UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NEURAL NETWORK

ASSIGNMENT#1

MOHAMMAD HEYDARI

810197494

UNDER SUPERVISION OF:

DR. AHMAD KALHOR

ASSISTANT PROFESSOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

Mar. 2022

1 CONTENTS

2 QUESTION #1

2.1 MCCULLOCH PITTS

In this part we intend to implement a Full adder using Mcculloch Pitts neuron

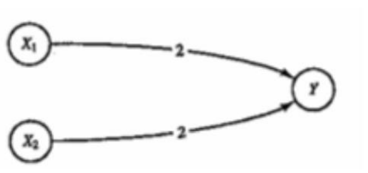
According to question description we have four neurons as of our inputs and also three neurons as of our outputs, all have binary representation.

First of all, we should understand how a full adder works and how we can use basic logical gates to construct a full adder block in such a way which produce two digits for binary summation results and also a bit for carry-out in our case.

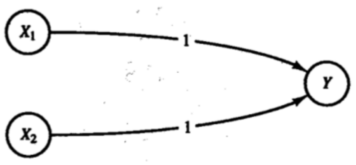
We have used four basic logical gates, XOR, AND, OR , if we could represent these three gates based on Mcculloch Pitts neuron we can say half of steps have been successfully done, so in this section we want to make these gates with their appropriate weights which tend to produce an adequate and exact result.

So, for more detailed information please have an accurate look on construction of these gates:

OR:



AND:



XOR:

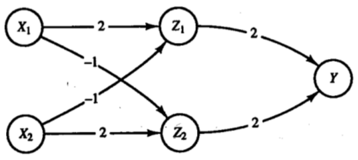
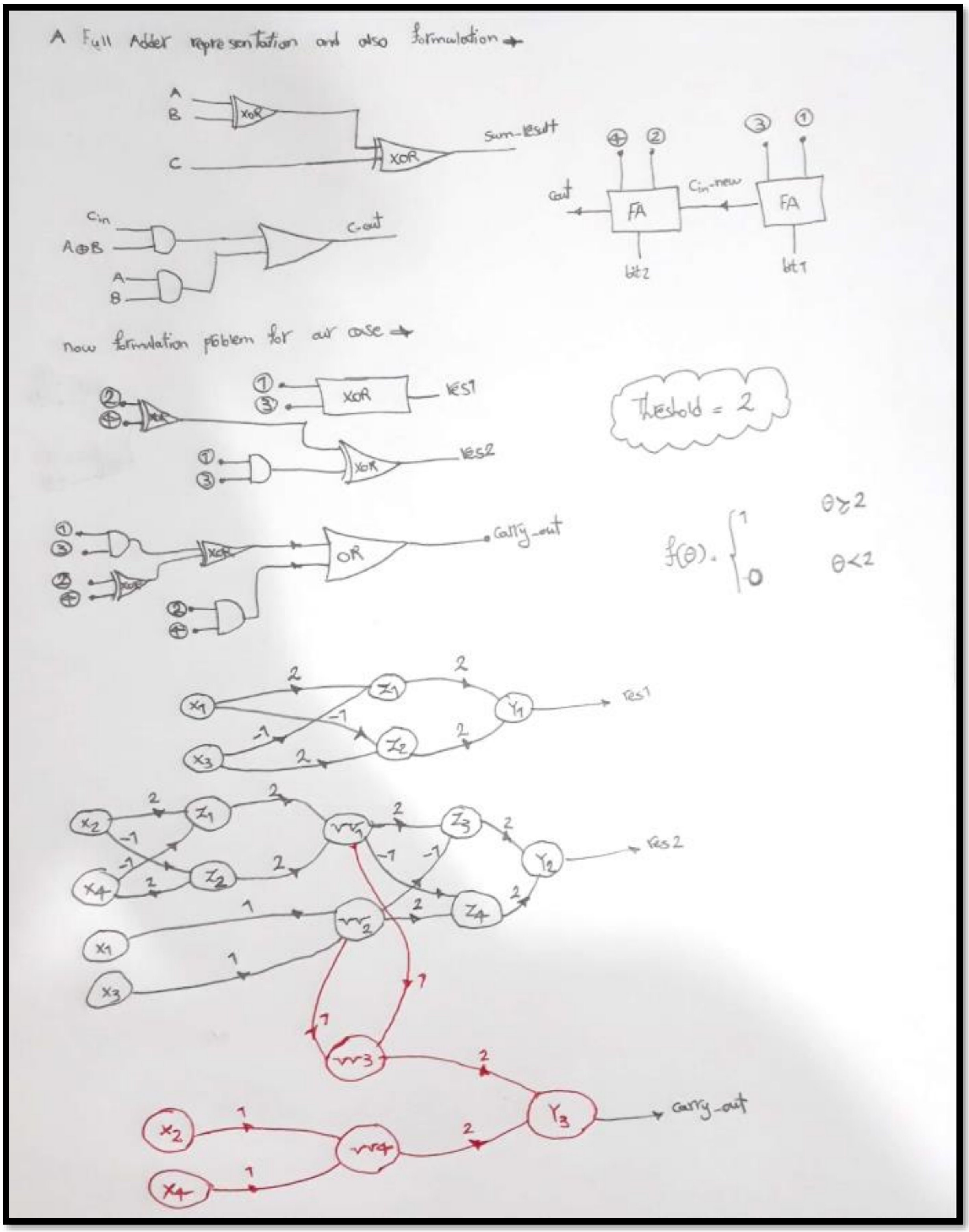


Figure 1: Architecture of Logical gates

Please note that the Threshold for activation function has been adjusted to 2 in our case.

Further, you can find whole theoretical approach which have been supported by above notation and will satisfy all the question requirements afterwards we are going to simulate that using Python environment.

If there is any issue with that my solution, please let me know to provide more detailed information. (Be in touch **here!**)


A Full Adder representation and also formulation →
A
B
XOR
C
XOR
Sum-result
Cin
A⊕B
A
B
C-out
④ ② ③ ①
out
FA
Cin-new
FA
bit2
bit1
now formulation problem for our case →
① ③ XOR res1
② ④ XOR
① ③ XOR res2
Threshold = 2
① ③ ② ④ XOR XOR OR carry-out ② ④
f(θ) = 1 θ≥2, 0 θ<2
x1 x3 z1 z2 Y1 res1 2 -1 -1 2 2 2
x2 x4 z1 z2 w1 z3 z4 Y2 res2 2 -1 -1 2 2 2 2 -1 -1 2 2 2
x1 x3 w2 1 1
w3 1 1 2
x2 x4 w4 1 1 2
Y3 carry-out

Now, we have simulated all in python environment in as exact as it could be.

Further you can see the results of simulation for all of possible states and as we know there are sixteen states to simulate :

```
0 0 + 0 0 = 0 0 0
0 1 + 0 0 = 0 0 1
1 0 + 0 0 = 0 1 0
1 1 + 0 0 = 0 1 1
0 0 + 0 1 = 0 0 1
0 1 + 0 1 = 0 1 0
1 0 + 0 1 = 0 1 1
1 1 + 0 1 = 1 0 0
0 0 + 1 0 = 0 1 0
0 1 + 1 0 = 0 1 1
1 0 + 1 0 = 1 0 0
1 1 + 1 0 = 1 0 1
0 0 + 1 1 = 0 1 1
0 1 + 1 1 = 1 0 0
1 0 + 1 1 = 1 0 1
1 1 + 1 1 = 1 1 0
```

Figure 2: simulate result

As you can see all results matches with our previous knowledge as well as it could be.

