %d\n', variance(varIndex));

fprintf('Mean = [%d; %d]\n', meanValues(1,meanIndex), meanValues(2,meanIndex));

fprintf('Epoch = %d\n', epochIndex);

fprintf('Total decision error = %d\n', totalDecisionError);

fprintf('Learning rate = %f\n', learningRate);

break;

end

end

end

%%

% This function finds the values of perceptron weights from feature groups.

% An example illustration is given as

% <code>

% </code>

function [W] = updatePerceptronWeights(mixedFeatures, targets, prevW)

if size(mixedFeatures,2) ~= numel(targets)

error('The size of the targets and the number of samples does not matches!');

end

W = prevW;

for i = 1:numel(targets)

S = [1; mixedFeatures(1,i); mixedFeatures(2,i)];

target = targets(i);

if target == 1

% For the first group

if sign(W'\*S) >= 0

% Update the weights using Feature 1.

W = W - S;

end

elseif target == 2

% For the second group

if sign(W'\*S) <= 0

% Update the weights using Feature 2.

W = W + S;

end

else

error('Unknown target!')

end

end

end

%%

% This function spatially generates the samples that belong to Feature 1.

% An example usage is illustrated below.

% <code>

% noSamples = 1000;

% [xVal, yVal] = generateFeature1Samples(noSamples);

% </code>

function [samples] = generateFeature1Samples(noSamples)

width = 100;

% The first point on which the line passes

x0 = -20.55;

y0 = -0.2771;

% The second point on which the line passes

x1 = 0.6928;

y1 = 10.35;

% We generate the samples in the following. If the generated sample is over

% the line then the sample is accepted. otherwise, we will try to find the

% value of y untill the point is over the line.

samples = zeros(2,noSamples);

for index = 1:noSamples

x = width\*(2\*rand()-1);

y = width\*(2\*rand()-1);

yhat = (y1-y0)/(x1-x0)\*(x-x0)+y0;

while y < yhat

% The point is not over the line so we generate y again.

y = width\*(2\*rand()-1);

end

samples(1,index) = x;

samples(2,index) = y;

end

end

%%

% This function spatially generates the samples that belong to Feature 2.

% An example usage is illustrated below.

% <code>

% noSamples = 1000;

% [xVal, yVal] = generateFeature2Samples(noSamples);

% </code>

function [samples] = generateFeature2Samples(noSamples)

width = 100;

% The first point on which the line passes

x0 = -0.692;

y0 = -10.35;

% The second point on which the line passes

x1 = 20.55;

y1 = 0.2771;

% We generate the samples in the following. If the generated sample is

% lower the line then the sample is accepted. otherwise, we will try to find the

% value of y untill the point is lower the line.

samples = zeros(2,noSamples);

for index = 1:noSamples

x = width\*(2\*rand()-1);

y = width\*(2\*rand()-1);

yhat = (y1-y0)/(x1-x0)\*(x-x0)+y0;

while y > yhat

% The point is not lower the line so we generate y again.

y = width\*(2\*rand()-1);

end

samples(1,index) = x;

samples(2,index) = y;

end

end

**Question 2**

You are required to prepare the framework of your framework for multi-layer perceptron which will form the basis of your project.

For this question, your code must fulfill the following requirements.

i.) Design an operable design for different loss and activation functions.

ii.) Depending on your programming language, if there is no automatic memory cleanup, memory allocation and so on. Perform operations.

iii.) Insert the interface (or function entries) that request the number of hidden layers and the number of neurons per layer.

iv.) For the input dataset, make the connections between the size detection, the corresponding input layer and the Hidden Input - 1. Hidden layer together with the data structure that holds the weights; complete this to the “last hidden layer’’ link.

v.) Enable the ability to add different activation functions for the last hidden layer.

**Answer**

Multilayer Perceptron’s (MLP) are composed of more than one perceptron. They are composed of an input layer to receive the input, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP. MLPs with one hidden layer are capable of approximating any continuous function.

As per the question we are supposed to design a network with varying number of hidden layers and with varying number of neurons in each of the hidden layer. In regards to activations functions, we are supposed to change the activation functions both in hidden layer and also at the output layer. While calculating the loss function at the output layer , different loss functions can be used.

