Deep Learning- CSE641

Assignment - 1

Name: Akanksha Shrimal **Roll No:** MT20055
Name: Vaibhav Goswami **Roll No:** MT20018
Name: Shivam Sharma **Roll No**: MT20121

FILES SUBMITTED : submitted .py file , .ipynb file and readme.pdf and output.pdf

1. Perceptron Training Algorithm:

METHODOLOGY :-

1. Perceptron Training Algorithm checks for misclassification of a sample and updates the weights if the sample is misclassified.
2. To check whether a sample is misclassified, the error condition is checked.
   Error condition: $yw^Tx \leq 0$
3. If the error condition is true, weights are updated as follows:
   $\Delta w = yx$
   $w = w + \Delta w$
4. Convergence of the algorithm is achieved when all the samples are correctly classified, i.e. there is no error.

PRE-PROCESSING: Data for points (input and output) was provided for each gate.

1(a) and (b) COMPUTING PTA - AND , OR, NOT

Methodology :-

- Initial weights are randomly selected.
- In each epoch, all the samples are checked and weights are updated if the sample is misclassified.
- To check if all the samples are correctly classified, *temp* list (boolean) is used.
- If all the samples are correctly classified, *temp* will have all ones, otherwise there will be at least a zero in the list.
- When the weights are updated, the plot is made for the decision boundary.
- *count* variable is used to keep track of the number of steps it takes for the algorithm to converge.

1. **AND Gate:**
   Initial weights are set as w1 = 2, w2 = 1, bias (or w0) = -3.
   Truth Table:

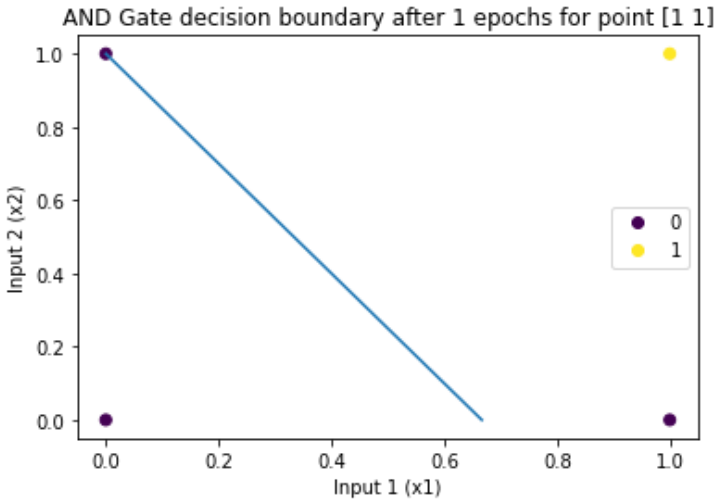
Input (x1)	Input (x2)	Output (y)
0	0	-1
0	1	-1
1	0	-1
1	1	1

Note: Output is 1 and -1 instead of 1 and 0 for mathematical convenience.

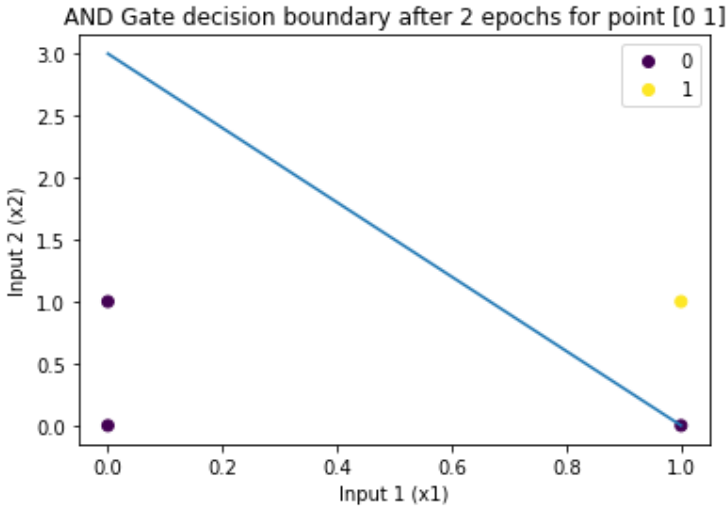
Convergence steps:- For AND gate, PTA took 7 steps for convergence.

Graphs:-

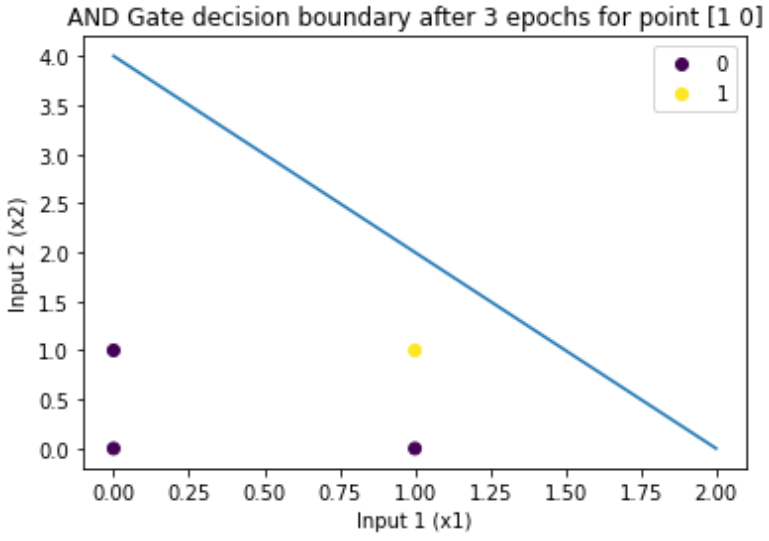
Step 1:



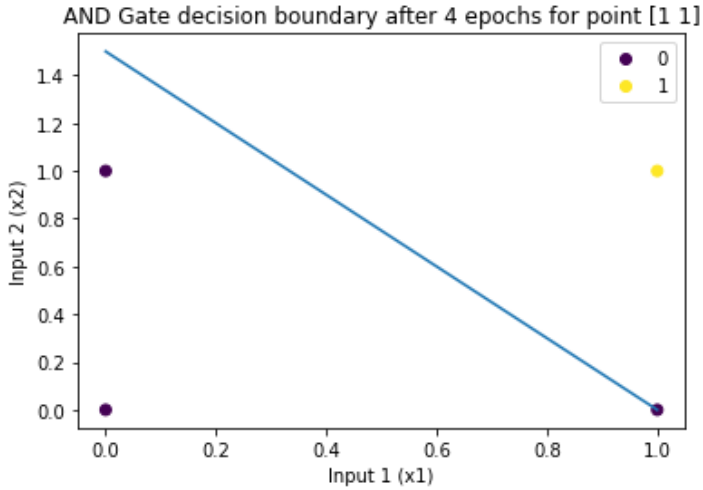
Step 2:



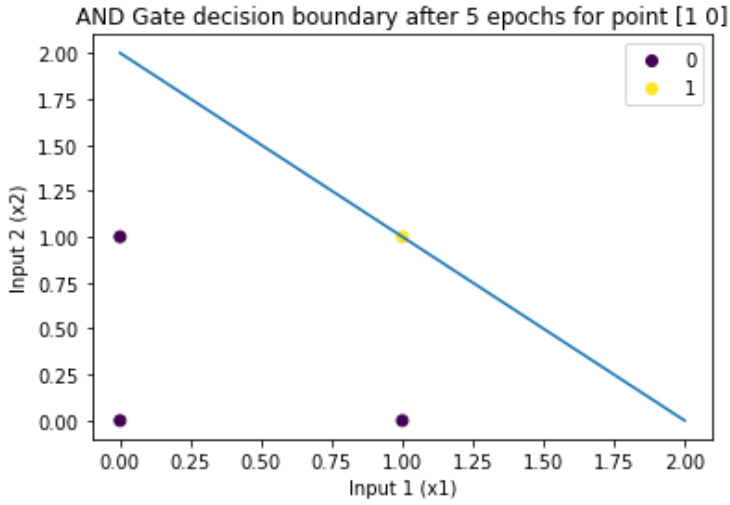
Step 3:



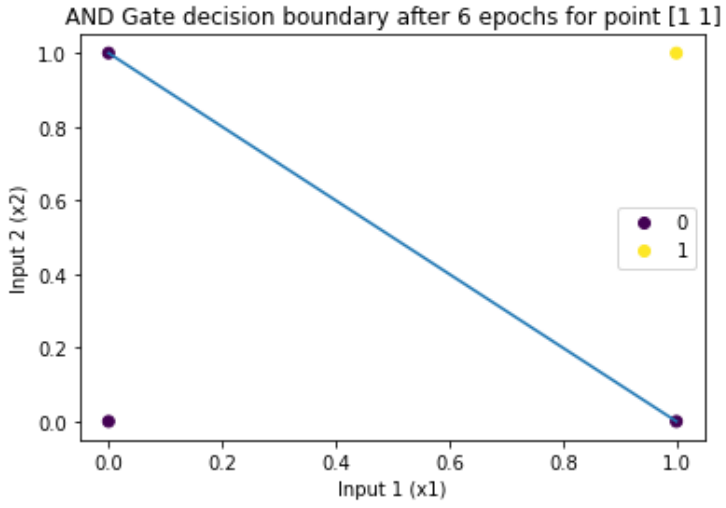
Step 4:



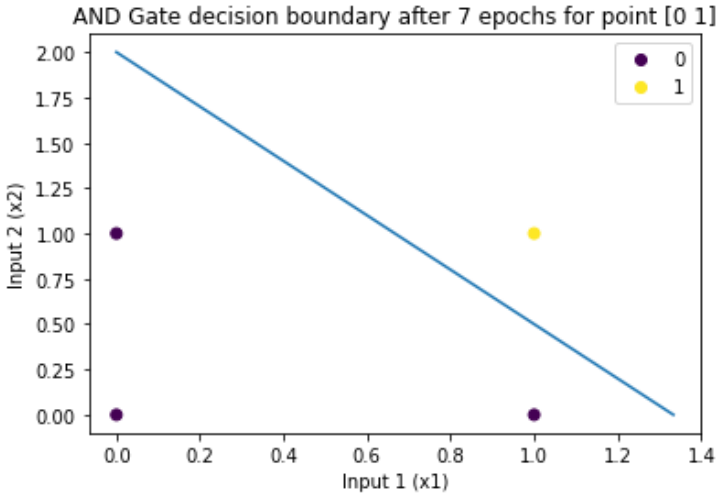
Step 5:



Step 6:



Step 7:

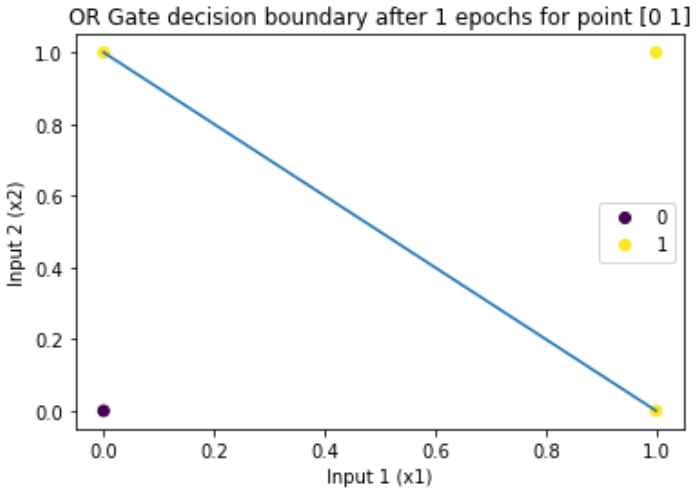


2. **OR Gate:**
Initial weights are set as w1 = 2, w2 = 1, bias (or w0) = -3.
Truth Table:

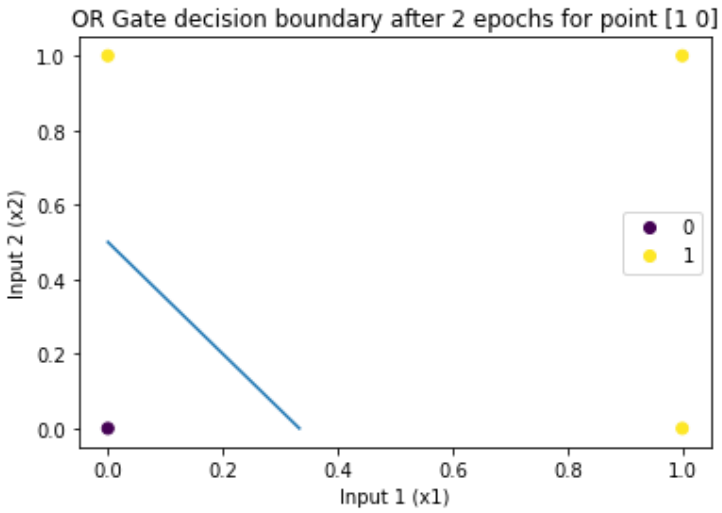
Input (x1)	Input (x2)	Output (y)
0	0	-1
0	1	1
1	0	1
1	1	1

Convergence steps:- For OR gate, PTA took 2 steps for convergence.
Graphs:-

Step1:



Step 2:



3. **NOT**

Initial weights are set as w1 = 1.5, bias (or w0) = 0.5.

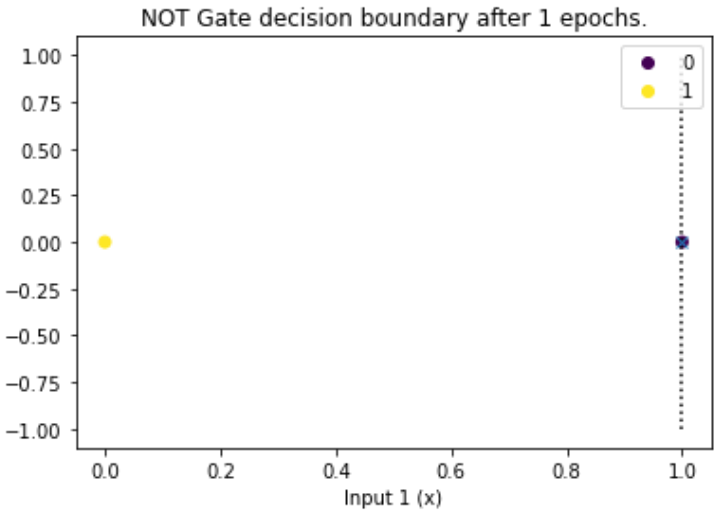
Truth Table:

Input(x)	Output(y)
0	1
1	-1

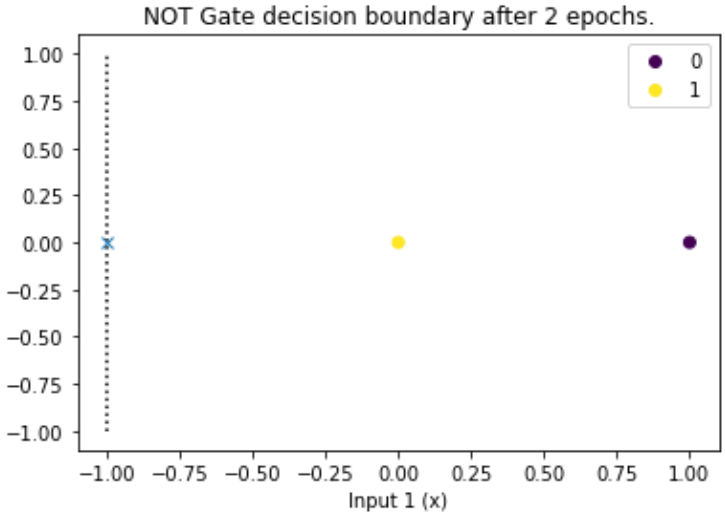
Convergence steps:- For NOT gate, PTA took 6 steps for convergence.

Graphs:-

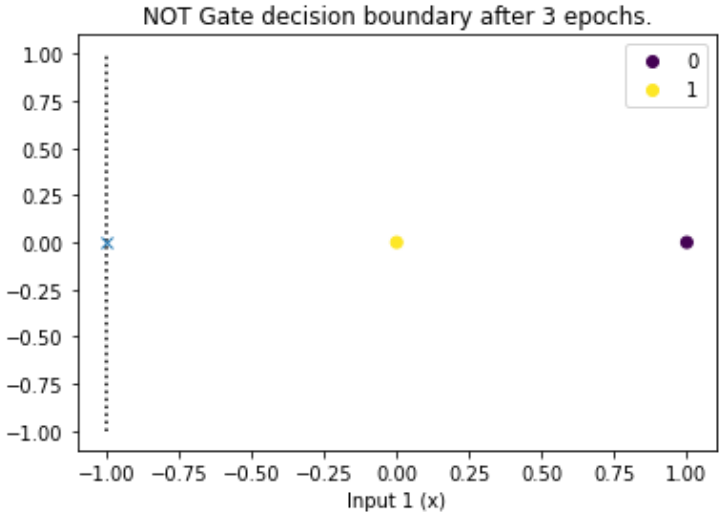
Step 1:



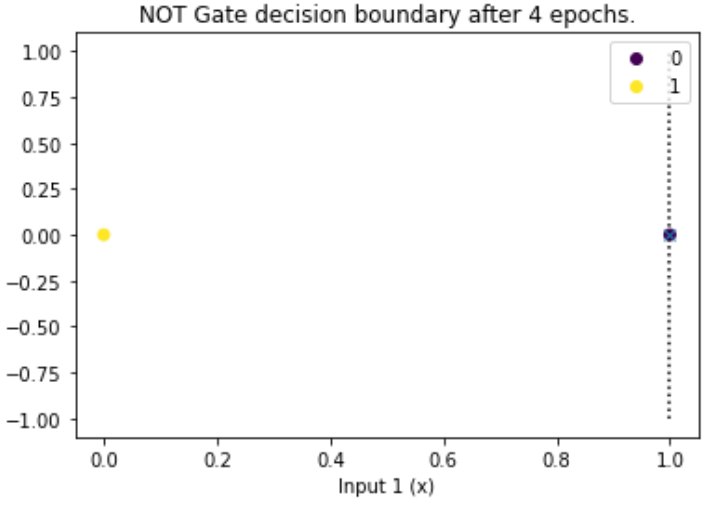
Step 2:



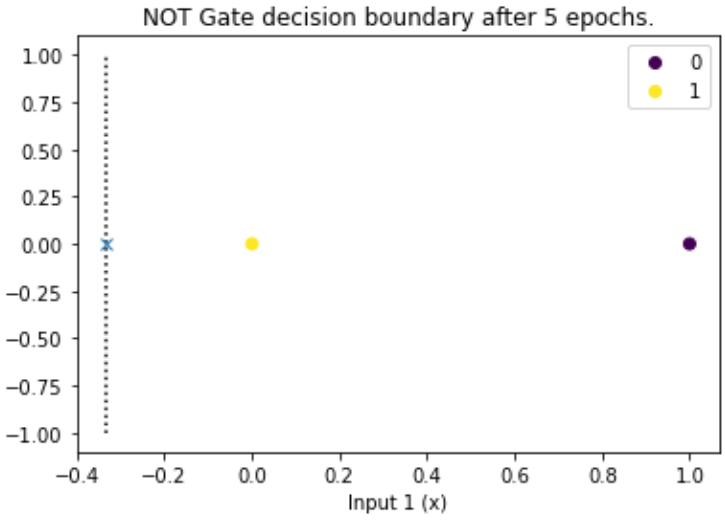
Step 3:



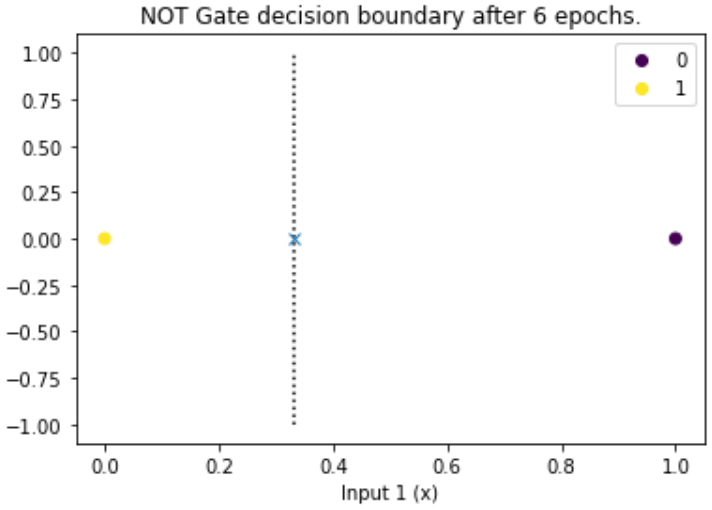
Step 4:



Step 5:



Step 6:



Note:

- Data for NOT gate is 1-dimensional. Therefore, points lie on the same line.
- As data is 1-D, there would be a separating point rather than a boundary. In the plots, point 'x' denotes the same.
- Dotted line is just for a better understanding of separation between two classes.

The following observations are made :

- If data is linearly separable it can be classified using perceptron.