Further I have attached all of my implementation in python which generate above result:

Implementing Full-Adder Using Mcculloch Pitts

```
import numpy as np
import matplotlib.pyplot as plt
```

```
def Mcculloch_Pitts(X,w):
    res=np.dot(w,X)
    return res
```

```
def activation_function(z):
    if z >= 2 :
        res=1
    else:
        res=0
    return res
```

XOR gate which produce first digit

```
def Digit_one(X1,w1,w2,w3):
    z1= Mcculloch_Pitts(X1,w1)
    z1=activation_function(z1)
    z2= Mcculloch_Pitts(X1,w2)
    z2=activation_function(z2)
    X2=[z1,z2]
    y1=Mcculloch_Pitts(X2,w3)
    out1=activation_function(y1)
    return out1
```

Complete Block which produce second digit

```
def Digit_two(X1,X2,w4,w5,w6,w7,w8,w9,w10):

    ## XOR
    z3= Mcculloch_Pitts(X2,w4)
    z3=activation_function(z3)
    z4= Mcculloch_Pitts(X2,w5)
    z4=activation_function(z4)
    X3=[z3,z4]
    y2=Mcculloch_Pitts(X3,w6)
    vv1=activation_function(y2)

    ## AND
    z5= Mcculloch_Pitts(X1,w7)
    vv2=activation_function(z5)

    ## XOR
    X4=[vv1,vv2]
    z6= Mcculloch_Pitts(X4,w8)
    z6=activation_function(z6)
    z7= Mcculloch_Pitts(X4,w9)
    z7=activation_function(z7)
    X5=[z6,z7]
    y3=Mcculloch_Pitts(X5,w10)
    out2=activation_function(y3)

    return out2,vv1,vv2
```

Making Carry-out Digit

```
def Carry_out(X2,X3,w11,w12):

    ## AND
    z8= Mcculloch_Pitts(X3,w11)
    vv3=activation_function(z8)

    ## AND
    z9= Mcculloch_Pitts(X2,w12)
    vv4=activation_function(z9)

    ## OR
    X4=[vv3,vv4]
    z10= Mcculloch_Pitts(X4,w13)
    Cout=activation_function(z10)

    return Cout
```

FULL ADDER

```
def FA(X1,X2,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13):
    out1=Digit_one(X1,w1,w2,w3)
    out2,vv1,vv2=Digit_two(X1,X2,w4,w5,w6,w7,w8,w9,w10)
    X3=[vv1,vv2]
    Cout=Carry_out(X2,X3,w11,w12)
    return Cout,out2,out1
```

```
w1=[2,-1]
w2=[-1,2]
w3=[2,2]
w4=[2,-1]
w5=[-1,2]
w6=[2,2]
w7=[1,1]
w8=[2,-1]
w9=[-1,2]
w10=[2,2]
w11=[1,1]
w12=[1,1]
w13=[2,2]
```

```
inputs=np.zeros((16,4))
inputs[0,:]=[0,0,0,0]
inputs[1,:]=[0,0,0,1]
inputs[2,:]=[0,0,1,0]
inputs[3,:]=[0,0,1,1]
inputs[4,:]=[0,1,0,0]
inputs[5,:]=[0,1,0,1]
inputs[6,:]=[0,1,1,0]
inputs[7,:]=[0,1,1,1]
inputs[8,:]=[1,0,0,0]
inputs[9,:]=[1,0,0,1]
inputs[10,:]=[1,0,1,0]
inputs[11,:]=[1,0,1,1]
inputs[12,:]=[1,1,0,0]
inputs[13,:]=[1,1,0,1]
inputs[14,:]=[1,1,1,0]
inputs[15,:]=[1,1,1,1]
inputs=inputs.astype(int)
```

```
for i in range(len(inputs)):
    X1=[inputs[i,3],inputs[i,1]]
    X2=[inputs[i,2],inputs[i,0]]
    cout,out2,out1=FA(X1,X2,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13)
    res=[cout,out2,out1]
    in1=[inputs[i,2],inputs[i,3]]
    in2=[inputs[i,0],inputs[i,1]]
    print(*in1,"+",*in2,"=",*res)
```

```
0 0 + 0 0 = 0 0 0
0 1 + 0 0 = 0 0 1
1 0 + 0 0 = 0 1 0
1 1 + 0 0 = 0 1 1
0 0 + 0 1 = 0 0 1
0 1 + 0 1 = 0 1 0
1 0 + 0 1 = 0 1 1
1 1 + 0 1 = 1 0 0
0 0 + 1 0 = 0 1 0
0 1 + 1 0 = 0 1 1
1 0 + 1 0 = 1 0 0
1 1 + 1 0 = 1 0 1
0 0 + 1 1 = 0 1 1
0 1 + 1 1 = 1 0 0
1 0 + 1 1 = 1 0 1
1 1 + 1 1 = 1 1 0
```

For running this part, please have an accurate look on related directory and There you can easily find all of you need and also, I would appreciate it if you would consider them ☺

3 Question #2

3.1 Adaline

In this part we intend to implement a classifier using Adaline concept which let us to find the line which tend to separate our binary classes as well as it could be.

Note that all of implementation in this part has been done using basic python libraries such as NumPy and matplotlib for plotting the results so we can say it has been done from scratch!

First of all, we must create the classes according to question description, as it said we have two separated classes which have -1 & 1 as their labels.

3.1.1 Dispersion Representation of Dataset

Let's create the classes, the features of each class have been supported by normal distribution with specific mean and standard deviation which clearly mentioned for each classes.

1. Class 1

in first category we have two features which have normal distribution **the first one with mean = 0 and std =0.3 presented in a 100-points vector called x and the second one with mean = 2 and std=0.3 presented in 100-points vector called y**

2. Class 2

in first category we have two features which have normal distribution **the first one with mean = 2 and std =0.3 presented in a 100-points vector called x and the second one with mean = 2 and std=0.3 presented in 100-points vector called y.**

Then we have plotted these two classes by their distribution in such a way that the vertical axis is belong to y variable and the horizontal axis is belong to x variable.

Further you can see more detailed result of this section:

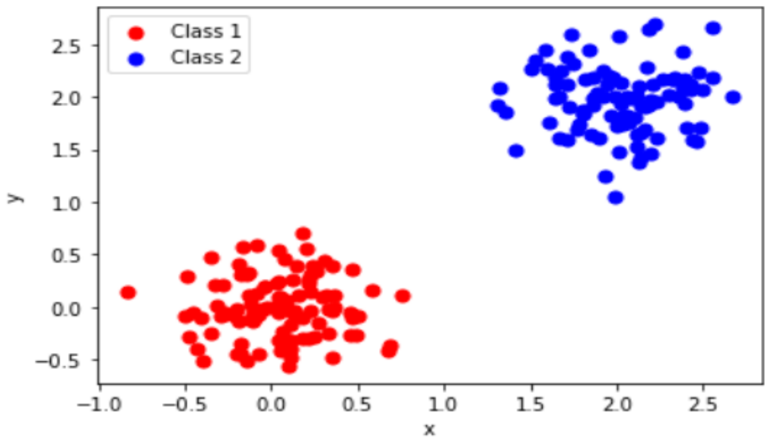


Figure 3: Representation of Data Dispersion

as you can see the result matches as well as we expected, and in whole of our further process the blue class will present the first class **with label = 1** and the red class will present the second class **with label=-1.**

3.1.2 Training our Model based on Adaline approach

In further process we tend to train a model based on Adaline approach to separate there two classes so the first thing we should take into account is about making dataset in such a way that mentioned classes will introduce with -1 & 1 label which will show a specific data-point tend to be member of red class or maybe blue class.

So, in this regards I have provided piece of code which put pair data-points in rows of our dataset with an extra column which represent their target values.

Now let's find out how we can train a model **using a single neuron** that all have been abbreviated in **Adaline approach.**

Adaptive Linear Unit (Adaline):

An ADALINE is a single unit (neuron) that receives input from several units. It also receives input from a "unit" whose signal is always + 1, in order for the bias weight to be trained by the same process (the delta rule) as is used to train the other weights. A single ADALINE is shown below:

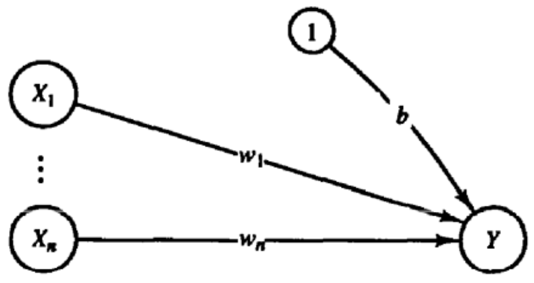
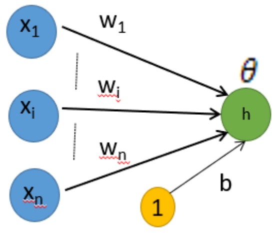


Figure 4: Architecture of an Adaline

Adaline is a type of one-layer network which is shown below:



(1)

$$\text{net} = \sum_{i=1}^{n} w_i x_i + b$$

$$h = f(net) = \begin{cases} +1 & \text{net } \geq 0 \\ -1 & \text{net } < 0 \end{cases}$$

After above contextualizing now let's review the all process of model training in case of further steps:

A training algorithm for an ADALINE is as follows:

Step 0: Initialize weights. (Small random values are usually used.) && Set learning rate a.

