HW7

Name: Swarna Prabhu

UIN: 932003046

Theory :

Question 1:

The decision boundary implemented by a neural network with n no of inputs, a single output neuron and no hidden layers will consist of a linear function which is of the form x0w0 + x1w1……xnwn + b =0 which will segregate the training data into 2 classes. Each input feature will be multiplied by corresponding weight to give an intermediate output y0 which is fed to an activation function. If that output is greater than a specific value, the final output from neuron is set to 1 else it is set to zero. Therefore, the single output from neuron will classify the data into 2 classes just like a perceptron and hence the decision boundary is a straight linear function separating the 2 classes.

Computer Assignment

Question 2:

Plot for the Gaussian with 200 sample points, the points marked in red is centered at (1,1) and the points marked in blue is centered at 3,2

The top figure shows the distribution for the narrow data where there is very less overlap between the 2 distributions. The standard deviation taken for the same is 0.4 and plot is plotted using the np.random.normal which is used to get normal distribution in python.

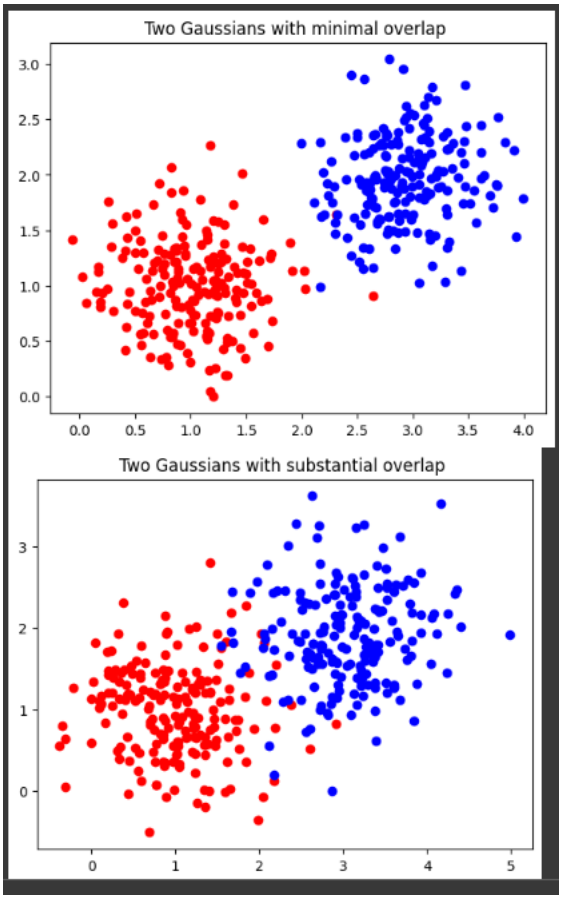

Fig 1 : Plot for narrow and wide data for the Gaussian distribution

The above experiment is repeated with a larger standard deviation of value 0.7 in order to create overlap between the 2 data as shown above. 4 The plot is plotted using the np.random.normal function which is used to get normal distribution in python as above.

Code for creating the 2 data sets which are Gaussians and 2D in space is given below:

```
##importing the necessary packages from matplot lib and pyplot to get the final plots
import matplotlib.pyplot as plt
import numpy as np
from numpy.random import seed
from numpy.random import normal

##to ensure that random data generated is constant
seed(2)
```

```
#create the first gaussian normal distribution with mean at 1,1 standard deviation 0.4 and data s
ize 200
datax = np.random.normal(loc=1, scale = 0.4, size=200)
datay = np.random.normal(loc =1,scale = 0.4, size =200)

#get the data plotted

plt.scatter(datax, datay, c = 'r')

#create the 2nd gaussian normal distribution centred at (3,2) and standard deviation 0.4 data siz
e 200

datax1 = np.random.normal(loc = 3, scale = 0.4, size =200)
datax2 = np.random.normal(loc = 2, scale = 0.4, size = 200)
plt.scatter(datax1, datax2, c = 'b')



plt.title('Two Gaussians with minimal overlap ')

plt.show()

#################################################################
##implementing the gaussian distribution with substantial overlap

#create the first gaussian normal distribution with mean at 1,1 standard deviation 0.6 and data s
ize 200
dataz = np.random.normal(loc=1, scale = 0.6, size=200)   ##previous 1.0 , then 0.7
dataw = np.random.normal(loc =1,scale = 0.6, size =200)

#get the data plotted

plt.scatter(dataz, dataw, c = 'r')

#create the 2nd gaussian normal distribution with mean at 3,2 and deviation 0.6 little higher for
 substantial overlap

dataz1 = np.random.normal(loc = 3, scale = 0.6, size =200)
dataw2 = np.random.normal(loc = 2, scale = 0.6, size = 200)
plt.scatter(dataz1, dataw2, c = 'b')

plt.title('Two Gaussians with substantial overlap ')

plt.show()
```

Question 3:

Training for the perceptron with Gradient Descent technique

Steps used in training the perceptron is as below:

1>The dataset from question 2 above has been used for the implementation for the perceptron with gradient decent optimizer.