1(c) XOR using PTA

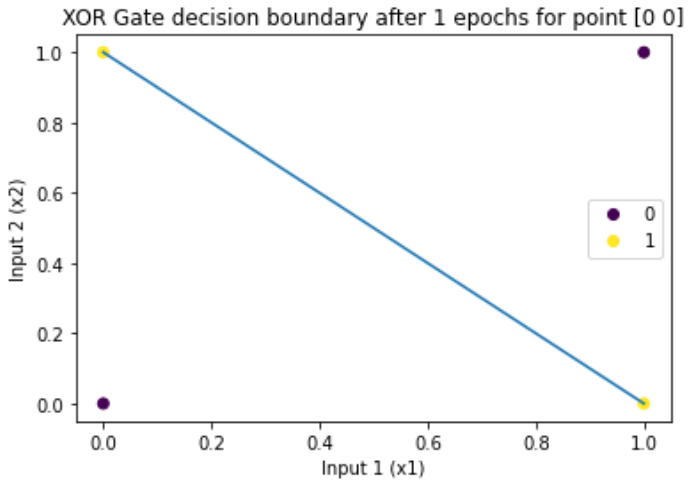
Initial weights are set as w1 = 0.5, w2 = 0.5, bias (or w0) = 0.5.
Maximum epochs used are 10.
Truth Table:

Input(x1)	Input(x2)	Output(y)
0	0	0
0	1	1
1	0	1
1	1	0

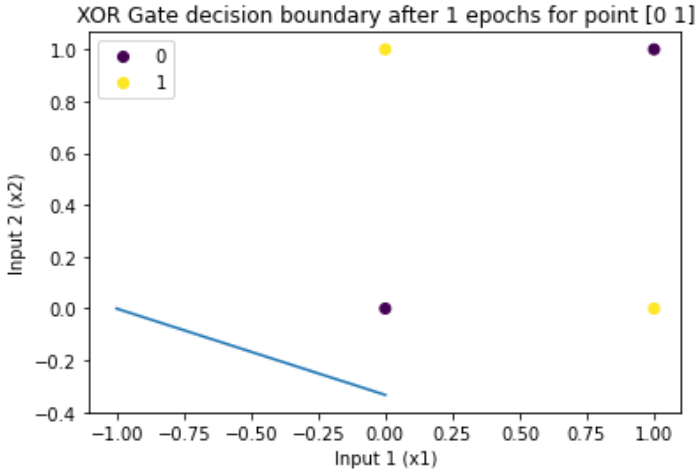
Note: XOR is not a linearly separable data (in 2-dimensional plane).

Graphs:

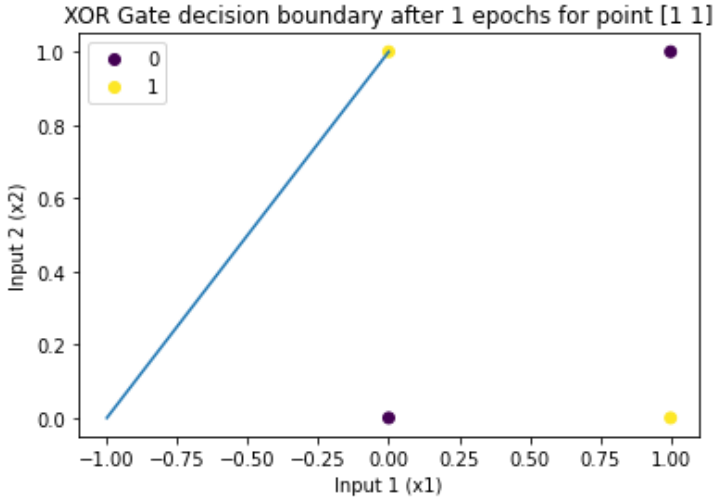
Step 1:



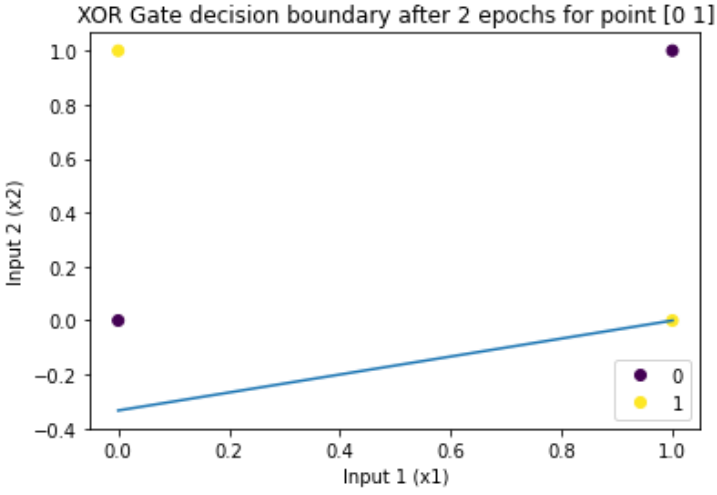
Step 2:



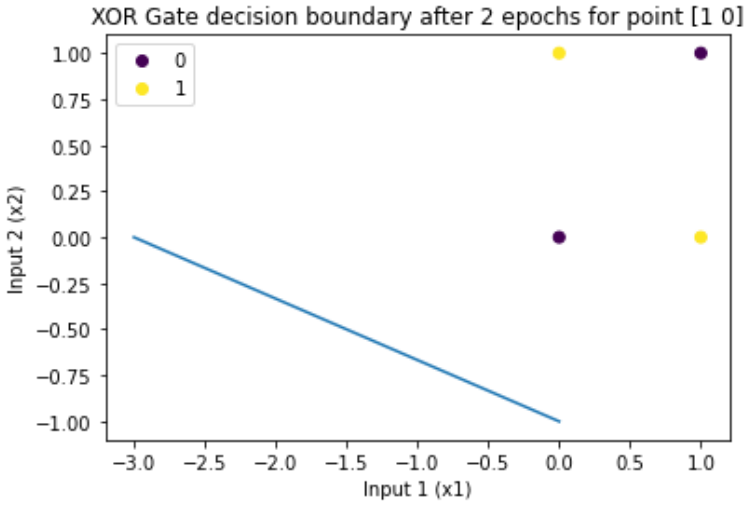
Step 3:



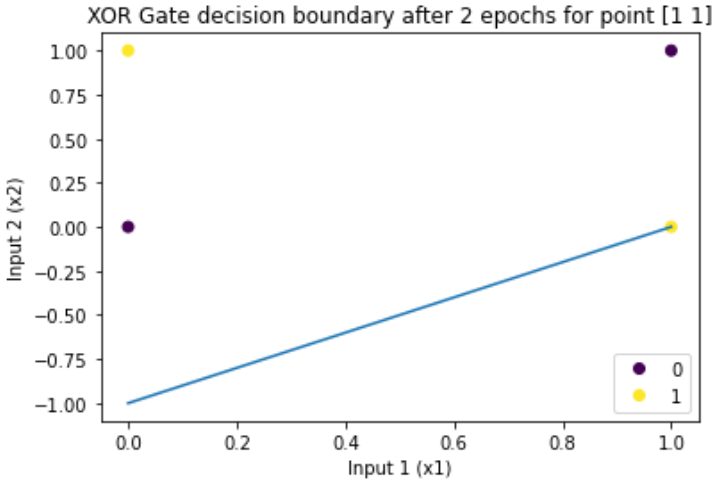
Step 4:



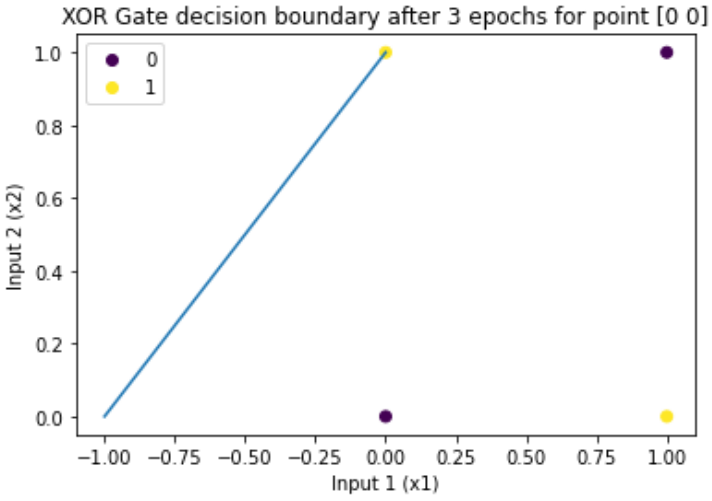
Step 5:


XOR Gate decision boundary after 2 epochs for point [1 0]
0
1
Input 2 (x2)
Input 1 (x1)

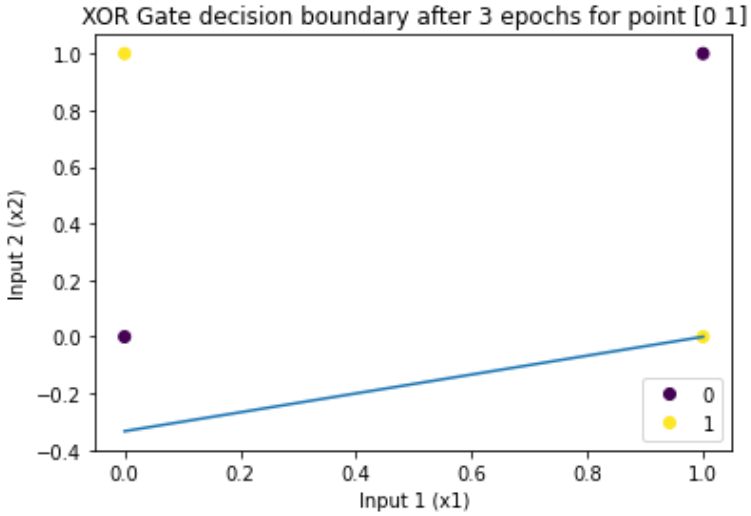
Step 6:


XOR Gate decision boundary after 2 epochs for point [1 1]
0
1
Input 2 (x2)
Input 1 (x1)

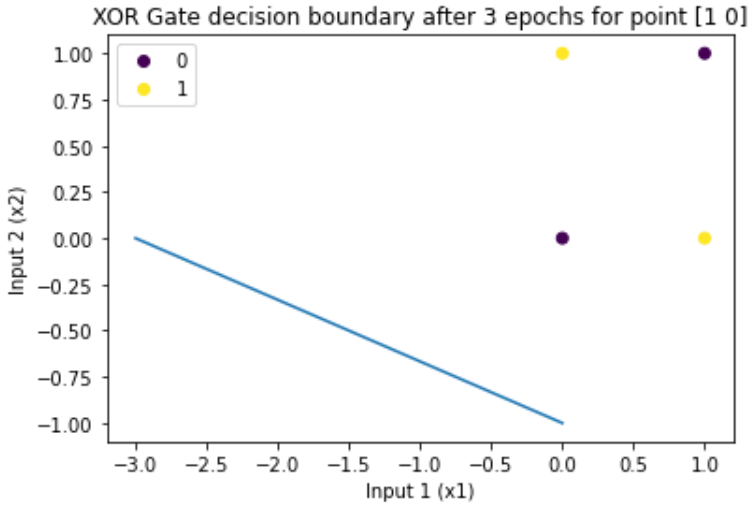
Step 7:


XOR Gate decision boundary after 3 epochs for point [0 0]
0
1
Input 2 (x2)
Input 1 (x1)

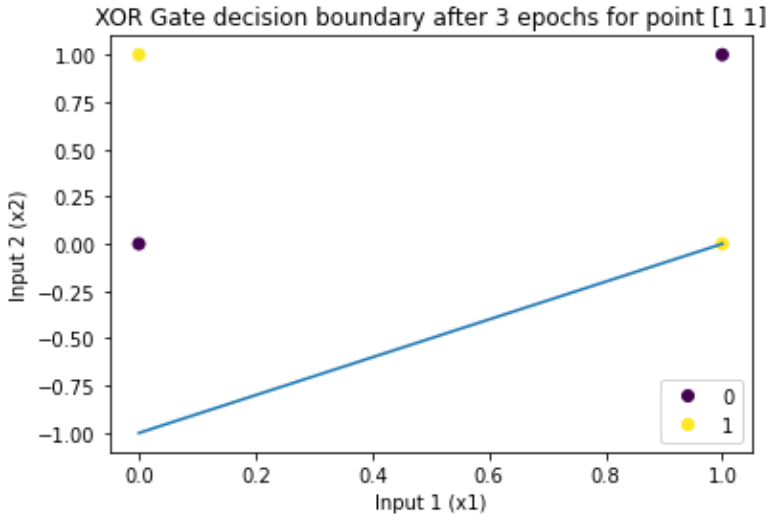
Step 8:



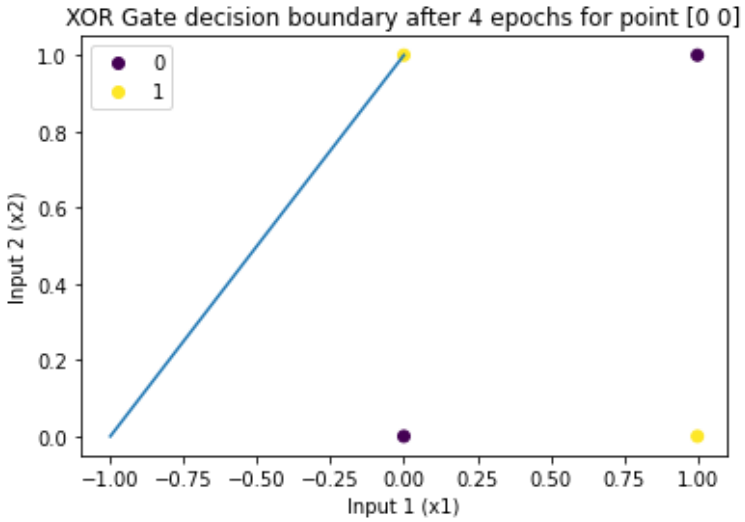
Step 9:



Step 10:



Step 11:



And so on…

Observations:

- The data (XOR) is not linearly inseparable.
- Cycling Theorem states that if the training data is not linearly separable, the algorithm will eventually repeat the same set of weights and, hence, the same set of lines and enter an infinite loop.
- From the plots above, it can be clearly seen that the Cycling Theorem holds as same set of lines is being repeated in a pattern and the algorithm never converges (infinite loop).
- The minimum number of steps required to prove the same is 10.
- It is because steps 3-6 are the set of steps that start repeating (set of 4).
- Therefore, steps 7-10 will verify the pattern as they will match steps 3-6 and verify the Cycling Theorem.
- The set of steps will keep on repeating in an infinite loop.

2. Madaline Learning Algorithm:

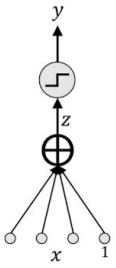
METHODOLOGY :-

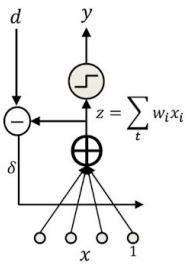
1. It follows a greedy approach and updates the weights only based on those instances which are mis-classified.
2. Data is defined and divided into training and testing sets.
3. For training and testing purposes the labels are made bipolar i.e 1 taken as 1 and 0 taken as -1.
4. A fully connected network is defined and weights are randomly initialised along with biases.
5. In each epoch do the following for each instance:-
    - Feed forward across the network
    - Find final y_hat, If d (desired output) == y_hat , do nothing correctly classified
    - If d (desired output) != y_hat , then for all adeline (hidden units) repeat till error is zero or no more hidden units left
        - find the hidden layer unit with lowest affine value i.e lowest confidence in prediction
        - Tentatively flip the sign of the unit. Apply feed forward with flipped value and find if d == y_hat. If so apply adeline on the respected unit, else move to next lowest affine value.

Note:- For each neuron, threshold activation function is used.
Madaline uses Adaline LMS rule for weight update

Adaline

- Adaptive Linear Element





$$Err(x) = \frac{1}{2}(d - z)^2$$

$$\frac{dErr(x)}{dw_i} = -(d - z)x_i$$

$$w_i = w_i + \eta(d - z)x_i$$

$$w_i = w_i + \eta\delta x_i$$

Delta or *LMS* weight update rule!

Pre-processing steps :- Data generation and Bipolarization of data (labels 0 are made -1 and labels 1 is kept 1)

Helper Function :-

To Plot Graph

```
def plot_double_line_graph(X1,Y1,label1 ,X2 ,Y2,label2 ,title):
 fig = plt.figure(figsize=(7,5))
 plt.subplot(111)
  plt.plot(X1,Y1 ,label=label1 ,marker = "x" , color="blue")
 plt.plot(X2, Y2 , label=label2 ,marker = "x" , color="red")
 plt.title(title)
 plt.ylabel('Accuracy')
 plt.xlabel('Epochs')
 plt.legend( loc='upper left',prop={'size': 13})
 plt.show()
```

Adeline function to update weights

```
 def apply_adeline(layer,neuron_num,d, z, eta, w, b,inputs):
   # print("w,b", w,b)
   for idx,row_val in enumerate(w[layer]):

     row_val[neuron_num] = row_val[neuron_num] + eta * (d - z) *
inputs[idx]

   b[layer][0][neuron_num] = b[layer][0][neuron_num] + eta * (d - z)
   print("Neuron found -----> Updated")
```

Function to flip y of one adeline unit

```
 def flip_y(layer,row,col,y):
   if y[layer][row][col] == 1:
     y[layer][row][col] = -1
   else:
     y[layer][row][col] = 1
   return y
```

There are two more functions for testing and classification

2(a) f1(x1,x2)

1. **Network Details / No of neurons**

   I have used a two hidden layer network for this function. A two hidden layer network is necessary to learn the given function because of the nature of threshold function (only tells on which side of boundary positive sample is but not where or how much far)

   NETWORK DETAILS

   Input layer

   Hidden layer 1 = 8 perceptrons

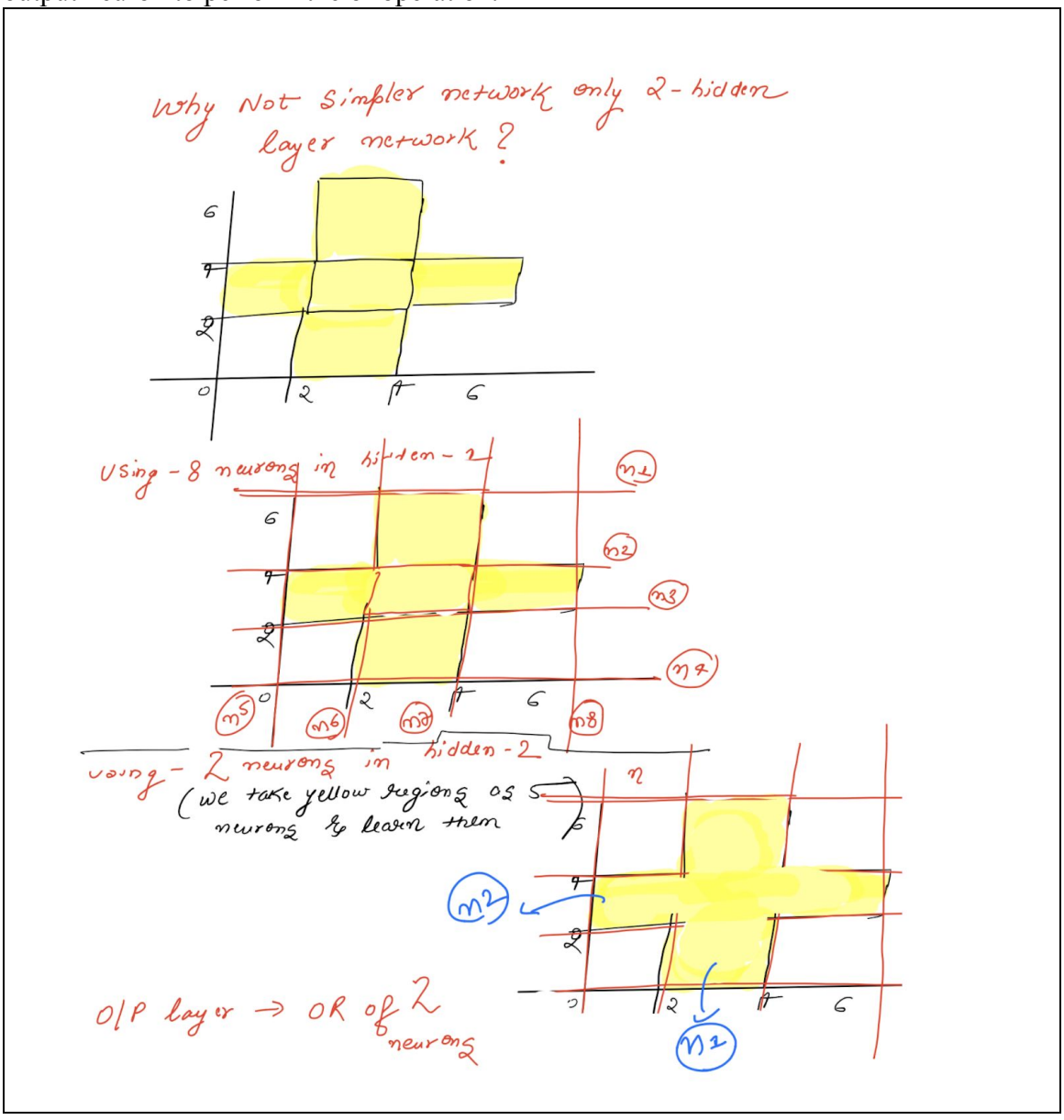
   Hidden layer 2 = 2 perceptrons

   Output layer = 1 perceptron

   TOTAL = 11 perceptrons (excluding input layer)

   Why only this neural network

   Using the first 8 neurons of hidden layer 1 i.e 4 horizontal and 4 vertical boundaries we identify the regions intended. Next 2 neurons are to identify exact positive areas and output neuron to perform the or operation.