Step 1: While stopping condition is false, do Steps 2-6.

Step 2: For each bipolar training pair s: t, do Steps 3-5.

Step 3: Set activations of input units, i = 1, . . . , n: xi= si.

Step 4: Compute net input to output unit:

$$y_{in} = b + \sum_i x_i w_i$$

Step 5: Update bias and weights, i = 1, . . . , n:

Step 6: Test for stopping condition: If the largest weight change that occurred in Step 2 is smaller than a specified tolerance, then stop; otherwise continue.

I have written pieces of codes which support all from above notation and let us to have an adequate and exact training process, further you can see the result of classification after training model with learning rate=0.01 & epochs=30.

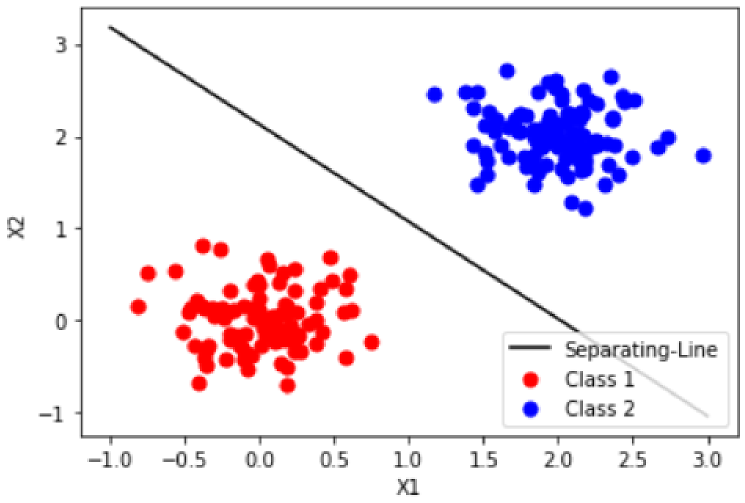


Figure 5: Classification using separating Line for first Sets

As our dataset was linearly separatable so we expect that we could separate our classes well using single neuron and with Adaline concept.

Then, as you can see in the above figure the classes have been divided very good and in other word, we predicted the classes with 100% accuracy and all separate using our algorithm.

Further we want to examine whether our approach has been done Good or not so we have plotted the Cost-function among epochs.

Note that the cost function inspired by Mean Square Error and calculated by summation of errors between target and predicted value and we expect this equation which represent the cost becomes converge during many epochs.

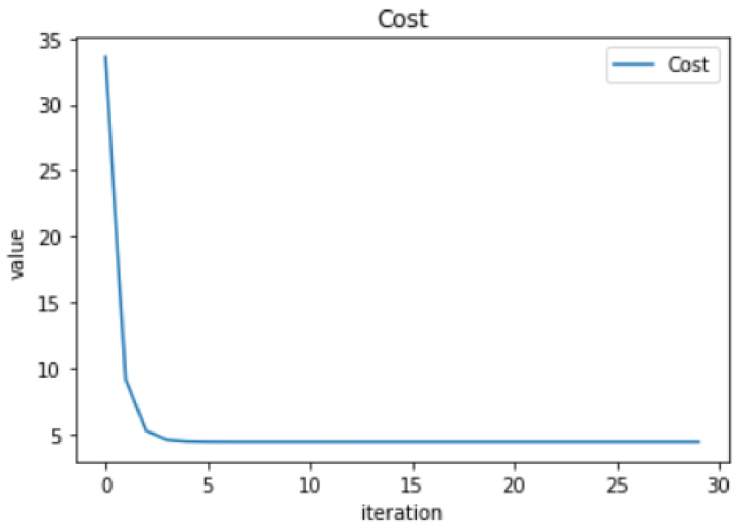


Figure 6: Cost Function for first Sets

As we can see the algorithm has been met the convergence well with just after few iterations and that's so nice and expectable because we have done all steps according to algorithm.

Furthermore, the figure has been converged smoothly which prove the fact that we have implemented algorithm true.

3.1.3 Repeating All Above Steps for the New Classes

Now, we have repeated all steps for the new sets.

further you can see the result of classification after training model with learning rate=0.001 & epochs=1000.

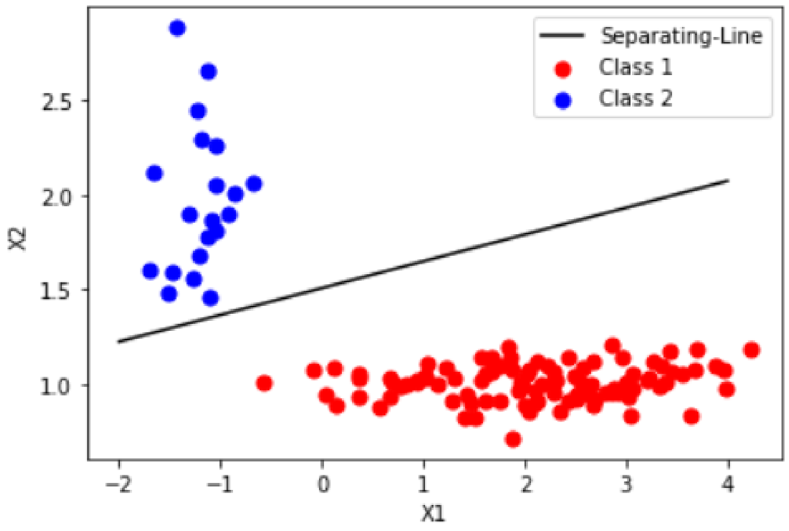


Figure 7: Classification using separating Line for the second Sets

As our dataset was linearly separatable so we expect that we could separate our classes well using single neuron and with Adaline concept.

Then, as you can see in the above figure the classes have been divided very good and in other word, we predicted the classes with 100% accuracy and all separate using our algorithm.

Please note that the line depends on initial weights so it may be different when we are dealing with different parameters so I have used seed command.

Further we want to examine whether our approach has been done Good or not so we have plotted the Cost-function among epochs.

Note that the cost function inspired by Mean Square Error and calculated by summation of errors between target and predicted value and we expect this equation which represent the cost becomes **converge** during many epochs.

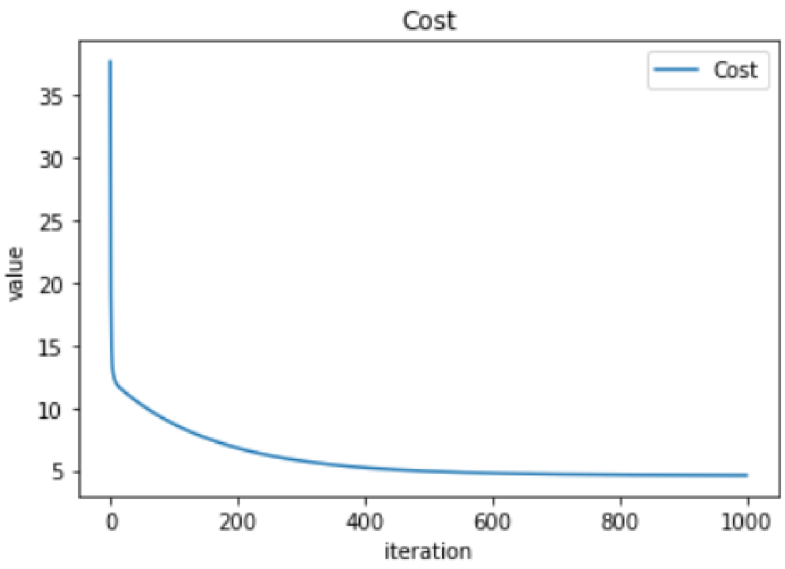


Figure 8: Cost Function for the second Sets

As we can see the algorithm has been met the convergence well with just after few iterations and that’s so nice and expectable because we have done all steps according to algorithm.

Furthermore, the figure has been converged smoothly which prove the fact that we have implemented algorithm true.

