LAB-E: WRITE A PROGRAM TO REALISE AND GATE USING:

A. HEBBIAN NEURAL NETWORK.

B. PERCEPTRON NEURAL NETWORK.

A. HEBBIAN NEURAL NETWORK.

THEORY:

Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network. It is used for pattern classification. It is a single layer neural network, i.e. it has one input layer and one output layer. The input layer can have many units, say n. The output layer only has one unit. Hebbian rule works by updating the weights between neurons in the neural network for each training sample.

ALGORITHM:

1. Set all weights to zero, $w_i = 0$ for i=1 to n, and bias to zero.
2. For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.
3. Set activations for input units with the input vector $X_i = S_i$ for i = 1 to n.
4. Set the corresponding output value to the output neuron, i.e. y = t.
5. Update weight and bias by applying Hebb rule for all i = 1 to n:

$$w_i\,(new) = w_i\,(old) + x_i\,y$$

$$b\,(new) = b\,(old) + y$$

Implementing AND Gate :

INPUT				TARGET	
	x_1	x_2	b		y
X_1	-1	-1	1	Y_1	-1
X_2	-1	1	1	Y_2	-1
X_3	1	-1	1	Y_3	-1
X_4	1	1	1	Y_4	1

Truth Table of AND Gate using bipolar sigmoidal function

There are 4 training samples, so there will be 4 iterations. Also, the activation function used here is Bipolar Sigmoidal Function so the range is [-1,1].

Step 1 :

Set weight and bias to zero, $w = [\ 0\ 0\ 0\ ]^T$ and b = 0.

Step 2 :

Set input vector $X_i = S_i$ for i = 1 to 4.

$X_1 = [\ -1\ -1\ 1\ ]^T$

$X_2 = [\ -1\ 1\ 1\ ]^T$

$X_3 = [\ 1\ -1\ 1\ ]^T$

$X_4 = [\ 1\ 1\ 1\ ]^T$

Step 3 :

Output value is set to y = t.

Step 4 :

Modifying weights using Hebbian Rule:

First iteration –

$w(new) = w(old) + x_1y_1 = [\ 0\ 0\ 0\ ]^T + [\ -1\ -1\ 1\ ]^T . [\ -1\ ] = [\ 1\ 1\ -1\ ]^T$

For the second iteration, the final weight of the first one will be used and so on.

Second iteration –

$w(new) = [\ 1\ 1\ -1\ ]^T + [\ -1\ 1\ 1\ ]^T . [\ -1\ ] = [\ 2\ 0\ -2\ ]^T$

Third iteration –

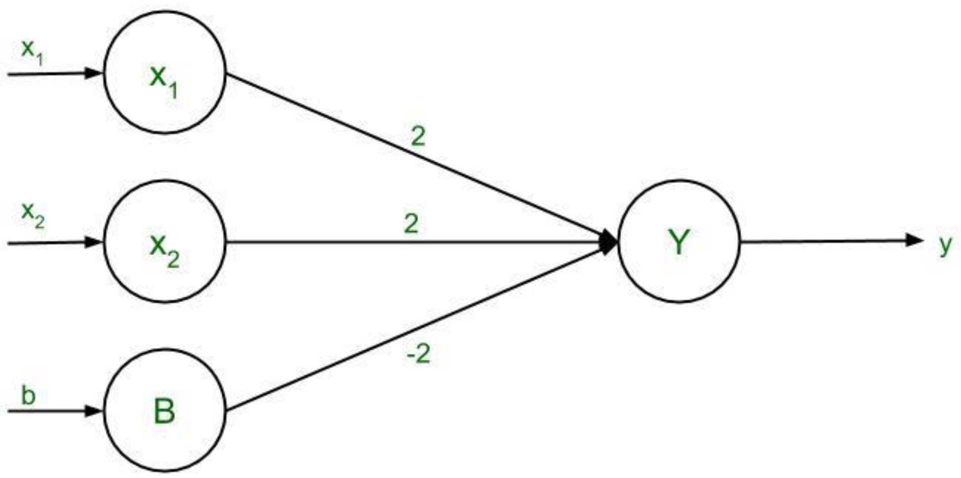
$w(new) = [\ 2\ 0\ -2]^T + [\ 1\ -1\ 1\ ]^T . [\ -1\ ] = [\ 1\ 1\ -3\ ]^T$