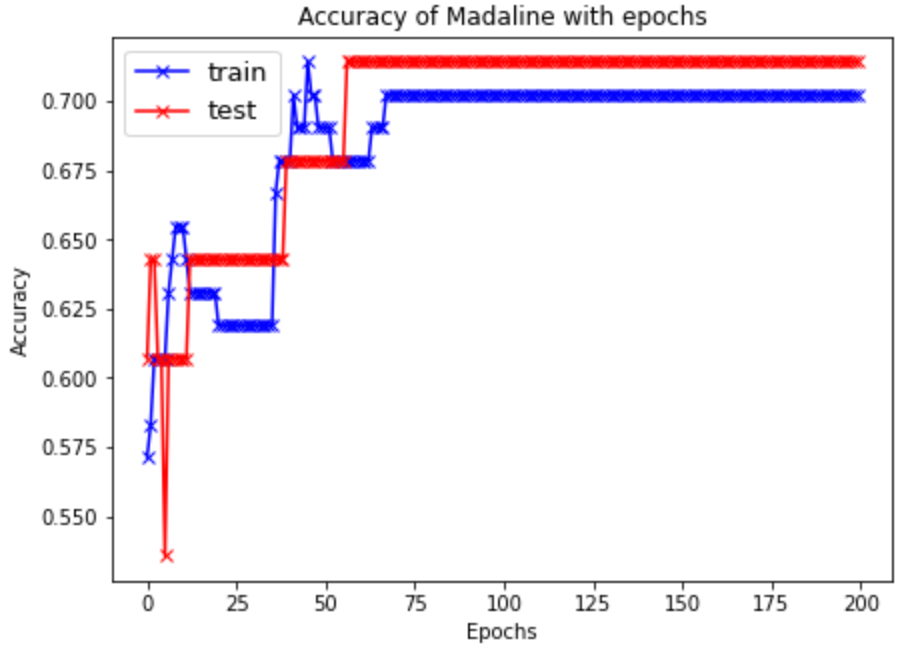
2. Implementation

DATA - 112 samples (Train : 126 Test: 42 -> stratified splitting used)

Epoc wise weight updated on each misclassified instance

```
---------------------------------------- 193 ----------------------------------------
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Correctly classified
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Correctly classified
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Incorrectly classified
Neuron found -----> Updated
Correctly classified
Correctly classified
Correctly classified
```

Epoc vs Accuracy



4. Results and Accuracy

TRAINING CLASSIFICATION REPORT

TRAIN

	precision	recall	f1-score	support
-1.0	1.00	0.36	0.53	39
1.0	0.64	1.00	0.78	45
accuracy			0.70	84
macro avg	0.82	0.68	0.66	84
weighted avg	0.81	0.70	0.66	84

TESTING CLASSIFICATION REPORT

TEST

	precision	recall	f1-score	support
-1.0	1.00	0.38	0.56	13
1.0	0.65	1.00	0.79	15
accuracy			0.71	28
macro avg	0.83	0.69	0.67	28
weighted avg	0.81	0.71	0.68	28

5. Observation
   - It is observed that weight initialization plays a very crucial role in weight updation using a madaline algorithm.
   - Here it can be observed that the algorithm gives low accuracy on training as well as test dataset even after 200 epocs and both train and test accuracy remains constant after 100 epocs and no significant learning is happening after it.
   - As train and test accuracy is low, the madaline algorithm is underfitting the given function. This is because the function is complex as it involves two hidden layers and madaline does not consider multiple neurons together while weight updation.
   - Thus the above results clearly show that the madaline algorithm is not successful in learning this function and underfits the given function.

2(b) f2(x1,x2)

1. **Network Details / No of neurons**

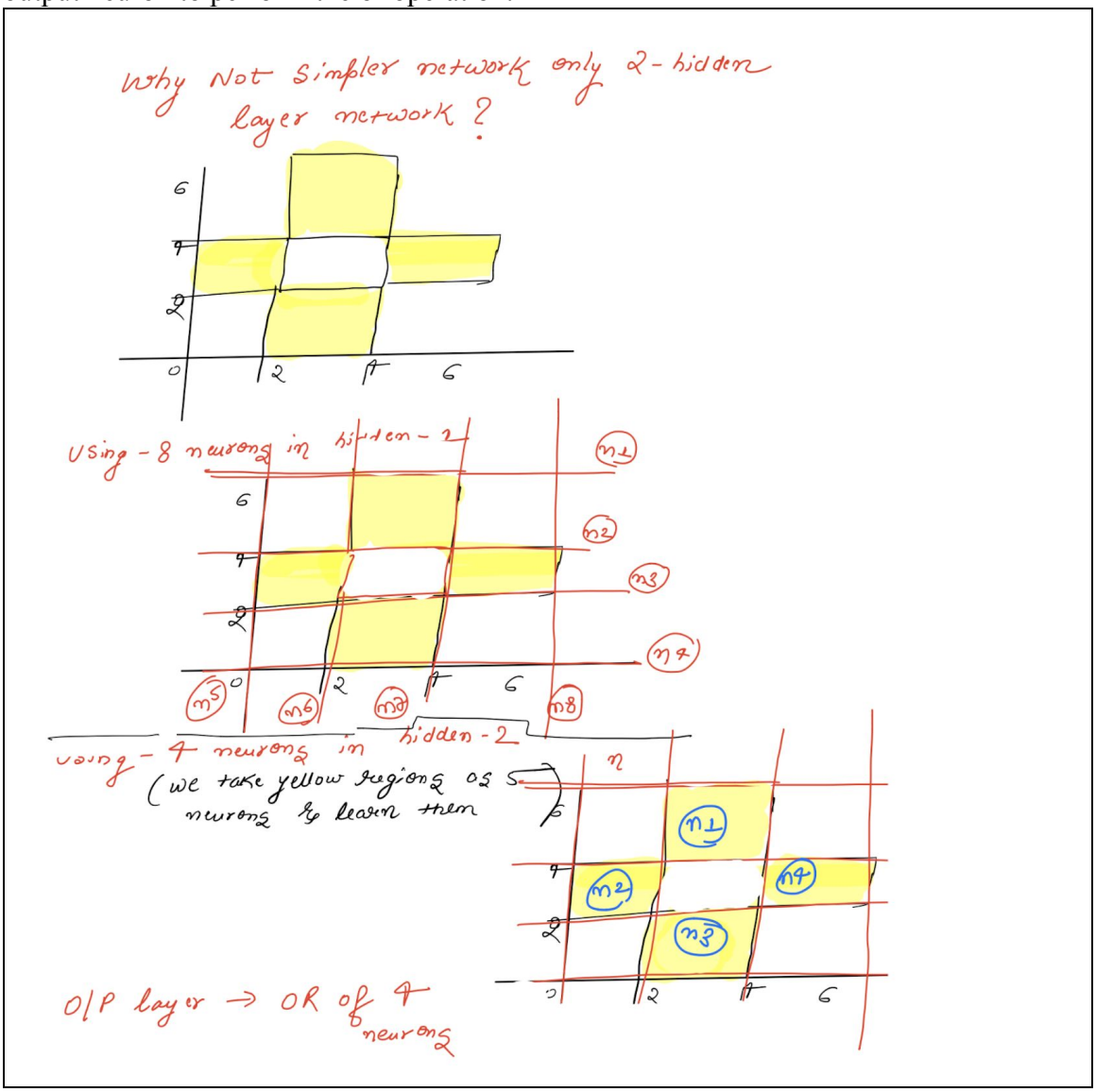
   I have used a two hidden layer network for this function. A two hidden layer network is necessary to learn the given function because of the nature of threshold function (only tells on which side of boundary positive sample is but not where or how much far)

   NETWORK DETAILS

   Input layer
   Hidden layer 1 = 8 perceptrons
   Hidden layer 2 = 4 perceptrons
   Output layer = 1 perceptron
   TOTAL = 13 perceptrons (excluding input layer)

   Why only this neural network

   Using the first 8 neurons of hidden layer 1 i.e 4 horizontal and 4 vertical boundaries we identify the regions intended. Next 4 neurons are to identify exact positive areas and output neuron to perform the or operation.



2. Implementation

MADALINE FAILED TO LEARN THIS FUNCTION FROM THE GIVEN INITIALIZATION

DATA - 100 samples (Train : 75 Test: 25 -> stratified splitting used)

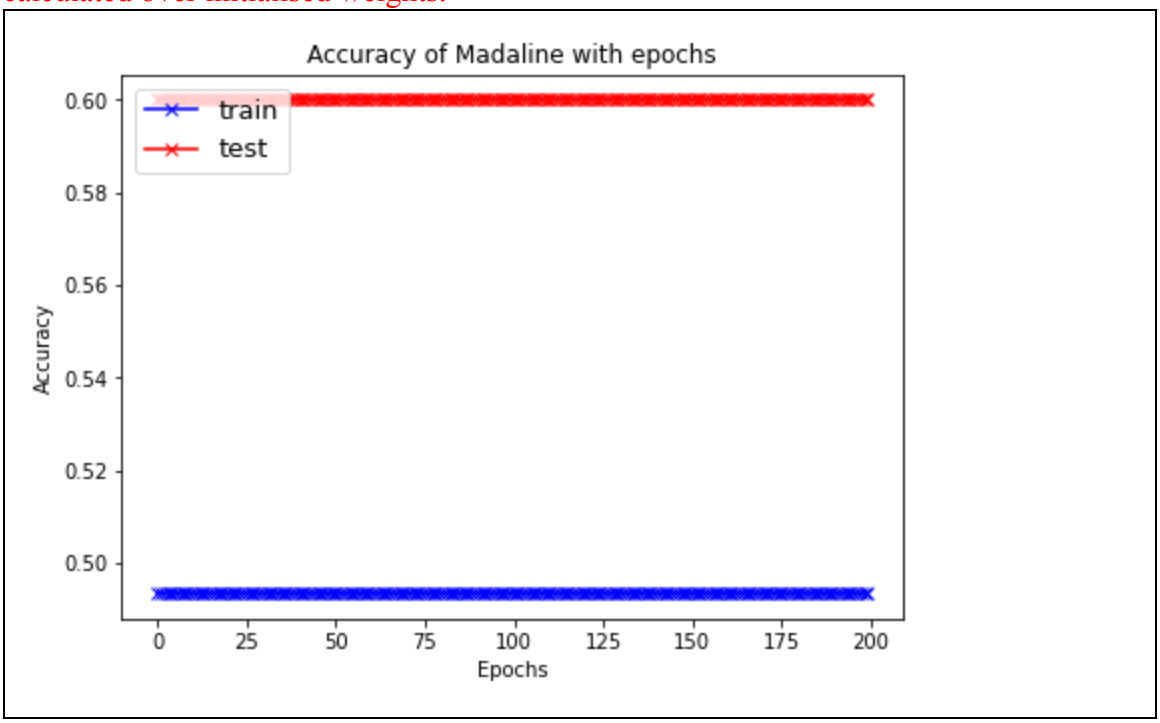
Epoc wise weight updated on each misclassified instance

(None of the weights were updated during complete 200 epocs)

```
---------------------------------------- 191 ----------------------------------------
Correctly classified
Incorrectly classified
Correctly classified
Incorrectly classified
Correctly classified
Correctly classified
Correctly classified
Correctly classified
Incorrectly classified
Correctly classified
Correctly classified
Incorrectly classified
Correctly classified
Correctly classified
Incorrectly classified
Incorrectly classified
Incorrectly classified
Correctly classified
Incorrectly classified
Incorrectly classified
Correctly classified
Incorrectly classified
Correctly classified
Incorrectly classified
Correctly classified
Correctly classified
Incorrectly classified
Incorrectly classified
Incorrectly classified
Correctly classified
Correctly classified
```

Epoc vs Accuracy

Note:- None of the weights got updated and this accuracy thus remains the same which is calculated over initialised weights.