Further more detailed information has been attached:

Adaline Algorithm

```
import numpy as np
import matplotlib.pyplot as plt
```

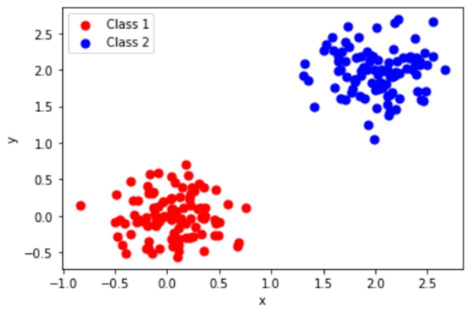
```
# set one
x1 = np.random.normal(0,0.3,100)
y1 = np.random.normal(0,0.3,100)

# set two
x2 = np.random.normal(2,0.3,100)
y2 = np.random.normal(2,0.3,100)
```

Section A: Plotting the Distribution of Data-point

```
plt.scatter(x1,y1,c='r',s=50)
plt.scatter(x2,y2,c='b',s=50)
plt.legend(["Class 1", "Class 2"])
plt.xlabel('x')
plt.ylabel('y')
```

```
Text(0, 0.5, 'y')
```



```
def activation_function(z):
    res=z;
    return res
```

```
def net_function(X,w,bias):
    res=np.dot(w,X)+bias;
    return res
```

```
def init_weights():
    random_gen1 = np.random.RandomState(1)
    weights = random_gen1.normal(loc = 0.0, scale = 0.01, size = (2))
    random_gen2 = np.random.RandomState(2)
    bias     = random_gen2.normal(loc = 0.0, scale = 0.01, size = 1)
    return weights,bias
```

```
def Adaline_training(X_train,w,bias,y_train,learning_rate,iteration):
    costs=[]
    for epoch in range(iteration):
        cost=0
        for i in range(len(X_train)):
            z=net_function(X_train[i],w,bias)
            out_pred=activation_function(z)
            errors = y_train[i] - out_pred
            w = w + learning_rate * X_train[i] * (errors)
            bias = bias + learning_rate * (errors)
            cost = cost+(errors**2)/2
        costs.append(cost[0])
    return w,bias,costs
```

Section B: Training Network & Plotting Cost-Function during Iterations

```
data_set=np.zeros((200,3))
data_set[0:100,0]=x1
data_set[0:100,1]=y1
data_set[0:100,2]=-1
data_set[100:200,0]=x2
data_set[100:200,1]=y2
data_set[100:200,2]=1
np.random.shuffle(data_set)
```

```
X_train=data_set[0:round(1*200),0:2]
y_train=data_set[0:round(1*200),2]
```

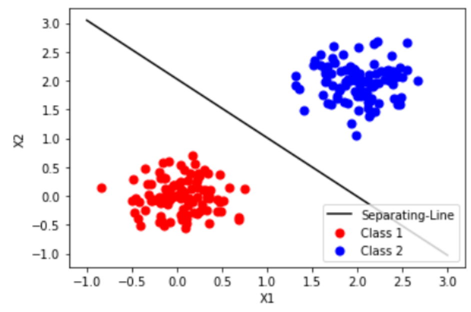
```
w,bias=init_weights()
learning_rate=0.01
iteration=30
```

```
trained_w,trained_b,costs=Adaline_training(X_train,w,bias,y_train,learning_rate,iteration)
```

```
xx=[-1,-1,0,1,2,3]
yy1=[]
for i in range (len(xx)):
    yy1.append(-xx[i]*(trained_w[0]/trained_w[1])-trained_b[0]/trained_w[1])
```

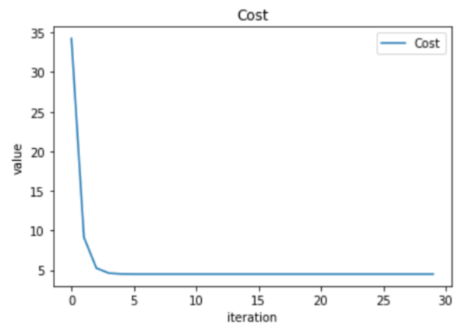
```
plt.scatter(x1,y1,s=50,c='r')
plt.scatter(x2,y2,s=50,c='b')
plt.plot(xx,yy1,'k')
plt.legend(["Separating-Line","Class 1", "Class 2"])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



```
plt.plot(costs)
plt.title("Cost")
plt.legend(["Cost"])
plt.xlabel('iteration')
plt.ylabel('value')
```

```
Text(0, 0.5, 'value')
```



Section C: Repeating all Analyses for new Categorization

```
# set one
x1 = np.random.normal(2,1,100)
y1 = np.random.normal(1,0.1,100)

# set two
x2 = np.random.normal(-1,0.4,20)
y2 = np.random.normal(2,0.4,20)
```

```
data_set=np.zeros((120,3))
data_set[0:100,0]=x1
data_set[0:100,1]=y1
data_set[0:100,2]=-1
data_set[100:120,0]=x2
data_set[100:120,1]=y2
data_set[100:120,2]=1
np.random.shuffle(data_set)
```

```
X_train=data_set[0:round(1*120),0:2]
y_train=data_set[0:round(1*120),2]
```

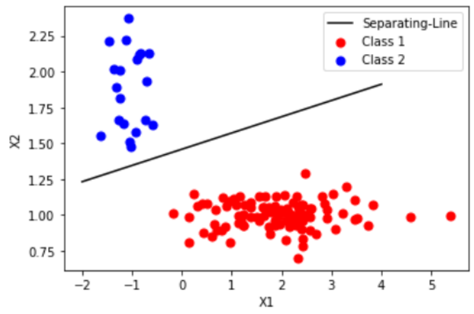
```
w,bias=init_weights()
learning_rate=0.001
iteration=1000
```

```
trained_w,trained_b,costs=Adaline_training(X_train,w,bias,y_train,learning_rate,iteration)
```

```
xx=[-2,-1,0,1,2,3,4]
yy1=[]
for i in range (len(xx)):
    yy1.append(-xx[i]*(trained_w[0]/trained_w[1])-trained_b[0]/trained_w[1])
```

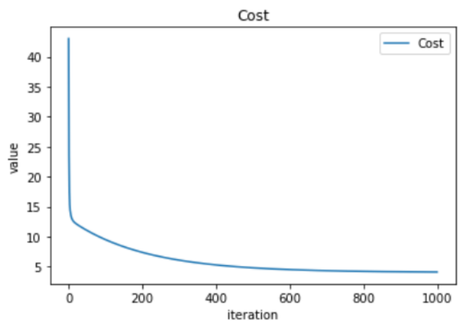
```
plt.scatter(x1,y1,s=50,c='r')
plt.scatter(x2,y2,s=50,c='b')
plt.plot(xx,yy1,'k')
plt.legend(["Separating-Line","Class 1", "Class 2"])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



```
plt.plot(costs)
plt.title("Cost")
plt.legend(["Cost"])
plt.xlabel('iteration')
plt.ylabel('value')
```

```
Text(0, 0.5, 'value')
```



4 QUESTION #3

4.1 MADALINE

In this part we intend to implement a classifier using Madeline concept which let us to find the separating lines which tend to separate our binary classes as well as it could be.

Note that all of implementation in this part has been done using basic python libraries such as NumPy and matplotlib for plotting the results so we can say it has been done from scratch!

4.1.1 MRI ALGORITHM

First of all, let’s declare Madeline as of our main purpose.

a MADALINE consists of Many Adaptive Linear Neurons arranged in a multilayer net. The examples given for the perceptron and the derivation of the delta rule for several output units both indicate there is essentially no change in the process of training if several ADALINE units are combined in a single-layer net. In this section we will discuss a MADALINE with one hidden layer (composed of different hidden ADALINE units) and one output ADALINE unit. Generalizations to more hidden units, more output units, and more hidden layers, are straightforward.

We consider the MRI algorithm:

In the MRI algorithm (the original form of MADALINE training) , only the weights for the hidden ADALINES are adjusted; the weights for the output unit are fixed.; the weights v1 and v2 and the bias b3 that feed into the output unit Y are determined so that the response of unit Y is 1 if the signal it receives from either Z1or Z2 (or both) is 1 and is -1 if both Z1 and Z2 send a signal of -1. In other words, the unit Y performs the logic function OR on the signals it receives from Z1and Z2.