In this we consider the less overlap data or narrow data

2>Prepare the data so that each dimension of the gaussian distribution will correspond to one particular input feature vector. Hence after combining the 2 datasets we get an X dataset (400 * 3) and Y data – label (400 * 1).

The third column in X will be initialized to one to update the bias data and the first 2 columns correspond to X and Y axis data.

```
##steps to generate the training data x, using random generator for the same
#in this step the data generated in the first step is converted to represent input features and b
ias which is used for training the perceptron
import matplotlib.pyplot as plt
import numpy as np
import random
sample_size = 200
#convert the orientation or reshape the array so that each
#datax and datay is a 200*1 column matrix
datax = datax.reshape(-1,1)
datay = datay.reshape(-1,1)
#combine the 2 column array into single with 2 columns to represent 2 feautres for data ceterd at
 1,1
X_combined1 = np.hstack((datax, datay))
#stores the output data with label zero
numy1 = np.zeros(200)
numy1 = numy1.reshape(-1,1)
#stpres the data output with label one
numy2 = np.ones(200)
numy2 = numy2.reshape(-1,1)

#combine the 2 column array into single with 2 columns to represent 2 feautres for the data cente
red at 3,2
datax1 = datax1.reshape(-1,1)
datax2 = datax2.reshape(-1,1)

X_combined2 = np.hstack((datax1, datax2))
```

```
##concatenate to represent the final X data with 400 rows and 3 columns

X = np.concatenate((X_combined1 , X_combined2))
#thrid column added as 1 to represent the bias
zero_arr = np.ones((400,1))
X = np.hstack((X, zero_arr))
Y = np.concatenate((numy1,numy2))
print(X.shape)
print(Y.shape)
#plt.hist(nums2)
plt.show()
```

Output

```
(400, 3)
(400, 1)
```

Step 3: Prepare the training and test data set by splitting them using test train split function.

```
##before training split the available data to training set and testing set  using test train spli
t from sklearn package
from sklearn.model_selection import train_test_split
X_train, X_test,  Y_train, Y_test = train_test_split(X,Y,test_size=0.2, random_state = 10)
#always a good way to split the training to test data is in the ration 8:1
##get the dimensions of the training x and y data
print(Y_train.shape)
print(X_train.shape)
print(X_test.shape)
print(Y_test.shape)
```

Output of the dimension of the training and testing data is given below:

Always a ratio of 80:20 between training and testing data is desirable to get the network trained correctly.

```
(320, 1)
(320, 3)
(80, 3)
(80, 1)
```

Step 4: Training of the perceptron using the gradient decent optimizer:

In the train function definition , there are two for loops used to optimize the weights, 1st iteration is for each iteration of the epoch and 2nd for loop for every row of the training data of size (319 * 3) .Since at every incremental step we gradually calculate the dot product of weights and input feature vector and add the bias and then compare this output to generate final prediction as 0 or 1 , this optimization is known as gradient decent optimizer.

```
implement a perceptron and a gradient decent optimizer to train wieghts w1 w2 and w3 for narrow d
ata--minimal overlap
```

```
##defines the class Perceptron that has multiple definitions for each task - training , predictin
g and checking the accuracy or correctness
class Percept():
###intialization of the class by passing the self and learning rate
    def __init__(self, learning_rate ):
        self.learn = learning_rate
        self.bias = None
        self.weights = None
 #bias and weight is kept as none initially

#here the weughts and bias are trained based om the epoch passed
    def train(self, X_trn, Y_trn, epoch):


      #initialize the weights to 3*1 array

        self.bias = 0
        self.weights = np.zeros(3)
        #training and learning of the weights
        #weight array to store the weigjts upon each iteration of the training process
        weight_arr = np.zeros((epoch, 3))

        for i in range(epoch):   ##different iterations
          #for every iteration we loop over evry row of the training data in this case with 319 r
ows
            for j in range(319):
            #for every row the fout is calculated as x0w0 + x1w1 + x2w3 = fout
                f_out = np.dot(X_trn[j, :], self.weights) + self.bias

                #if fout is greater than zero keep final output as 1 else 0
                if f_out > 0:

                  y_out = 1
                else:
                  y_out = 0



                #updating the weight and bias based on diff in the actual data and prediction bas
ed on weights
                diff = Y_trn[j] - y_out
                #find the delta to calculate the updated weights and bias
                delta = self.learn*diff*X_trn[j]
                self.weights = self.weights + delta
                #print(self.weights[0])
                self.bias = self.bias + self.learn*diff
```

```
                #after updating for each row at the end of every iteration the final weigjts and
bias is updated to the array
            w1 = self.weights[0]
            w2 = self.weights[1]
            w3 = self.weights[2]
            weight_arr[i, 0] =w1
            weight_arr[i, 1] =w2
            weight_arr[i, 2] =w3
            # print(weight_arr)

        #return the weight and bias for every iteration after training
        return weight_arr, self.bias


#defines the prediction function where the for a given training data we find the prediction
    def prediction(self, X_trn):
        #prediction is done based on caluclating the xowo + x1w1 + b = fout
        final_out =np.dot(X_trn, self.weights) + self.bias
        #based on fout get a final output prediction of 0 or1
        y_predn = np.where(final_out > 0. , 1, 0)

        #if final_out > 0:
         #         y_pred = 1
        #else:
         #         y_pred = 0
        #print(final_out)
        y_pred = y_predn.reshape((80,1))
        return y_pred

    def accuracy(self,y_actual, y_ours):


        #calulcates the length of the ypredicted to get the denominator for accuracy
        val = len(y_ours)
        # for i in range(val):
        #   err = (y_actual[i] - y_ours[i])**2
        # mean_square = err/val
        #print(mean_square)
        #wherever the prediction and the actual output label data matches that is considered as a
ccurate
        bool_val = y_actual == y_ours
        #print(bool_val)
        ##accruacry percentage is total correct values / total values
        num = np.sum(bool_val)
```

```
        acc = num / val
        #returns the accuracy
        return acc
```

Step 5: Testing for the implementation of the perceptron network using a low learning rate on narrow data and 100 iterations. We see that an accuracy of 97.5 has been obtained which shows the correctness of the implementation and training of the weights.

```
##Train the model, to show variation of the model with alpha loweer learning rate 0.000006 -----
>minimal overlap-> narrow data

obj = Percept(0.000006)
epoch = 100
#defiens the epoch
#initialize the final weigt array in numpy
final_weight =  np.zeros((epoch, 3))
#for i in range(len(epoch)):
#for i in range(epoch):
  #final_weight[i, :], bi= obj.train(X_train, Y_train, epoch)
#calls the train function by passing training data and epoch
final_weight, bi= obj.train(X_train, Y_train, epoch)
yp = obj.prediction(X_test)
#gets the a=prediction and accuracy on test data
result = obj.accuracy(Y_test, yp)
result = result*100

#print the bias and the accuracy on the console
#print(final_weight) ## 7 *3
print(bi)
#print(yp)
print('Accuracy on testing data is for lower learning rate %.2f%%', result)
```

```
[-0.000108]
Accuracy on testing data is for lower learning rate %.2f%% 97.5
```

Question 4:

Case 1: Plot values for ω1, ω2, and ω3 over the training iterations for two values of α that show different behavior, for the narrow data

When learning rate alpha is smaller-→ learning rate = 0.000006

```
Accuracy on testing data is for lower learning rate %.2f%% 100.0%
```

Variation of weights w1 w2 and w3 with training iterations with a lower learning rate = 0.000006:

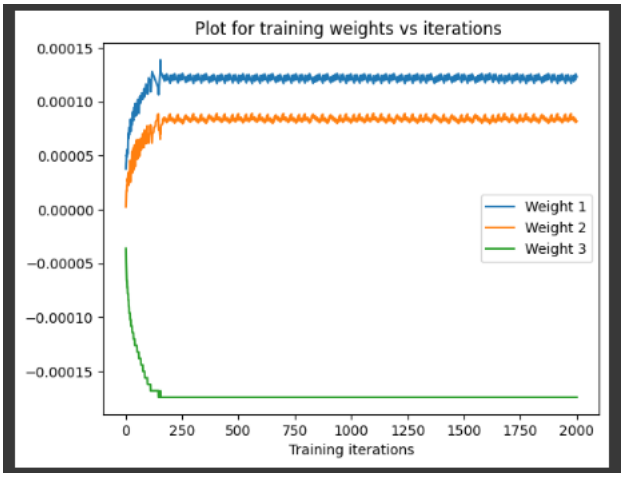

Fig 2 Weights vs iterations for alpha0.000006 – narrow data

This plot was plotted by storing the weights w1, w2 and bias after each iteration for a low learning rate for the narrow data. It is observed that until 250 iterations the weights fluctuate and later, they stabilize as they reach convergence for larger iterations. Since the learning rate is very less, it takes small steps and hence we see less oscillations.