4. Results and Accuracy

(Results are over the initial weights as madaline was unable to update weights)

TRAINING CLASSIFICATION REPORT

TRAIN

	precision	recall	f1-score	support
-1.0	1.00	0.06	0.12	63
1.0	0.52	1.00	0.68	63
accuracy			0.53	126
macro avg	0.76	0.53	0.40	126
weighted avg	0.76	0.53	0.40	126

TESTING CLASSIFICATION REPORT

TEST

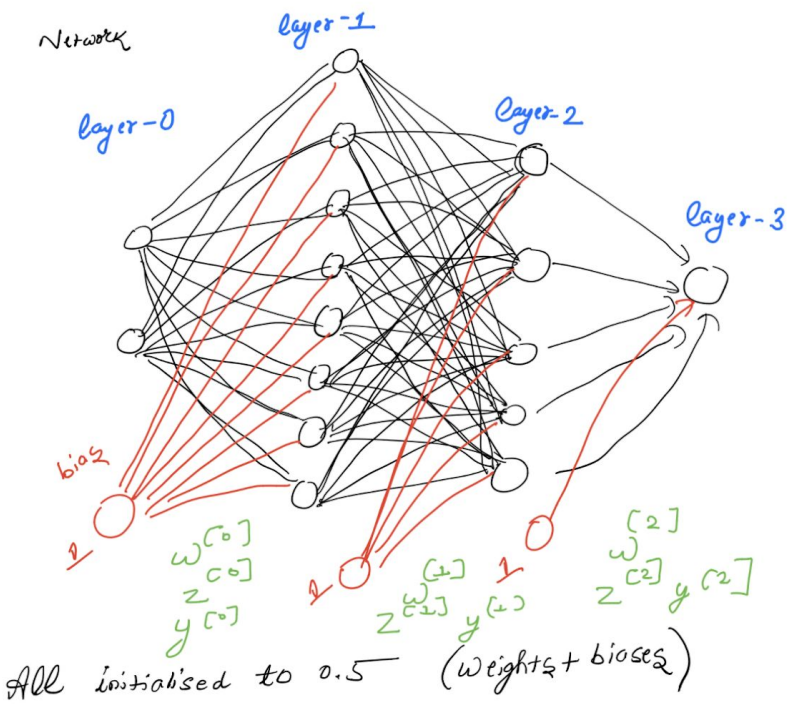
	precision	recall	f1-score	support
-1.0	1.00	0.19	0.32	21
1.0	0.55	1.00	0.71	21
accuracy			0.60	42
macro avg	0.78	0.60	0.52	42
weighted avg	0.78	0.60	0.52	42

3. Observation

- It is observed that weight initialization plays a very crucial role in weight updation using a madaline algorithm.
- Madaline was unable to learn the given function because it was complex and at any particular step flipping just one adeline unit did not make any change in the final output. This happened because multiple neurons together must have been flipped to obtain the correct output but as madaline does not consider multiple neurons together, It fails to learn this particular function.
- Thus the above results clearly show that the madaline algorithm is not successful in learning this function.
- Detailed Calculation are shown below.

CALCULATIONS

1. Network Explanation



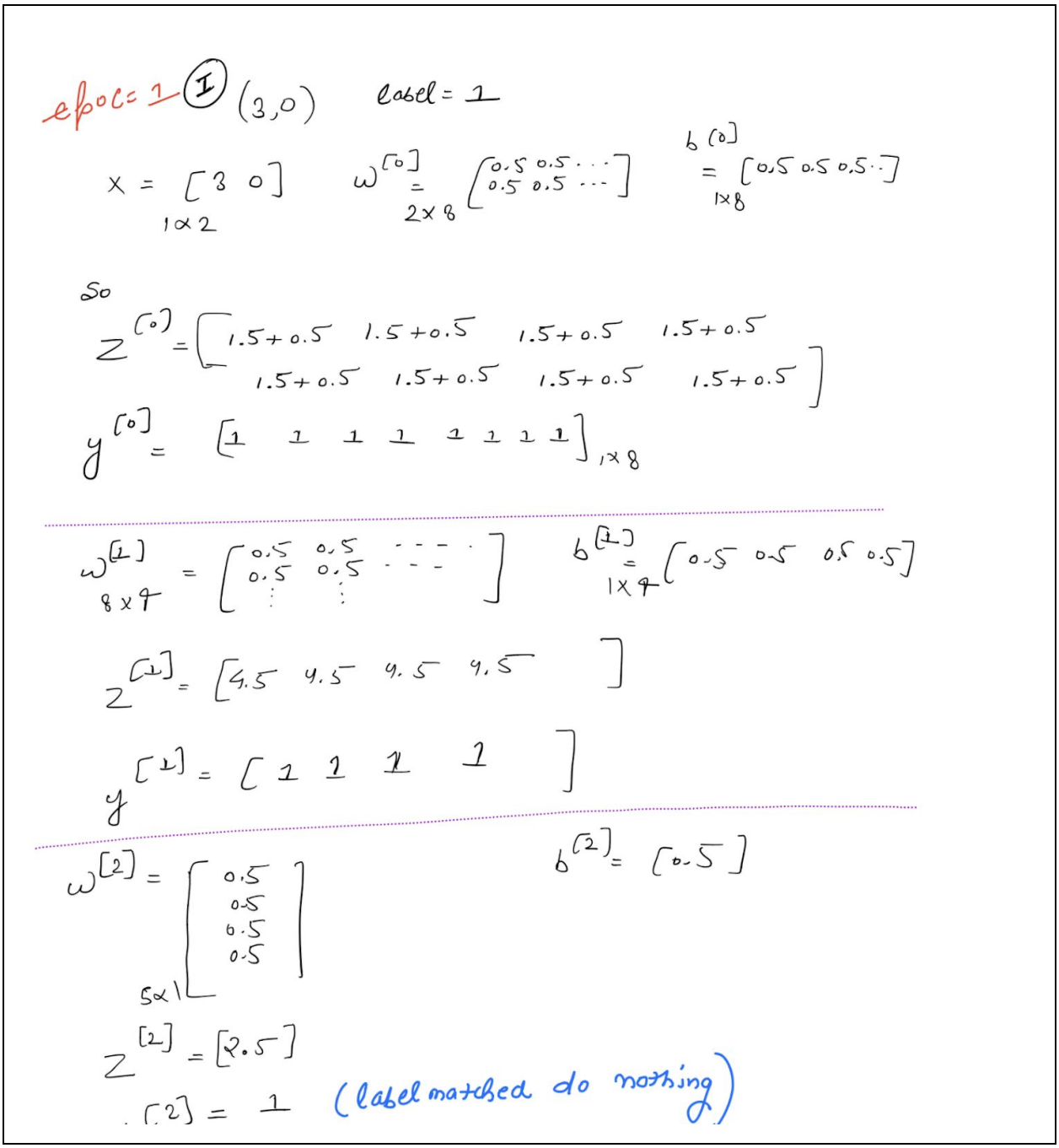
All initialised to 0.5 (weights + biases)

(1) initialization

$w^{[0]} = (2 \times 8)$
$b^{[0]} = (1 \times 8)$
$w^{[1]} = (8 \times 4)$
$b^{[1]} = (1 \times 4)$
$w^{[2]} = (4 \times 1)$
$b^{[2]} = (1 \times 1)$

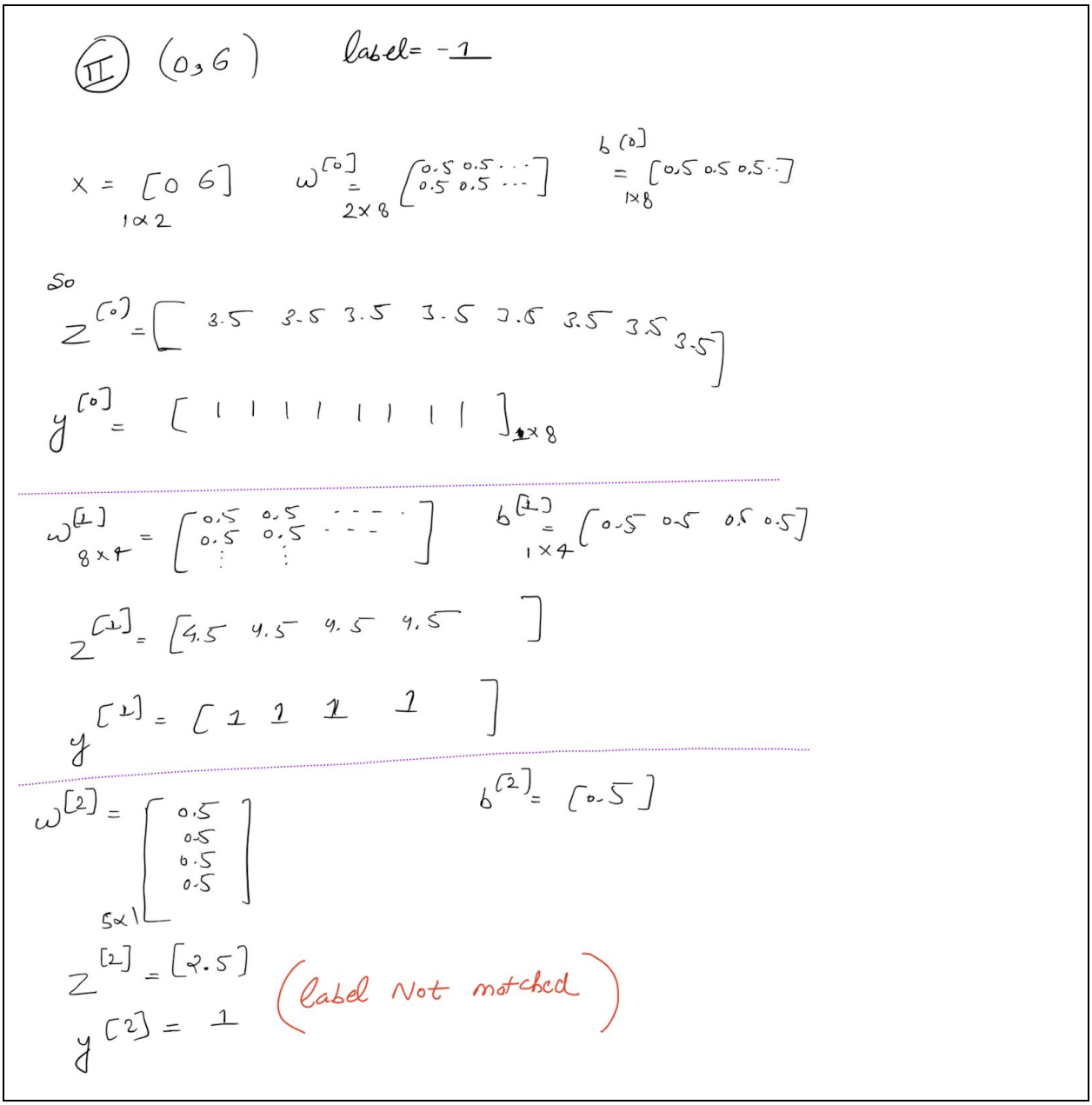
} all 0.5

2. Epoc-one (positive sample taken)
Weights are all initialized to 0.5 and the first positive sample taken which is correctly classified so no updates made in weights required.



3. Epoc-one (negative sample taken)

Here feedforward is done over a negative sample which is incorrectly classified, then all affine values flipped and checked that flipping any single adeline unit affine value does not make the classification correct. Further just by changing a few adeline units flips together , one could easily observe that the network was properly able to classify negative samples. But madaline does not incorporate, or allow updating multiple adeline units together. This fails to classify this particular function.