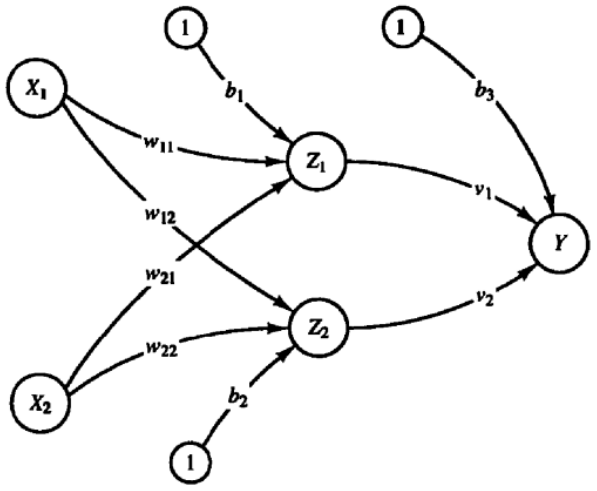


Figure 9: Architecture of an Madaline

The weights on the first hidden layer and the correspond biases are adjusted according to the algorithm which abbreviated below:

A training algorithm for a **Madaline** in an **MRI form** is as follows:

Step 0: Initialize weights. Weights v1 and v2 and the bias b3 are set as described: (small random values are usually used for ADALINE weights. Set the learning rate a as in the ADALINE training algorithm (a small value))

Step 1: While stopping condition is false, do Steps 2-8.

Step 2: For each bipolar training pair s: t, do Steps 3-7.

Step 3: Set activations of input units: xi= si.

Step 4: Compute net input to each hidden Adaline unit:

$z\_in_1 = b_1 + x_1 w_{11} + x_2 w_{21}$

$z\_in_2 = b_2 + x_1 w_{12} + x_2 w_{22}$

Step 5: Determine output of each hidden ADALINE unit:

$z_1 = f(z\_in_1)$.

$z_2 = f(z\_in_2)$.

Step 6: Determine output of net:

$y\_in \; = \; b_3 + z_1 v_1 + z_2 v_2;$

$y \; = \; f(y\_in).$

Step 7: Determine error and update weights:

If t = y, no weight updates are performed.

Otherwise:

If t = 1, then update weights on Z_j

the unit whose net input is closest to 0,

$bj(new) \; = \; b1(old) \; + \; a(1 — \; z\_in \, j),$

$wij(new) \; = \; wif(old) \; + \; all — \; z\_in \, j)xi \, ;$

If t = -1, then update weights on all units Z_k that have positive net input,

$bk(new) \; = \; bk(old) \; + \; a(— \; 1 — \; z — ink),$

$wik(new) \; = \; wik(old) \; + \; a(— \; 1 — \; z\_ink)xi.$

Step 8: Test stopping condition.

If weight changes have stopped (or reached an acceptable level), or if a specified maximum number of weight update iterations (Step 2) have been performed, then stop; otherwise continue.

4.1.2 Dispersion Representation of Dataset

Let's create the classes, the features of each class have been supported by Question3.csv excel file.

As you can see, we have two separated classes which have 0 & 1 as their labels.

The first thing should be issued is about replacing negative class labels with -1 instead of 0 in all rows of dataset.

Therefore, as the question has mentioned we have plotted these two classes in such a way that the vertical axis is belong to y variable and the horizontal axis is belong to x variable.

Further you can see more detailed result of this section:

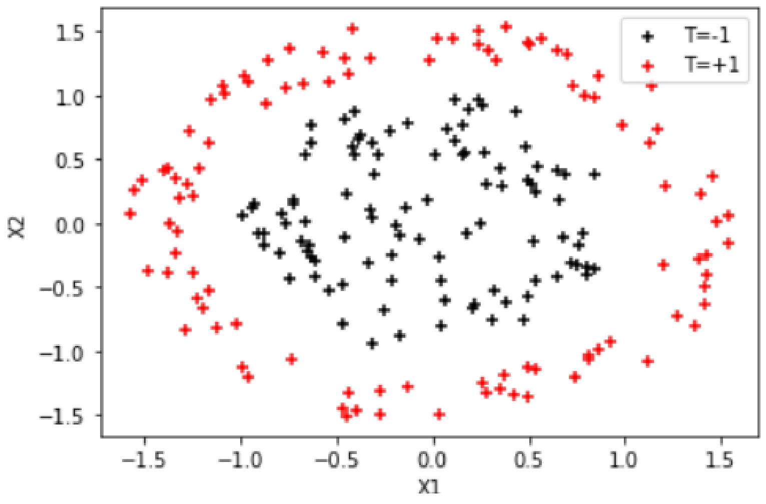


Figure 10: Representation of Data Dispersion

as you can see the result matches as well as we expected, and in whole of our further process the black class will present the first class **with label = -1** and the red class will present the second class **with label=1.**

4.1.3 Training our Model based on Adaline approach

Now all of things are prepared to train a model based on Madaline approach and in an MRI form which previously discussed.

In further process we tend to train a model based on Madaline approach to separate there two classes so the first thing we should take into account is about making dataset in such a way that mentioned classes will introduce with their features and correspond label in separate vector will show a specific data-point tend to be member of red class or maybe Black class.

So, in this regards I have provided piece of code which provide pair data-points in such a way that we have train features vector and correspond label vector separately.

Now let's find out how we can train a model **using one hidden layer consisting different number of neurons** that all have been abbreviated in **Madaline approach.**

In this section we do exact based on MRI algorithm which discussed in part 1.

I have written pieces of codes which support all from above notation and let us to have an adequate and exact training process **I have attached all of my code and functions which make those results in the end of this question.**

4.1.4 REPORTING RESULTS AND COMPARING

in this section we have reported the results for using different number of neurons in training process:

first, I have added the results and afterwards I have made a comparison between all at the end of this part.

Neuron-Number=3:

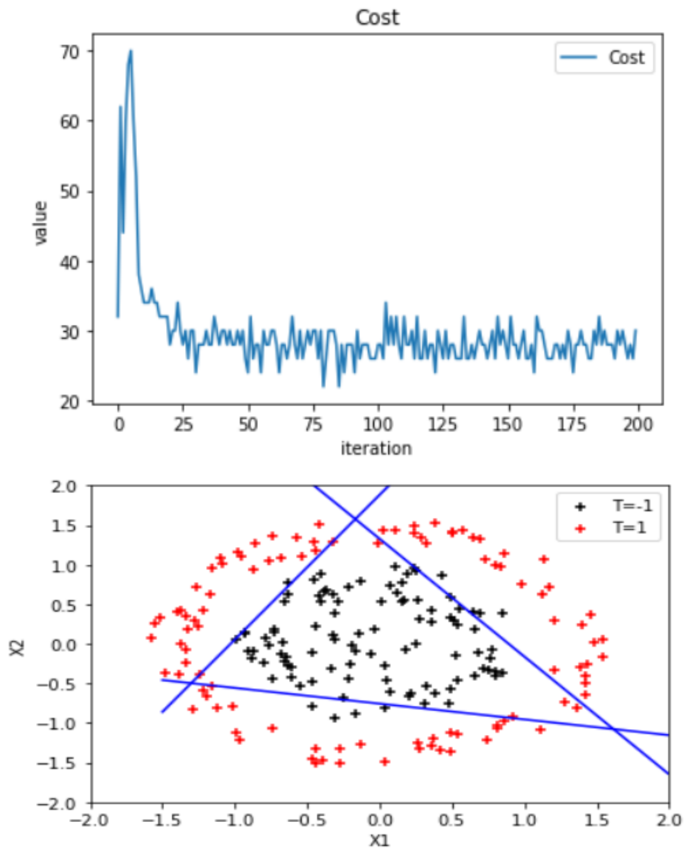


Figure 11: Classification Using Three neurons & Correspond Cost Function

Neuron-Number=4:

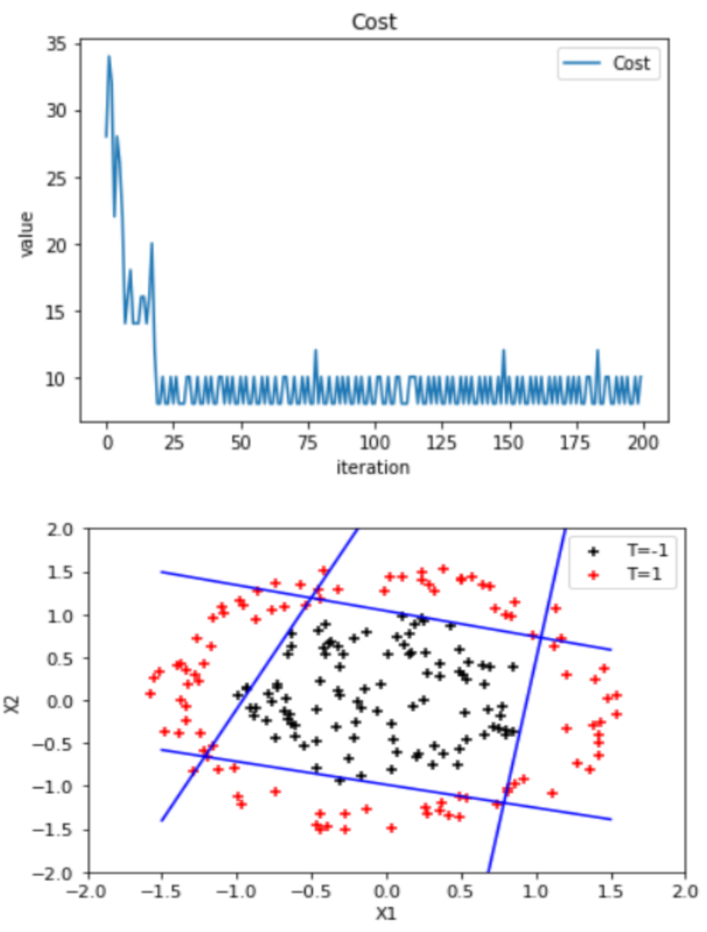


Figure 12: Classification Using Four neurons & Correspond Cost Function

Neuron-Number=8:

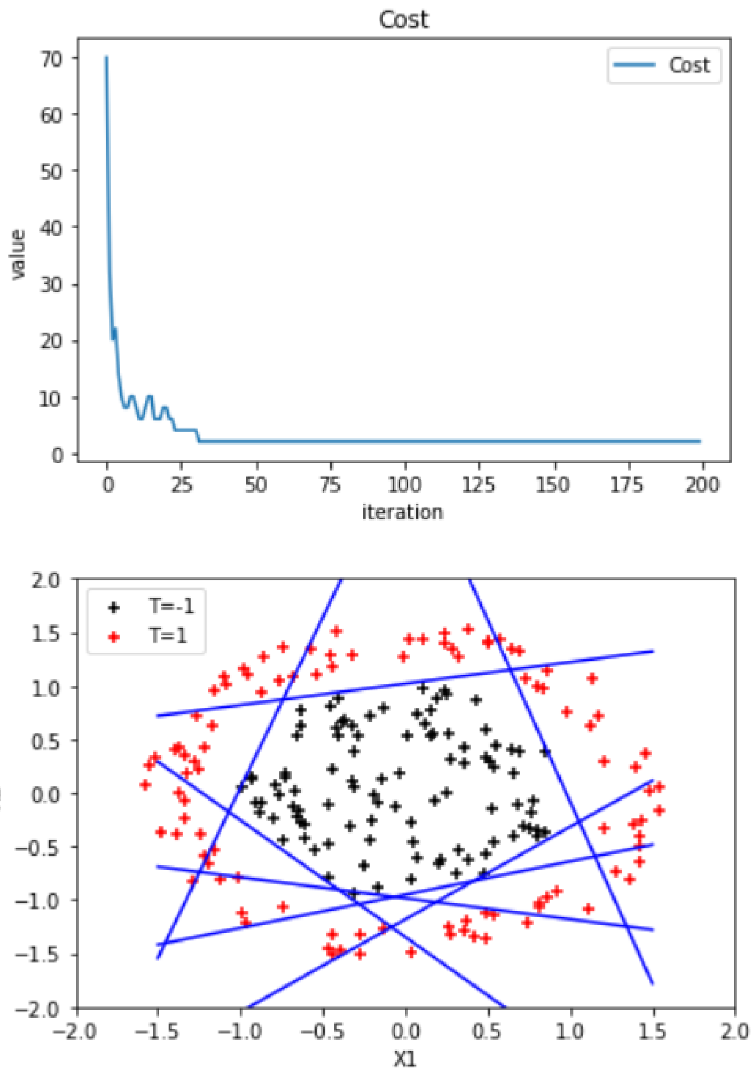


Figure 13: Classification Using Eight neurons & Correspond Cost Function

Analyse the results: As you can see the accuracy has increased when we use a greater number of neurons, for example when we used 4 neuron in hidden layer instead of 3 neuron obviously with four separator lines, we can separate class pretty much better and, in this case, we can cover more member of black class in the surface.

Also when we jump from 4 neurons to 8 neurons we can see that after convergence only 5 lines contributes for task of classification and rest of that doesn't contribute anymore because after a while the weight parameters have adjusted and also doesn't occur any updating on parameters then the other weighted has been fixed but in this state we have a more adequate classification task and using more lines let us to have a better cover on class data although the other line doesn't contribute at all.

One more thing I want to report is **about number of epochs** used in each state after several attempt to investigate convergence trend I found out that with increasing the neurons we meet the convergence much earlier, for getting more insight about that further I have reported the needed epochs number to meet convergence in each states:

if you can see in cost-function figures in all states, you will find out that in all cases we meet convergence as well as we supposed.

In each cases I have determined a **maximum epoch number** and run the algorithm during those epochs and when we meet the 100% accuracy or in other word waiting until weights don't change anymore then we have reported those epoch number as of our epoch which let us to meet convergence which abbreviated below :

For Three neurons Test : 150 epochs

For Four neurons Test : 70 epochs

For eight neurons Test : 25 epochs

As you can see convergence rate has been improved when we use greater number of neurons or needed neurons to break the while loop during the algorithm, we efficiently face with more convergence speed and also more stability during the algorithm, but in all cases, you can observe that after a while we face with a smooth figure that satisfies the convergence as well as we supposed.

Note: please note that these Reported-Number are based on my initial-random weights and obviously it may change when we dealing with different initialization.

Further you can see more detailed implementation in python environment, I would me my great honour if you could have a look on them .

Madaline Algorithm: MRI

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

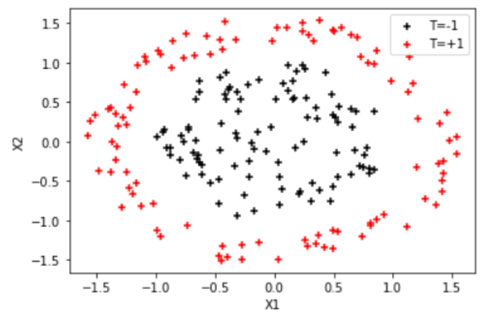
```
data = pd.read_csv("Question3.csv")
data=data.to_numpy()
```

```
# set one
x1 = data[0:99,0]
y1 = data[0:99,1]

# set two
x2 = data[99:199,0]
y2 = data[99:199,1]
```

```
plt.scatter(x1,y1,c='k',marker='+')
plt.scatter(x2,y2,c='r',marker='+')
plt.legend(["T=-1","T=+1"])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



```
def Accuracy(y_pred,y_test):
    n_zeros = np.count_nonzero((y_pred-y_test)==0)
    acc=n_zeros/len(y_pred)
    return acc
```

```
def activation_function(z):
    res=[]
    for i in range(len(z)):
        if z[i] >= 0 :
            res.append(1)
        else:
            res.append(-1)
    return res
```

```
def net_function(X,w,bias):
    res=np.dot(w,X)+bias;
    return res
```

```
def init_weights(neuron_Num):
    random_gen1 = np.random.RandomState(10)
    L1_weights = np.random.normal(0,0.01,(neuron_Num,2))
    random_gen2 = np.random.RandomState(20)
    L1_bias    = np.random.normal(0,0.01,neuron_Num)
    return L1_weights,L1_bias
```

```
def Adaline_training(X_train,w1,w2,bias1,bias2,y_train,learning_rate,iteration):
    costs=[]
    for epoch in range(iteration):
        cost=0
        for i in range(len(X_train)):
            z=net_function(X_train[i],w1,bias1)
            y_in=np.dot(w2,activation_function(z))+bias2;
            if y_in >= 0 :
                out_pred=1
            else :
                out_pred=-1
            errors = y_train[i] - out_pred
            if errors != 0 and y_train[i]==1 :
                idx=np.argmin(abs(z))
                p1=learning_rate * X_train[i][0]*(1-z[idx])
                p2=learning_rate * X_train[i][1]*(1-z[idx])
                p=np.zeros((len(w1),2))
                p[idx,0]=p1
                p[idx,1]=p2
                w1 = w1 + p
                bias1[idx] = bias1[idx] + learning_rate*(1-z[idx])
            if errors != 0 and y_train[i]==-1 :
                for i in range(len(z)):
                    if z[i]>0 :
                        p1=learning_rate * X_train[i][0]*(-1-z[i])
                        p2=learning_rate * X_train[i][1]*(-1-z[i])
                        p=np.zeros((len(w1),2))
                        p[i,0]=p1
                        p[i,1]=p2
                        w1 = w1 + p
                        bias1[i] = bias1[i] + learning_rate*(-1-z[i])
            cost = cost+(errors**2)/ 2.0
        costs.append(cost)
    return w1,bias1,costs
```

Training Network & Classifying with Different Number of Neurons