Fourth iteration –

$w(new) = [\ 1\ 1\ -3]^T + [\ 1\ 1\ 1\ ]^T . [\ 1\ ] = [\ 2\ 2\ -2\ ]^T$

So, the final weight matrix is $[\ 2\ 2\ -2\ ]^T$

Testing the network :



The network with the final weights

For x_1 = -1, x_2 = -1, b = 1, Y = (-1)(2) + (-1)(2) + (1)(-2) = -6

For x_1 = -1, x_2 = 1, b = 1, Y = (-1)(2) + (1)(2) + (1)(-2) = -2

For x_1 = 1, x_2 = -1, b = 1, Y = (1)(2) + (-1)(2) + (1)(-2) = -2

For x_1 = 1, x_2 = 1, b = 1, Y = (1)(2) + (1)(2) + (1)(-2) = 2

The results are all compatible with the original table.

Decision Boundary :

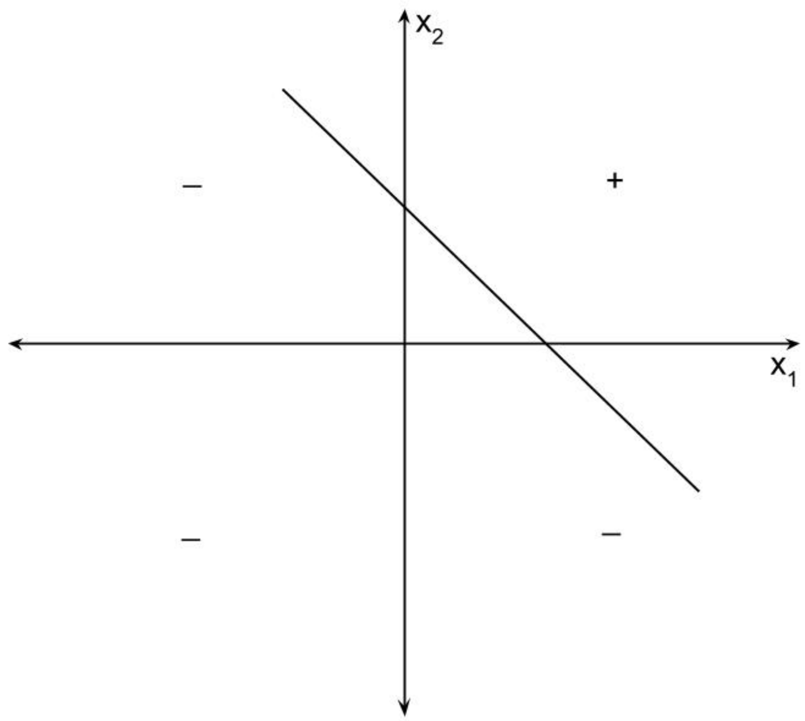
$2x_1 + 2x_2 – 2b = y$

Replacing y with 0, $2x_1 + 2x_2 – 2b = 0$

Since bias, b = 1, so $2x_1 + 2x_2 – 2(1) = 0$

$2(x_1 + x_2) = 2$

The final equation, $x_2 = -x_1 + 1$



PROGRAM:

import numpy as np

 # initial values

INPUTS = np.array([[1, 1], [1, -1], [-1, 1], [-1, -1]])

 # step function (activation function)

```
def step_function(sum):
if sum >= 0:
        return 1
return -1


# calculateing output
def calculate_output(weights, instance, bias):
        sum = instance.dot(weights)+ bias #dot product
        return step_function(sum)
def hebb(outputs, weights, bias):
        for i in range(4):
                weights[0] = weights[0] + (INPUTS[i][0] * outputs[i])
                weights[1] = weights[1] + (INPUTS[i][1] * outputs[i])
                bias = bias + (1 * outputs[i])
                print("Weight updated: " + str(weights[0]))
                print("Weight updated: " + str(weights[1]))
                print("Bias updated: " + str(bias))
                print("----------------------------------------")
                return weights, bias
if __name__ == "__main__":
        and_outputs = np.array([1, -1, -1, -1])
        weights = np.array([0.0, 0.0])
        bias = 0
        returned_weights, returned_bias = hebb(and_outputs, weights, bias)
        print('predicted output[1, 1]: ' + str(calculate_output(returned_weights,
        np.array([[1, 1]]), returned_bias)))
        print('predicted output [1, -1]: ' + str(calculate_output(returned_weights,
        np.array([[1, -1]]), returned_bias)))
        print('predicted output [-1, 1]: ' + str(calculate_output(returned_weights,
        np.array([[-1, 1]]), returned_bias)))
        print('predicted output [-1, -1]: ' + str(calculate_output(returned_weights,
        np.array([[-1, -1]]), returned_bias)))
```