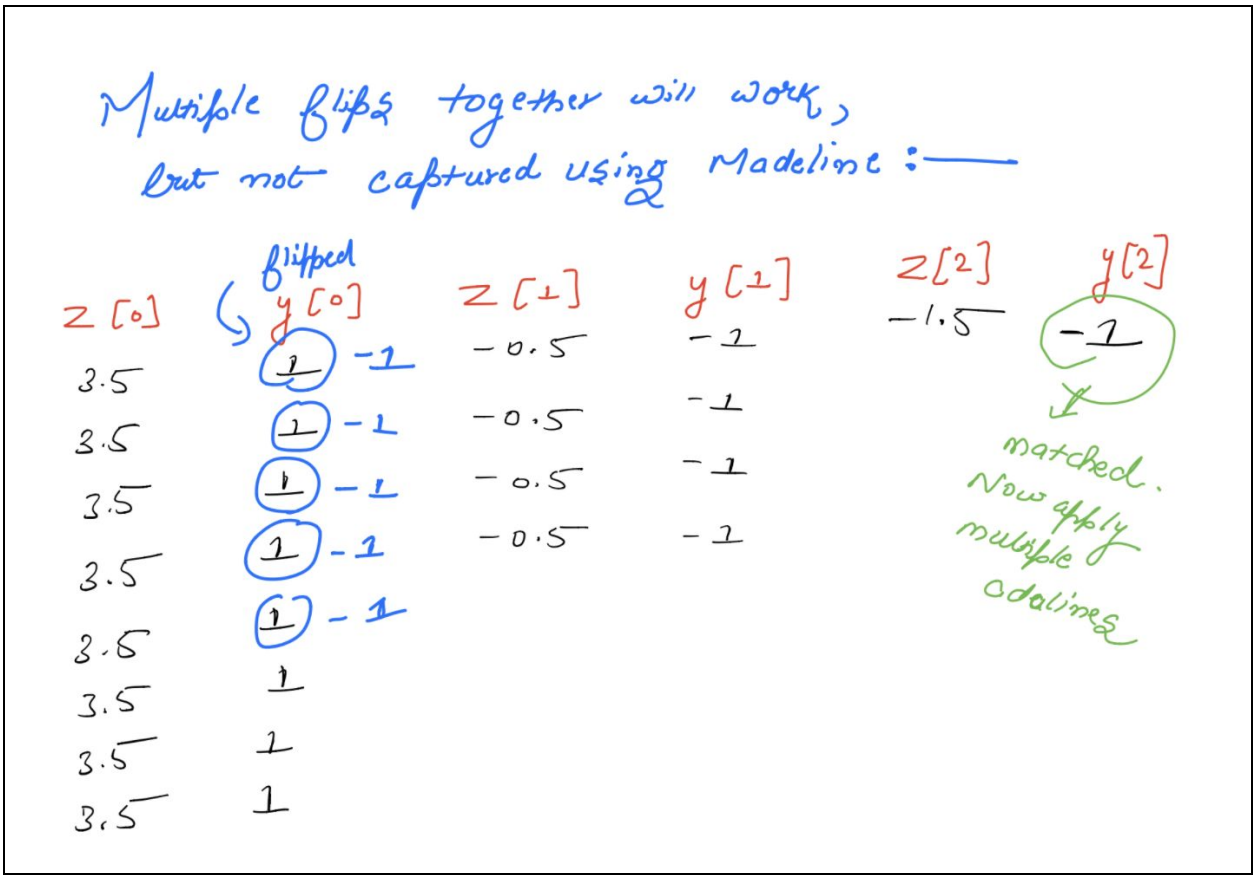
changing $Z^{(i)}$ values & checking

z[0]	y[0] (flipped)	z[1]	y[1]	z[2]	y[2]
3.5	1 → -1	3.5-0.5+0.5	1	3	1
3.5	1	3.5	+		
3.5	1	3.5	1		
3.5	1	3.5	1		
3.5	1		1		
3.5	1				
3.5	1				
3.5	1				
3.5	1				

Similarly flipping any one does not work in layer [0]

z[0]	y[0]	z[1]	y[1]	z[2]	y[2]
3.5	1	4.5	1 → -1	0.25	1
3.5	1	4.5	+		
3.5	1	4.5	1		
3.5	1	4.5	1		
3.5	1		1		
3.5	1				
3.5	1				
3.5	1				
3.5	1				

flipping any one does not work in layer [1]



CONCLUSION :-

1. Madaline is not able to learn complex functions because of the fact that it fails to update multiple adaline neurons together and considers only one at a time.
2. Madaline is highly affected by initial weights , that is one set of weights may help madaline converge while other may not even update any weights.
3. Madaline is easily able to converge for AND, OR, XOR in a few epochs as they are simpler functions.

3. Generalized Delta Rule

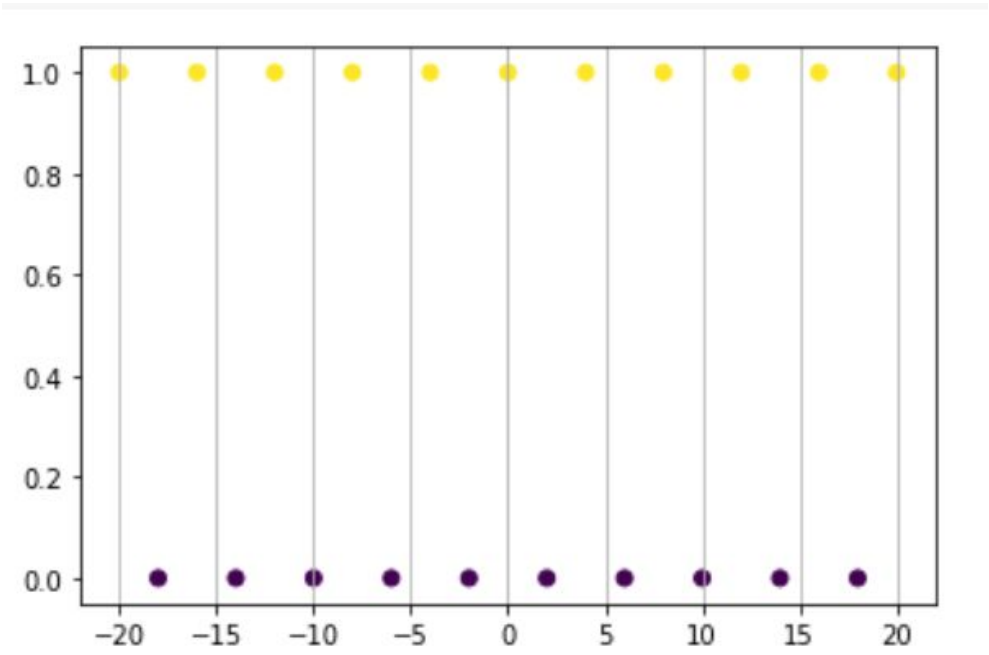
Methodology

The given in figure is a 1-D dataset with alternating labels 0,1. Similar dataset is generated using a helper function and we have changed the label from 0 to -1 as needed. Different activation functions are used to classify the dataset. These are Sigmoid, Tanh, Relu, Cosine, Sine. These are described below. The cosine and Sine curve are able to learn the function and have classified the dataset.

Assumptions

We have changed the label from 0 to -1 as tanh, cosine and sine have ranges from -1 to 1. Using this assumption we are able to classify the dataset completely using cosine or sine function.

Scatter plot of the dataset



1-D dataset with labels 0 and 1

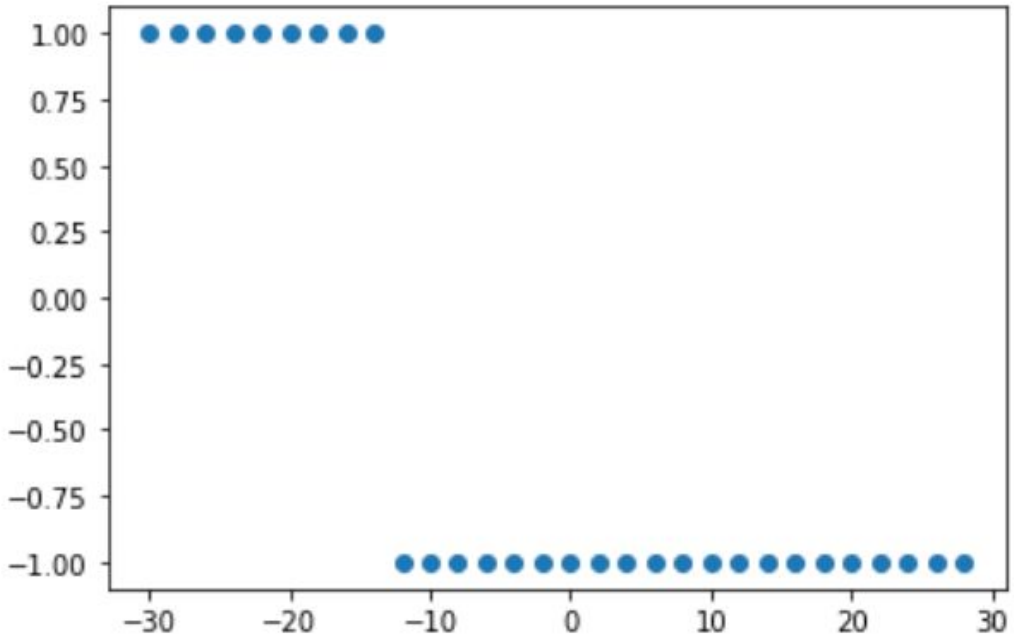
```
#Helper function to generate the required dataset
def generate_data():
  data = []
  for i in range(-20,21,2):
    if abs(i)%4 == 0:
      data.append([i,1, 1])

      data.append([i,1, -1])
```

```
return array(data).reshape(-1,3)
```

Observations

1. Sigmoid

```
Learnt weights are [[-0.02321425 -0.29468309]]
Train Accuracy 59.09090909090909
Test Accuracy 37.5
Dummy sample Accuracy 100.0
<matplotlib.collections.PathCollection at 0x7f2dc11b2358>
```



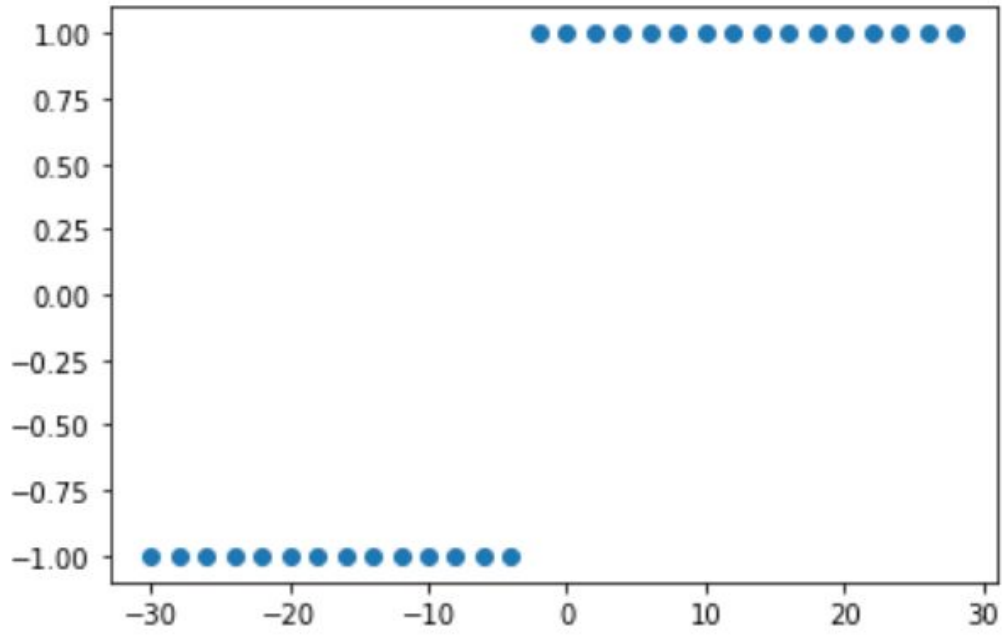
Sigmoid: Learnt Weights and Accuracy

f(x) = 1/(1+exp(-x))
f'(x) = f(x)(1-f(x))

Sigmoid function scales the value b/w 1 and 0. The given dataset cannot be classified completely using sigmoid as there is no threshold value after which the label is fixed. The label -1 is transformed to 0 before learning because sigmoid cannot generate less than 0 and so there will always be an error. Similarly during prediction as well 0 is transformed back to -1.