```
X_train=data[0:round(1*len(data)),0:2]
y_train=data[0:round(1*len(data)),2]
for i in range (len(y_train)):
    if y_train[i]== 0:
        y_train[i]=-1
```

Number of Neurons = 3

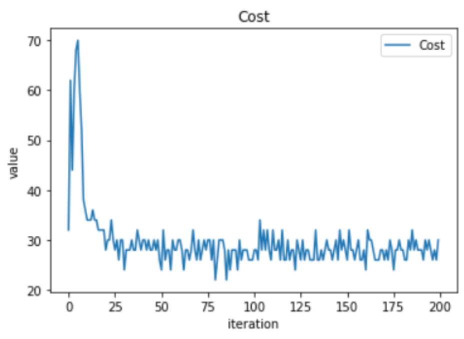
```
np.random.seed(19)
neuron_Num=3
w1,bias1=init_weights(neuron_Num)
w2=np.zeros((neuron_Num))
w2[0:neuron_Num]=1
bias2=neuron_Num-1
learning_rate=0.01
iteration=200
```

```
trained_w1,trained_b1,costs=Adaline_training(X_train,w1,w2,bias1,bias2,y_train,learning_rate,iteration)
```

Plotting Cost-Function & Separating Lines

```
plt.plot(costs)
plt.title("Cost")
plt.legend(["Cost"])
plt.xlabel('iteration')
plt.ylabel('value')
```

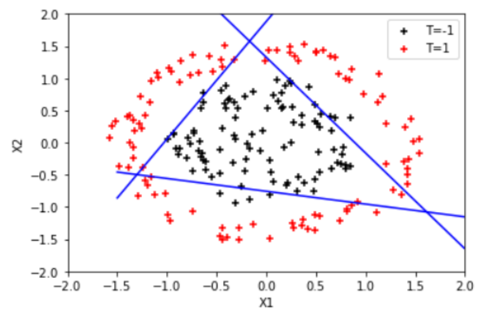
```
Text(0, 0.5, 'value')
```



```
domain=[-1.5,-1,0,1,1.5,2]
Line1=[]
Line2=[]
Line3=[]
for i in range (len(domain)):
    Line1.append(-domain[i]*(trained_w1[0][0]/trained_w1[0][1])-trained_b1[0]/trained_w1[0][1])
    Line2.append(-domain[i]*(trained_w1[1][0]/trained_w1[1][1])-trained_b1[1]/trained_w1[1][1])
    Line3.append(-domain[i]*(trained_w1[2][0]/trained_w1[2][1])-trained_b1[2]/trained_w1[2][1])
```

```
plt.scatter(x1,y1,c='k',marker='+')
plt.scatter(x2,y2,c='r',marker='+')
plt.legend(["T=-1","T=1"])
plt.plot(domain,Line1,'b')
plt.plot(domain,Line2,'b')
plt.plot(domain,Line3,'b')
plt.xlabel('X1')
plt.ylabel('X2')
plt.ylim([-2,2])
plt.xlim([-2,2])
```

```
(-2.0, 2.0)
```



Number of Neurons = 4

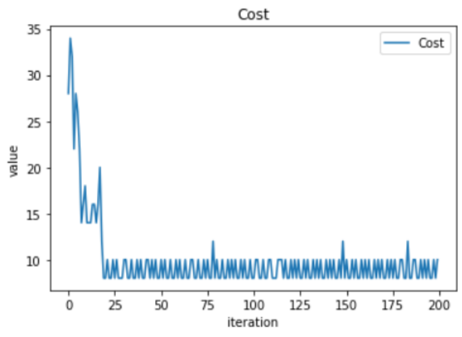
```
np.random.seed(21)
neuron_Num=4
w1,bias1=init_weights(neuron_Num)
w2=np.zeros((neuron_Num))
w2[0:neuron_Num]=1
bias2=neuron_Num-1
learning_rate=0.01
iteration=200
```

```
trained_w1,trained_b1,costs=Adaline_training(X_train,w1,w2,bias1,bias2,y_train,learning_rate,iteration)
```

Plotting Cost-Function & Separating Lines

```
plt.plot(costs)
plt.title("Cost")
plt.legend(["Cost"])
plt.xlabel('iteration')
plt.ylabel('value')
```

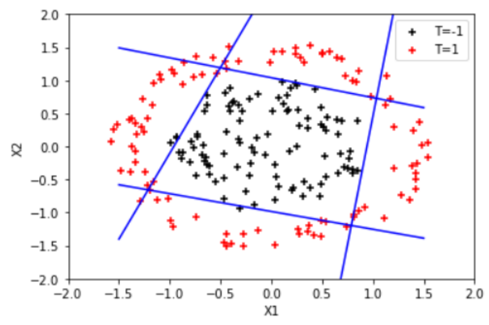
```
Text(0, 0.5, 'value')
```



```
domain=[-1.5,-1,0,1,1.5]
Line1=[]
Line2=[]
Line3=[]
Line4=[]
for i in range (len(domain)):
    Line1.append(-domain[i]*(trained_w1[0][0]/trained_w1[0][1])-trained_b1[0]/trained_w1[0][1])
    Line2.append(-domain[i]*(trained_w1[1][0]/trained_w1[1][1])-trained_b1[1]/trained_w1[1][1])
    Line3.append(-domain[i]*(trained_w1[2][0]/trained_w1[2][1])-trained_b1[2]/trained_w1[2][1])
    Line4.append(-domain[i]*(trained_w1[3][0]/trained_w1[3][1])-trained_b1[3]/trained_w1[3][1])
```

```
plt.scatter(x1,y1,c='k',marker='+')
plt.scatter(x2,y2,c='r',marker='+')
plt.legend(["T=-1","T=1"])
plt.plot(domain,Line1,'b')
plt.plot(domain,Line2,'b')
plt.plot(domain,Line3,'b')
plt.plot(domain,Line4,'b')
plt.xlabel('X1')
plt.ylabel('X2')
plt.ylim([-2,2])
plt.xlim([-2,2])
```

```
(-2.0, 2.0)
```



Number of Neurons = 8

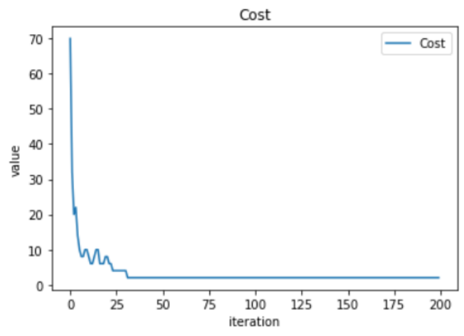
```
np.random.seed(21)
neuron_Num=8
w1,bias1=init_weights(neuron_Num)
w2=np.zeros((neuron_Num))
w2[0:neuron_Num]=1
bias2=neuron_Num-1
learning_rate=0.001
iteration=200
```

```
trained_w1,trained_b1,costs=Adaline_training(X_train,w1,w2,bias1,bias2,y_train,learning_rate,iteration)
```

Plotting Cost-Function & Separating Lines

```
plt.plot(costs)
plt.title("Cost")
plt.legend(["Cost"])
plt.xlabel('iteration')
plt.ylabel('value')
```