OUTPUT:

```
source /home/tilak/anaconda3/bin/activate
(base) tilak@tilak:~/Desktop/ai (2)$ source /home/tilak/anaconda3/bin/activate
 /usr/bin/env /home/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launcher 44105 -- "/home/tilak/Desktop/ai (2)/hebbian.py"
conda activate orange3
(base) tilak@tilak:~/Desktop/ai (2)$  /usr/bin/env /home/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launcher 44105 -- "/home/tilak/Desktop/ai (2)/hebbian.py"
Weight updated: 1.0
Weight updated: 1.0
Bias updated: 1
----------------------------------------
Weight updated: 0.0
Weight updated: 2.0
Bias updated: 0
----------------------------------------
Weight updated: 1.0
Weight updated: 1.0
Bias updated: -1
----------------------------------------
Weight updated: 2.0
Weight updated: 2.0
Bias updated: -2
----------------------------------------
predicted output[1, 1]: 1
predicted output [1, -1]: -1
predicted output [-1, 1]: -1
predicted output [-1, -1]: -1
(base) tilak@tilak:~/Desktop/ai (2)$ conda activate orange3
(orange3) tilak@tilak:~/Desktop/ai (2)$
```

CONCLUSION:

Hence, we implemented AND gate using Hebbian Neural Net

B. PERCEPTRON NEURAL NETWORK.

OBJECTIVE:

THEORY:

Truth Table of AND Logical GATE is,

A	B	A ^ B
0	0	0
0	1	0
1	0	0
1	1	1

Weights w1 = 1.2, w2 = 0.6, Threshold = 1 and Learning Rate n = 0.5 are given

For Training Instance 1: A=0, B=0 and Target = 0

wi.xi = 0*1.2 + 0*0.6 = 0

This is not greater than the threshold of 1, so the output = 0, Here the target is same as calculated output.

For Training Instance 2: A=0, B=1 and Target = 0

wi.xi = 0*1.2 + 1*0.6 = 0.6

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

For Training Instance 2: A=1, B=0 and Target = 0

wi.xi = 1*1.2 + 0*0.6 = 1.2

This is greater than the threshold of 1, so the output = 1. Here the target does not match with the calculated output.

Hence we need to update the weights.

$$wi = wi + n(t - o)xi$$

$$w1 = 1.2 + 0.5(0 - 1)1 = 0.7$$

$$w2 = 0.6 + 0.5(0 - 1)0 = 0.6$$

Now,

After updating weights are w1 = 0.7, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

w1 = 0.7, w2 = 0.6 Threshold = 1 and Learning Rate n = 0.5

For Training Instance 1: A=0, B=0 and Target = 0

wi.xi = 0*0.7 + 0*0.6 = 0

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

For Training Instance 2: A=0, B=1 and Target = 0

wi.xi = 0*0.7 + 1*0.6 = 0.6

This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

For Training Instance 3: A=1, B=0 and Target = 0

wi.xi = 1*0.7 + 0*0.6 = 0.7

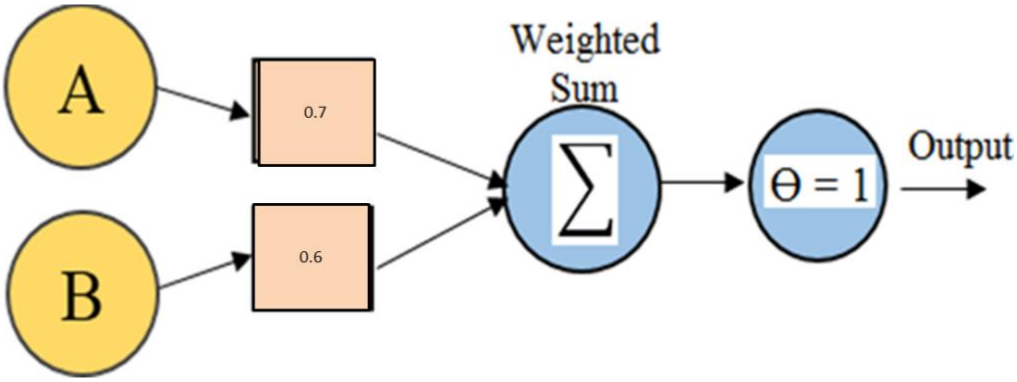
This is not greater than the threshold of 1, so the output = 0. Here the target is same as calculated output.