2. Tanh

```
Learnt weights are [[0.10434457 0.3381356 ]]
Train Accuracy 50.0
Test Accuracy 50.0
Dummy sample Accuracy 100.0
<matplotlib.collections.PathCollection at 0x7f2dc1558828>
```



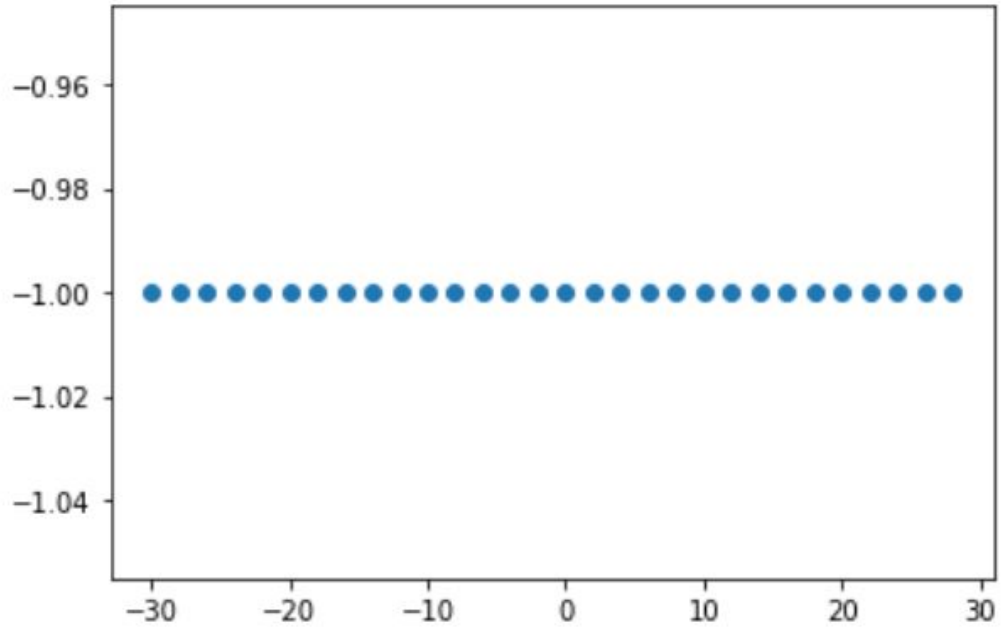
Tanh: Learnt Weights and Accuracy

f(x) = tanh(x)
f'(x) = 1 - tanh^2(x)

Tanh function scales the value b/w 1 and -1. The given dataset cannot be classified completely using tanh as there is no threshold value after which the label is fixed.

3. Relu

```
Learnt weights are [[ 0.15855732 -8.32313963]]
Train Accuracy 54.54545454545454
Test Accuracy 37.5
Dummy sample Accuracy 0.0
<matplotlib.collections.PathCollection at 0x7f2dc1292ba8>
```



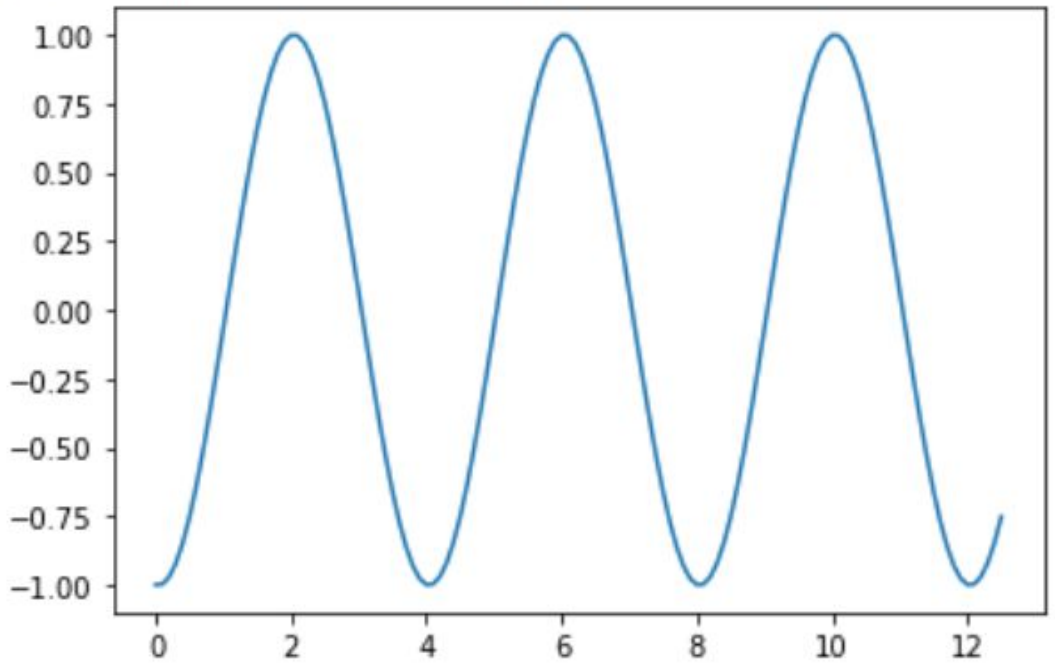
Relu: Learnt Weights and Accuracy

f(x) = x when x >=0 else 0
f'(x) = 1 when x >= 0 else 0

It seems Relu is also not a good activation function when the dataset is spread around the given pattern.

4. Cosine

```
Learnt weights are [[-1.57093784  0.06613848]]
Train Accuracy 100.0
Test Accuracy 100.0
Dummy sample Accuracy 100.0
```



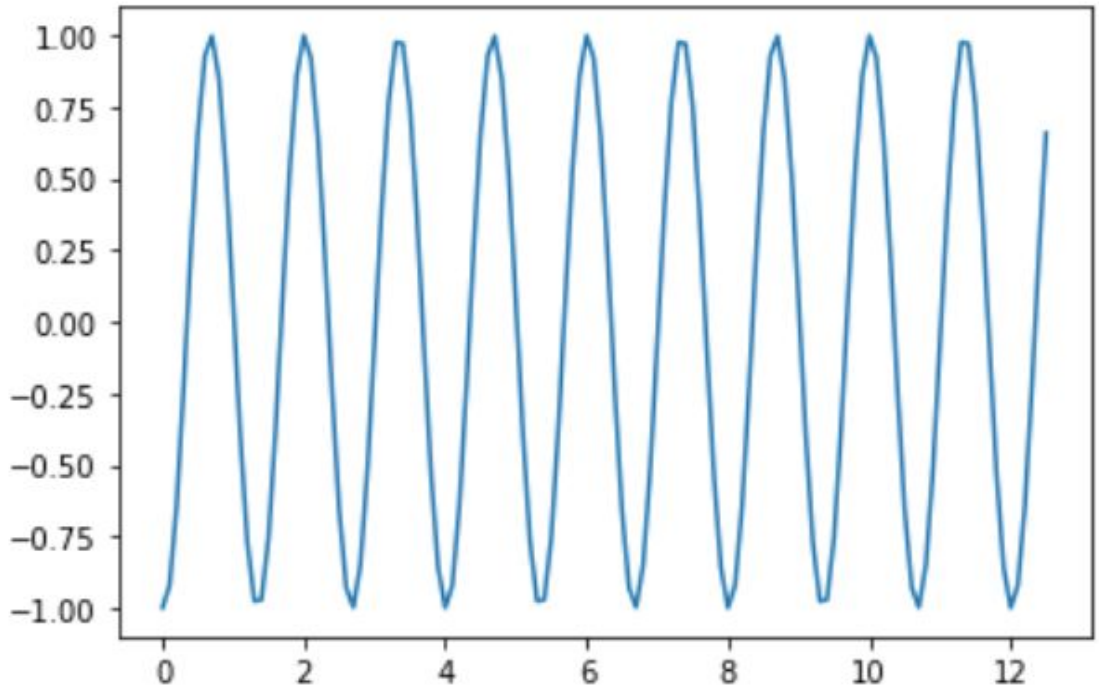
Cosine: Learnt Weights and Accuracy

f(x) = -cos(x)
f'(x) = sin(x)

With cosine as the activation function we are able to classify the dataset using a single neuron. The activation function is negative because we want the complement of the actual cosine curve.When the curve lies above the point the value is taken as negative and when the curve lies below the value is taken as positive. Bias term is around 0 therefore the learnt function curve is exactly the same as cosine (cos(0) = 1) but complemented because we have taken negative of the function.

5. Sine

```
Learnt weights are [[4.71253547 1.50175503]]
Train Accuracy 100.0
Test Accuracy 100.0
Dummy sample Accuracy 100.0
```



Sine: Learnt Weights and Accuracy

f(x) = -sin(x)
f'(x) = -cos(x)

With sine as the activation function we are able to classify the dataset using a single neuron. When the curve lies above the point the value is taken as negative and when the curve lies below the value is taken as positive. There is a bias learnt which is around pi/2 which moves the sine curve on the x axis.

CONTRIBUTION BY EACH MEMBER

1. Akanksha Shrimal
   Implemented complete madaline algorithm in Q2 along with all calculations and justifications. Brain stormed the network architecture and studied madaline algorithm to implement Q2.

2. Vaibhav Goswami
   Implemented perceptron training algorithm for AND, OR, NOT and XOR gate in Q1, proving convergence of the algorithm along with visualizing the change of boundary with each update.

3. Shivam Sharma
   Implemented generalized delta rule in question 3. Tried various activation functions like sigmoid, relu, tanh, sine and cosine. Plotted the dataset and predictions using the mentioned activation functions.