Variation of weights w1, w2 and w3 wrt to higher learning rate like 0.2 for narrow data

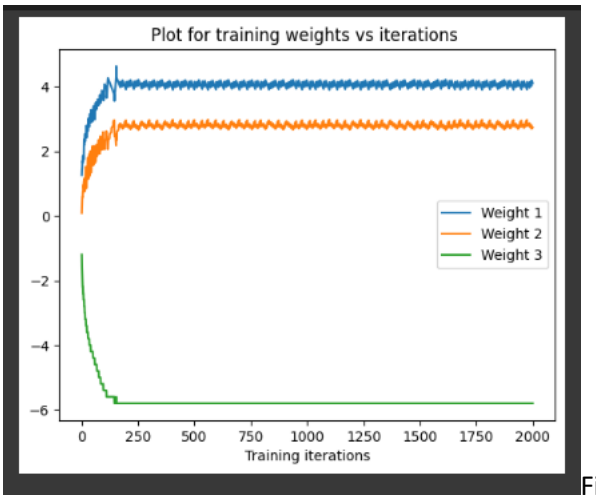

Fig 3 Weights vs iterations for alpha 0.2 – narrow data

The graph above gives the plot for weights w1, w2 and bias w3 when plotted for high learning rate 0.2 for the same number of iterations. Weights w1 ,w2 and w3 are stored after each iteration for a higher learning rate for the narrow data and we see that the value of the weights that are updated (y axis data)is much higher compared to the value of the weights in Fig 1. This is because with high LR , update to weights are also higher but the convergence is similar to that in Fig1.

This can be expected as the data is already well classified where the overlap is minimal , hence we can expect less number of misclassifications and similar convergence

Case 2: Plot values for ω1, ω2, and ω3 over the training iterations for two values of α that show different behavior, for the wider data

When learning rate is higher = 0.2

```
[-0.12]
Accuracy on training %.2f%% 92.5
```

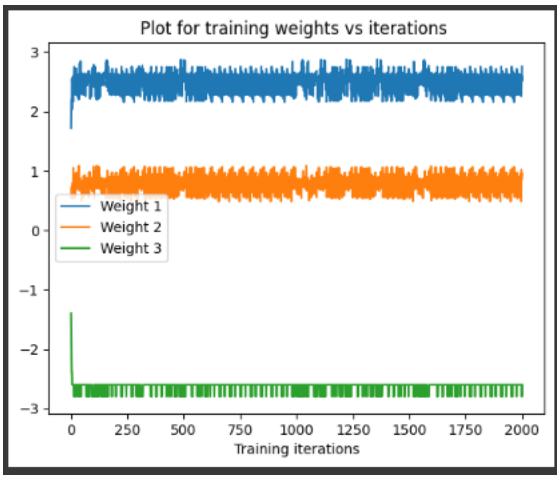

Fig 4 Weights vs iterations for alpha 0.2 – Wide data

This plot was plotted by storing the weights w1, w2 and bias after each iteration for a higher learning rate for the wider data. Here for the same number of iterations convergence is not great since for wider data where there is more misclassifications, the weights tend to oscillate much higher than it did for the narrow data.

```
When learning  rate is lower as  0.000006
```

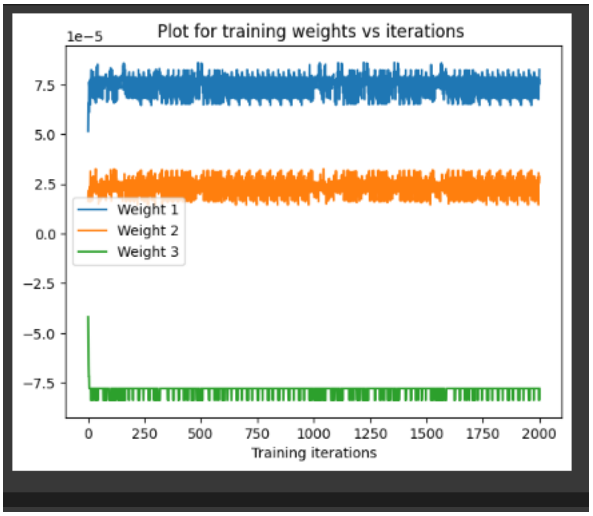

Fig 5 Weights vs iterations for alpha 0.000006 – Wide data

This plot was plotted by storing the weights w1, w2 and bias after each iteration for a low learning rate for the wider data. Here for the same number of iterations convergence is not great since for wider data where there is more misclassifications, the weights tend to show lesser number of oscillations for low learning rate than compared to the high learning rate but the difference has been not significant.