```
Text(0, 0.5, 'value')
```

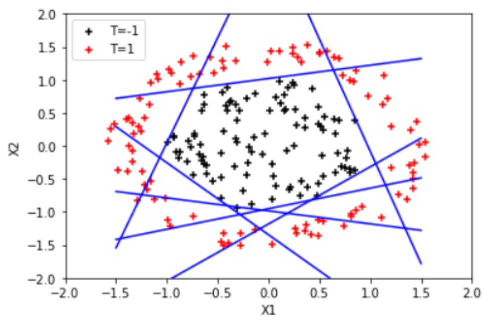


```
domain=[-1.5,-1,0,1,1.5]
Line1=[]
Line2=[]
Line3=[]
Line4=[]
Line5=[]
Line6=[]
Line7=[]
Line8=[]

for i in range (len(domain)):
    Line1.append(-domain[i]*(trained_w1[0][0]/trained_w1[0][1])-trained_b1[0]/trained_w1[0][1])
    Line2.append(-domain[i]*(trained_w1[1][0]/trained_w1[1][1])-trained_b1[1]/trained_w1[1][1])
    Line3.append(-domain[i]*(trained_w1[2][0]/trained_w1[2][1])-trained_b1[2]/trained_w1[2][1])
    Line4.append(-domain[i]*(trained_w1[3][0]/trained_w1[3][1])-trained_b1[3]/trained_w1[3][1])
    Line5.append(-domain[i]*(trained_w1[4][0]/trained_w1[4][1])-trained_b1[4]/trained_w1[4][1])
    Line6.append(-domain[i]*(trained_w1[5][0]/trained_w1[5][1])-trained_b1[5]/trained_w1[5][1])
    Line7.append(-domain[i]*(trained_w1[6][0]/trained_w1[6][1])-trained_b1[6]/trained_w1[6][1])
    Line8.append(-domain[i]*(trained_w1[7][0]/trained_w1[7][1])-trained_b1[7]/trained_w1[7][1])
```

```
plt.scatter(x1,y1,c='k',marker='+')
plt.scatter(x2,y2,c='r',marker='+')
plt.legend(["T=-1","T=1"])
plt.plot(domain,Line1,'b')
plt.plot(domain,Line2,'b')
plt.plot(domain,Line3,'b')
plt.plot(domain,Line4,'b')
plt.plot(domain,Line5,'b')
plt.plot(domain,Line6,'b')
plt.plot(domain,Line7,'b')
plt.plot(domain,Line8,'b')
plt.xlabel('X1')
plt.ylabel('X2')
plt.ylim([-2,2])
plt.xlim([-2,2])
```

```
(-2.0, 2.0)
```



For running this part, please have an accurate look on related directory and There you can easily find all of you need and also, I would appreciate it if you would consider them ☺

5 QUESTION #4

5.1 PERCEPTRON (THEORETICAL BASED)

In this part we intend to analyse some common topics about perceptron and get a better insight about theoretical rules which support this algorithm and finally become familiar with updating weights and how we should deal with them.

First let's start with perceptron main concepts:

Perceptron is a linear classifier (binary). Also, it is used in supervised learning. It helps to classify the given input data.

The perceptron consists of 4 parts:

1.Input values or One input layer

2.Weights and Bias

3.Net sum

4.Activation Function

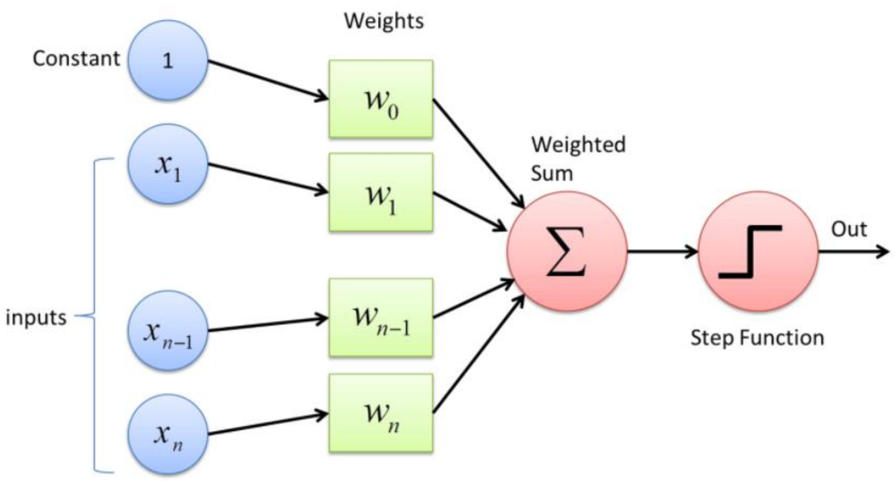


Figure 14: Architecture of Perceptron

1.All the inputs x are multiplied with their weights w

2.Then sum all the multiplied values and call them Weighted Sum

3.Apply that weighted sum to the correct Activation Function.

In most cases we use zero-thresholding activation function as we have done in this question!

One more thing we should care is about updating rule which is the half of story, if we don't know how to learn the parameters, we may didn't meet convergence at all so it plays an import role in our case.

$(*)\ w_i(new) = w_i(new) + ax_i t$

$(**)\quad b(new) = b(old) + \alpha t$

where x_i denotes ith input and t denotes the target value for output.

Now, let's have the dive on our problem:

W1	0.2
W2	0.7
W3	0.9
bias	-0.7

Table 1: design parameters

We have above parameters as our initial values, the learning rate equals to $\alpha = 0.3$ and our target equals to -1

X = $(0,1,1)$

And as me mentioned before the activation function is a zero-thresholding unit.

Iteration 1:

$net = w_2 + w_3 - 0.7 \quad \xrightarrow{yields} \quad net = 0.7 + 0.9 - 0.7 = 0.9$

Now we pass the net value from our activation function to find the predicted-target:

$f(net) = f(0.9) = 1$

so, we have error because the target value and the predicted-value are not same.

updating parameters:

$w_1(new) = w_0(new) + ax_i t = (0.2,0.7,0.9) - 0.3(0,1,1) = (0.2,0.4,0.6)$

$b(new) = b(old) + \alpha t = -0.7 - 0.3 = -1$

So, we must continue the steps to meet convergence:

Iteration 2:

$net = w_2 + w_3 - 1 \quad \xrightarrow{yields} \quad net = 0.4 + 0.6 - 1 = 0$

Now we pass the net value from our activation function to find the predicted-target:

$f(net) = f(0) = 0$

so, we have error because the target value and the predicted-value are not same.

updating parameters:

$w_2(new) = w_1(new) + ax_it = (0.2,0.4,0.6) - 0.3(0,1,1) = (0.2,0.1,0.3)$

$b(new) = b(old) + \alpha t = -1 - 0.3 = -1.3$

So, we must continue the steps to meet convergence:

Iteration 3:

$net = w_2 + w_3 - 1.3 \quad \xrightarrow{yields} \quad net = 0.1 + 0.3 - 1.3 = -0.9$

Now we pass the net value from our activation function to find the predicted-target:

$f(net) = f(-0.9) = -1$

so, the target-value and the predicted-value are same but for ensuring that we have met convergence we will iterate one more time.

updating parameters:

$w_3(new) = w_2(new) + ax_it = (0.2,0.1,0.3) - 0.3(0,1,1) = (0.2,-0.2,0)$

$b(new) = b(old) + \alpha t = -1.3 - 0.3 = -1.6$

$net = w_2 + w_3 - 1.3 \quad \xrightarrow{yields} \quad net = -0.2 + 0 - 1.6 = -1.8$

As we can see after three iterations, we meet convergence and the predicted and target value are match as well as we suppose.

6 ACKNOWLEDGEMENT

I am really grateful for Mr MohammadHossein Vaeedi (810197605) because in some part of this project we had a nice and effective discussion which highly helped us to have a better analyse and also have more adequate results! (Note that we only talked about ideas behind different problems.)

Afterwards, I am thankful to all of course teaching assistants: MehrdadMomen(Mehrdad.Momen@ut.ac.ir) and KamyabMorshedi(kamyabmorshedihaghighi@gmail.com) who designed this project with high quality.

7 REFERENCES

[1] https://sebastianraschka.com/faq/docs/diff-perceptron-adaline-neuralnet.html
[2]https://towardsdatascience.com/adaline-neural-networks-the-origin-of-gradient-descent-783ed05d7c18
[3] https://vitalflux.com/adaline-explained-with-python-example/
[4] https://blog.oureducation.in/adaline-madaline/
[5] https://dzone.com/articles/adaline-explained-with-python-example-data-analyti
[6] https://github.com/abundrew/ml/blob/master/adaline.ipynb
[7] https://www.geeksforgeeks.org/implementing-or-gate-using-adaline-network/
[8]https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_supervised_learning.htm
[9] https://www.geeksforgeeks.org/full-adder-in-digital-logic/