In three layer network , it is divided in three main parts

1. Build model
2. Feedforward propagation (Training the model)
3. Loss function calculation
4. Backward Propagation (Weight updation)

While training the model,we built the model with varying number of hidden layer neurons, the initial weight and bias dimensions were calculated. İn the function, there is a provision to change the number of neurons in hidden layers.

|  |
| --- |
| model = build\_model(X,10,2)  **def** build\_model(X,hidden\_nodes,output\_dim=2):  model = {}  input\_dim = X.shape[1]  model[**'W1'**] = np.random.randn(input\_dim, hidden\_nodes) / np.sqrt(input\_dim)  model[**'b1'**] = np.zeros((1, hidden\_nodes))  model[**'W2'**] = np.random.randn(hidden\_nodes, hidden\_nodes) / np.sqrt(hidden\_nodes)  model[**'b2'**] = np.zeros((1, hidden\_nodes))  model[**'W3'**] = np.random.randn(hidden\_nodes, output\_dim) / np.sqrt(hidden\_nodes)  model[**'b3'**] = np.zeros((1, output\_dim))  **return** model |

The second step invloves the Feedforward Propagation phase which includes training the model. Inputs are the initialsed model values and the training dataset. Two different activation functions were used to calcualted the activated output both in the hidden layers as well as at output layers.

Both tanh and Relu functions were used. One function has been commented out to pay the way for another one to give output at both the hidden neuron and output layer.

|  |
| --- |
| z1, a1, z2, a2, z3, out = feed\_forward(model, X)  **def** feed\_forward(model, x):  W1, b1, W2, b2, W3, b3 = model[**'W1'**], model[**'b1'**], model[**'W2'**], model[**'b2'**], model[**'W3'**], model[**'b3'**]  *# Forward propagation* z1 = x.dot(W1) + b1  *#a1 = np.tanh(z1)* a1 = relu(z1)  z2 = a1.dot(W2) + b2  a2 = relu(z2)  z3 = a2.dot(W3) + b3  exp\_scores = np.exp(z3)  out = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=**True**)  **return** z1, a1, z2, a2, z3, out |

Third step invloves loss function calculation (if the difference between expected and target value difference is not equal to zero). While training the model, using Mean Square Error formula at the output layer, loss functions were calculated. If the difference between expected and target value difference is not equal to zero, then the function backwardpropagtion to make is near or equal to zero.

Updation step (BackwardPropagation) invloves updating the weight values. This is done by taking the derivates of the loss function with respect to the weights which effect the loss function most (i.e those weight values through which the error rate doesn’t reach zero). To avoid the overfitting problem of model, another hyperparameter called was used i.e regularization parameter,.

Derivates with respect to weight parameters are taken and weights are updated. This step involves Hyperparameter called learning rate. Activation function derivates used in backpropagation a) Relu derivative and b) tanh derivative.

With different learning rates,the algorithm is run until the specific value of learning rate provides a point where the mean square error is equal or global maxima is found.Two different derivative activation functions were used

|  |
| --- |
| **def** backprop(X,y,model,z1,a1,z2,a2,z3,output,reg\_lambda):  delta3 = output  delta3[range(X.shape[0]), y] -= 1 *#yhat - y* dW3 = (a2.T).dot(delta3)  db3 = np.sum(delta3, axis=0, keepdims=**True**)  delta2 = delta3.dot(model[**'W3'**].T) \* relu\_derivative(a2) *#if ReLU* dW2 = np.dot(a1.T, delta2)  db2 = np.sum(delta2, axis=0)  *#delta2 = delta3.dot(model['W2'].T) \* (1 - np.power(a1, 2)) #if tanh* delta1 = delta2.dot(model[**'W2'**].T) \* relu\_derivative(a1) *#if ReLU* dW1 = np.dot(X.T, delta1)  db1 = np.sum(delta1, axis=0)  *# Add regularization terms* dW3 += reg\_lambda \* model[**'W3'**]  dW2 += reg\_lambda \* model[**'W2'**]  dW1 += reg\_lambda \* model[**'W1'**]  **return** dW1, dW2, dW3, db1, db2, db3 |

|  |
| --- |
| model[**'W1'**] -= learning\_rate \* dW1 model[**'b1'**] -= learning\_rate \* db1 model[**'W2'**] -= learning\_rate \* dW2 model[**'b2'**] -= learning\_rate \* db2 model[**'W3'**] -= learning\_rate \* dW3 model[**'b3'**] -= learning\_rate \* db3 |

**Code File**

**import** numpy **as** np  
**import** math  
**from** sklearn **import** datasets  
  
**def** relu(X):  
 **return** np.maximum(X, 0)  
  
**def** relu\_derivative(X):  
 **return** 1. \* (X > 0)  
  
**def** build\_model(X,hidden\_nodes,output\_dim=2):  
 model = {}  
 input\_dim = X.shape[1]  
 model[**'W1'**] = np.random.randn(input\_dim, hidden\_nodes) / np.sqrt(input\_dim)  
 model[**'b1'**] = np.zeros((1, hidden\_nodes))  
 model[**'W2'**] = np.random.randn(hidden\_nodes, hidden\_nodes) / np.sqrt(hidden\_nodes)  
 model[**'b2'**] = np.zeros((1, hidden\_nodes))  
 model[**'W3'**] = np.random.randn(hidden\_nodes, output\_dim) / np.sqrt(hidden\_nodes)  
 model[**'b3'**] = np.zeros((1, output\_dim))  
 **return** model  
  