Case 3: Choose one of the α's and plot the data and final decision boundary for (a) the narrow data and for (b) the wide data.

Alpha chosen is 0.2

Decision boundary plot for narrow data and alpha 0.2

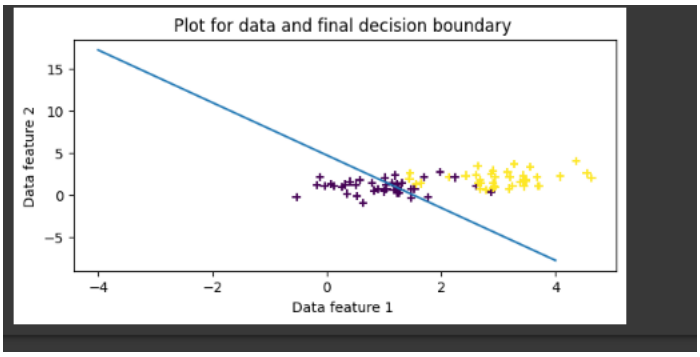

Fig 5: represents the hyperplane decision boundary.

A straight line is fitted by taking the weights w1 and w2 and bias and plug these values obtained after running the Perceptron class in the equation x1w1 + x2w2 + w3 = 0 where w3 = bias

In the code for plot , x1_max and x1_min corresponds to the y axis (vertical axis) and x0_max and x0_min correspond to x axis (horizontal axis).We convert the above equation to form y = mx + c where

slope of the line is m = weights1/weights2 and intercept given by w3 /weight2.The weights are fetched after training the model for narrow data with learning rate 0.2

Decision boundary plot for wide data and alpha 0.2

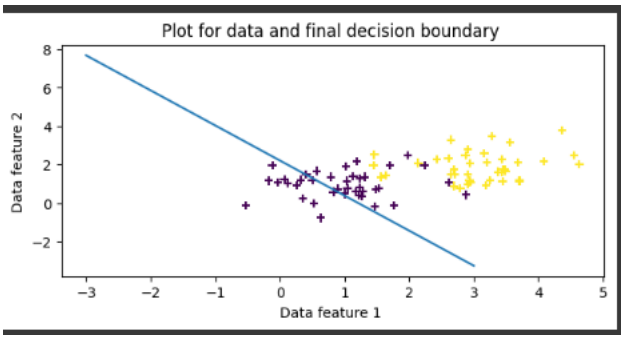

Fig 5: represents the hyperplane decision boundary – wide data.

A straight line is fitted by taking the weights w1 and w2 and bias and plug these values obtained after running the Perceptron class in the equation x1w1 + x2w2 + w3 = 0 where w3 = bias

In the code for plot , x1_max and x1_min corresponds to the y axis (vertical axis) and x0_max and x0_min correspond to x axis (horizontal axis).We convert the above equation to form y = mx + c where slope of the line is m = weights1/weights2 and intercept given by w3 /weight2.The weights are fetched after training the model for wider data with learning rate 0.2

Question 5: Discuss your choice of α, the time until convergence and any other behavior you observed during the optimization.

Choice of α: Choice of alpha would be the one with lower learning rate (0.000006) as the weights vary slowly while training and achieves convergence. In the analysis, it is observed that there was not any significant difference in the learning rate with low or high for convergence based on the distribution since the narrow data and the wide data both represented extreme cases where there could be either over fitting or under fitting due to the learning rate . This is because when we chose high learning rate weight update takes large steps so overfitting can occur in small learning rate weights increment slowly so underfitting can occur. This is also the reason why we see spikes in the initial iterations of weights w 1, w2 and w3 in narrow data case. Also, the oscillations are relatively more in wide data case since high overlap of the points make it difficult to fit the boundary decision.

This is also the reason the decision boundary in both cases for wide data and narrow data is not very ideal in terms of classifying the test data into 2 features.

By comparing the accuracy for the narrow data and wider data the dataset that is well classified in the case of narrow data (less overlap), learns faster and hence shows quicker convergence as the accuracy for 2000 iterations was 100 % on the test data.

On the other hand, we observed that the accuracy for the same 2000 iterations was 92.5 % since there is a large substantial overlap in the case of wider data. Therefore, we observe that the updating of weights produced more oscillations (from plot), and it takes more time to convergence since at the end of 2000 iterations it doesn't show smooth convergence as it did for wider data.