For Training Instance 4: A=1, B=1 and Target = 1

wi.xi = 1*0.7 + 1*0.6 = 1.3

This is greater than the threshold of 1, so the output = 1. Here the target is same as calculated output.

Hence the final weights are w1= 0.7 and w2 = 0.6, Threshold = 1 and Learning Rate n = 0.5.



ALGORITHM:

Our goal is to find the **w** vector that can perfectly classify positive inputs and negative inputs in our data. I will get straight to the algorithm. Here goes:

Algorithm: Perceptron Learning Algorithm

```
P ← inputs with label 1;
N ← inputs with label 0;
Initialize w randomly;
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P and w.x < 0 then
        w = w + x ;
    end
    if x ∈ N and w.x ≥ 0 then
        w = w − x ;
    end
end
//the algorithm converges when all the
 inputs are classified correctly
```

We initialize **w** with some random vector. We then iterate over all the examples in the data, (*P* U *N*) both positive and negative examples. Now if an input **x** belongs to *P*, ideally what should the dot product **w.x** be? I'd say greater than or equal to 0 because that's the only thing what our perceptron wants at the end of the day so let's give it that. And if **x** belongs to *N*, the dot product MUST be less than 0. So if you look at the if conditions in the while loop:

```
while !convergence do
    Pick random x ∈ P ∪ N ;
    if x ∈ P and w.x < 0 then
        w = w + x ;
    end
    if x ∈ N and w.x ≥ 0 then
        w = w − x ;
    end
end
```

Case 1: When **x** belongs to *P* and its dot product **w.x** < 0

Case 2: When **x** belongs to *N* and its dot product **w.x** ≥ 0

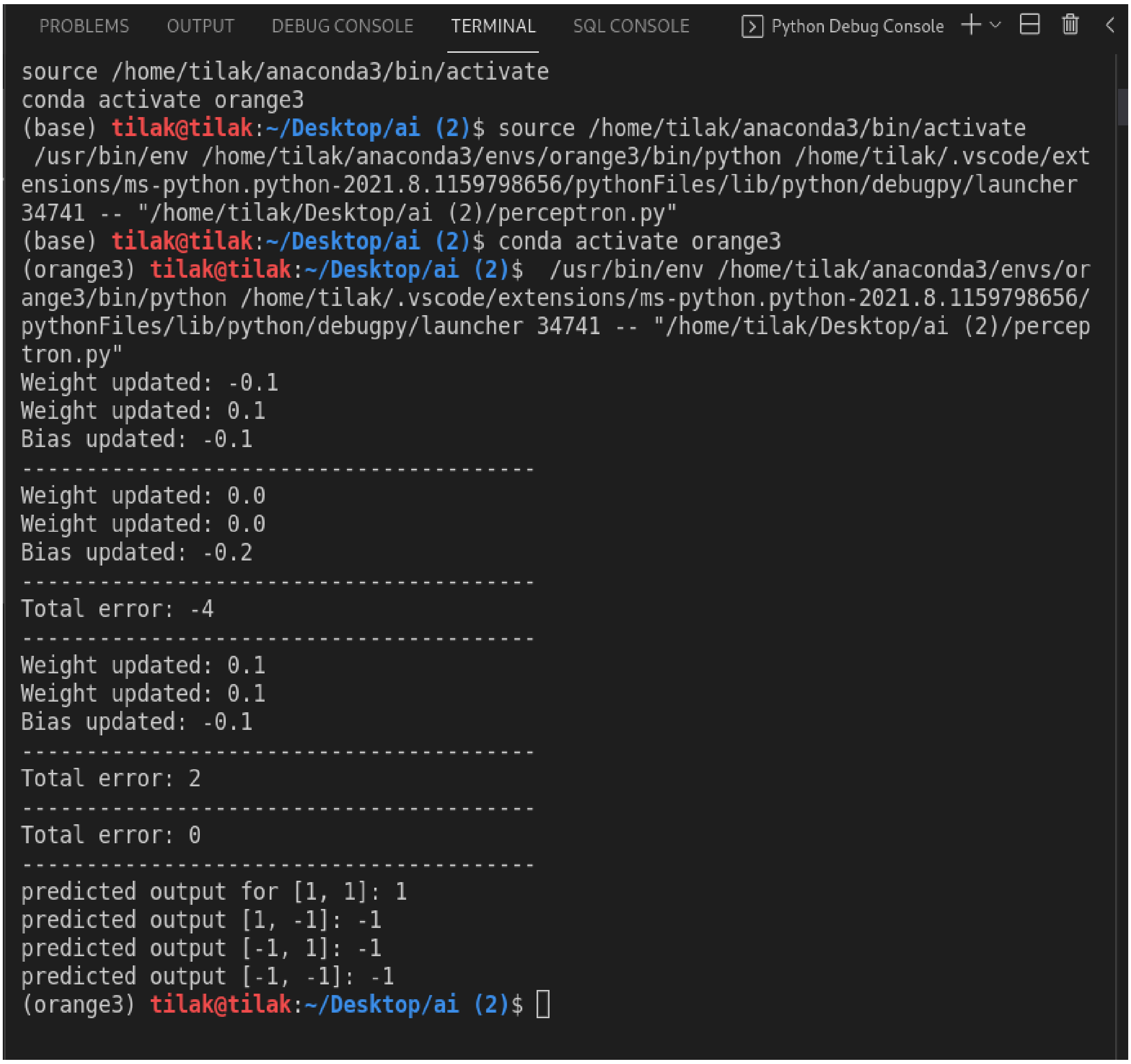
Only for these cases, we are updating our randomly initialized **w**. Otherwise, we don't touch **w** at all because Case 1 and Case 2 are violating the very rule of a perceptron. So we are adding **x** to **w** (ahem vector addition ahem) in Case 1 and subtracting **x** from **w** in Case 2.