**def** feed\_forward(model, x):  
 W1, b1, W2, b2, W3, b3 = model[**'W1'**], model[**'b1'**], model[**'W2'**], model[**'b2'**], model[**'W3'**], model[**'b3'**]  
 *# Forward propagation* z1 = x.dot(W1) + b1  
 *#a1 = np.tanh(z1)* a1 = relu(z1)  
 z2 = a1.dot(W2) + b2  
 a2 = relu(z2)  
 z3 = a2.dot(W3) + b3  
 exp\_scores = np.exp(z3)  
 out = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=**True**)  
 **return** z1, a1, z2, a2, z3, out  
  
**def** calculate\_loss(model,X,y,reg\_lambda):  
 num\_examples = X.shape[0]  
 W1, b1, W2, b2, W3, b3 = model[**'W1'**], model[**'b1'**], model[**'W2'**], model[**'b2'**], model[**'W3'**], model[**'b3'**]  
 *# Forward propagation to calculate our predictions* z1, a1, z2, a2, z3, out = feed\_forward(model, X)  
 probs = out / np.sum(out, axis=1, keepdims=**True**)  
 *# Calculating the loss* corect\_logprobs = -np.log(probs[range(num\_examples), y])  
 loss = np.sum(corect\_logprobs)  
 *# Add regulatization term to loss* loss += reg\_lambda/2 \* (np.sum(np.square(W1)) + np.sum(np.square(W2)) + np.sum(np.square(W3)))  
 **return** 1./num\_examples \* loss  
  
**def** backprop(X,y,model,z1,a1,z2,a2,z3,output,reg\_lambda):  
 delta3 = output  
 delta3[range(X.shape[0]), y] -= 1 *#yhat - y* dW3 = (a2.T).dot(delta3)  
 db3 = np.sum(delta3, axis=0, keepdims=**True**)  
 delta2 = delta3.dot(model[**'W3'**].T) \* relu\_derivative(a2) *#if ReLU* dW2 = np.dot(a1.T, delta2)  
 db2 = np.sum(delta2, axis=0)  
 *#delta2 = delta3.dot(model['W2'].T) \* (1 - np.power(a1, 2)) #if tanh* delta1 = delta2.dot(model[**'W2'**].T) \* relu\_derivative(a1) *#if ReLU* dW1 = np.dot(X.T, delta1)  
 db1 = np.sum(delta1, axis=0)  
 *# Add regularization terms* dW3 += reg\_lambda \* model[**'W3'**]  
 dW2 += reg\_lambda \* model[**'W2'**]  
 dW1 += reg\_lambda \* model[**'W1'**]  
 **return** dW1, dW2, dW3, db1, db2, db3  
  
  
**def** train(model, X, y, num\_passes=1000, reg\_lambda = .1, learning\_rate=0.1):  
 *# Batch gradient descent* done = **False** previous\_loss = float(**'inf'**)  
 i = 0  
 losses = []  
 **while** done == **False**:  
 *#while i < 1300:  
 #feed forward* z1,a1,z2,a2,z3,output = feed\_forward(model, X)  
 *#backpropagation* dW1, dW2, dW3, db1, db2, db3 = backprop(X,y,model,z1,a1,z2,a2,z3,output,reg\_lambda)  
 *#update weights and biases* model[**'W1'**] -= learning\_rate \* dW1  
 model[**'b1'**] -= learning\_rate \* db1  
 model[**'W2'**] -= learning\_rate \* dW2  
 model[**'b2'**] -= learning\_rate \* db2  
 model[**'W3'**] -= learning\_rate \* dW3  
 model[**'b3'**] -= learning\_rate \* db3  
 **if** i % 1000 == 0:  
 loss = calculate\_loss(model, X, y, reg\_lambda)  
 losses.append(loss)  
 print (**"Loss after iteration %i: %f"** %(i, loss))  
 **if** (previous\_loss-loss)/previous\_loss < 0.01:  
 done = **True** *#print i* previous\_loss = loss  
 i += 1  
 **return** model, losses  
  
**def** main():  
 X, y = datasets.make\_moons(16, noise=0.10)  
 num\_examples = len(X) *# training set size* nn\_input\_dim = 2 *# input layer dimensionality* nn\_output\_dim = 2 *# output layer dimensionality* learning\_rate = 0.01 *# learning rate for gradient descent* reg\_lambda = 0.01 *# regularization strength* model = build\_model(X,20,2)  
 model, losses = train(model,X, y, reg\_lambda=reg\_lambda,learning\_rate=learning\_rate)  
 output = feed\_forward(model, X)  
 preds = np.argmax(output[3], axis=1)  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()