PROGRAM:

```
import numpy as np

# initial values
INPUTS = np.array([[1, 1], [1, -1], [-1, 1], [-1, -1]])
LEARNING_RATE = 0.1


# step function (activation function)
def step_function(sum):
    if sum >= 0:
    return 1
    return -1



# calculateing output
def calculate_output(weights, instance, bias):
    sum = instance.dot(weights) + bias
    return step_function(sum)
def perceptron(outputs, weights, bias):
    total_error = 1
    counter = 0
    while total_error != 0 and counter < 10:


        total_error = 0
        counter += 1
        for i in range(len(outputs)):
            sum = INPUTS[i].dot(weights)
            prediction = step_function(sum + bias)


            total_error += outputs[i] - prediction


            if outputs[i] != prediction:
                weights[0] = weights[0] + (LEARNING_RATE * outputs[i] *
                INPUTS[i][0])
                weights[1] = weights[1] + (LEARNING_RATE * outputs[i] *
                INPUTS[i][1])
                bias = bias + (LEARNING_RATE * outputs[i])
                print("Weight updated: " + str(weights[0]))
                print("Weight updated: " + str(weights[1]))
                print("Bias updated: " + str(bias))
                print("----------------------------------------")


    print("Total error: " + str(total_error))
    print("----------------------------------------")


    return weights, bias
if __name__ == "__main__":
    and_outputs = np.array([1, -1, -1, -1])
    weights = np.array([0.0, 0.0])
    bias = 0
```

```
returned_weights, returned_bias = perceptron(and_outputs, weights, bias)
print('predicted output for [1, 1]: ' + str(calculate_output(returned_weights, np.array([[1, 1]]), returned_bias)))
print('predicted output [1, -1]: ' + str(calculate_output(returned_weights, np.array([[1, -1]]), returned_bias)))
print('predicted output [-1, 1]: ' + str(calculate_output(returned_weights, np.array([[-1, 1]]), returned_bias)))
print('predicted output [-1, -1]: ' + str(calculate_output(returned_weights, np.array([[-1, -1]]), returned_bias)))
```

OUTPUT:



CONCLUSION:

This we implemented the AND GATE Perceptron and Hebb Training Rule in Machine Learning successfully.