LAB EXERCISE – 7

Perceptron

1. Aim of the Experiment:

Implement and demonstrate perceptron model, a linear binary classifier used for supervised learning.

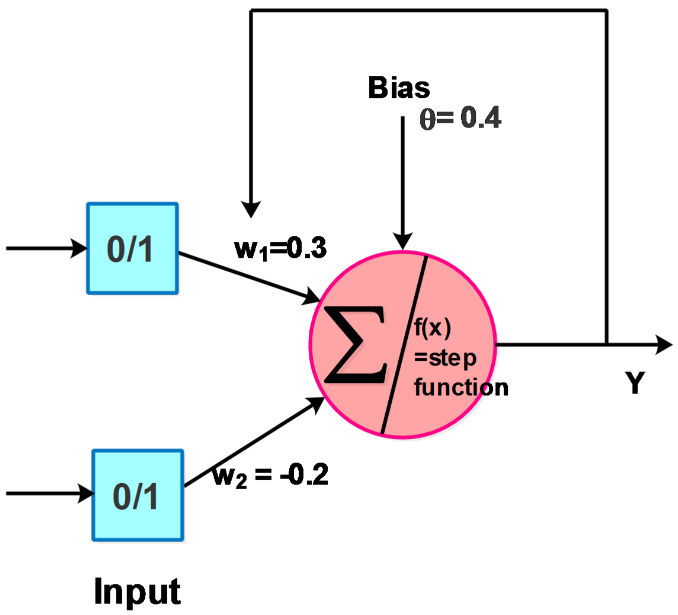


Figure 7: Perceptron for Boolean Function OR

Desired output for Boolean function OR is shown in Table 7.1.

Table 7.1: OR Truth Table

X_1	X_2	Ydes
0	0	0
0	1	1
1	0	1
1	1	1

Consider the perceptron to represent the Boolean function OR with the initial weights W_1 = 0.3, W_2 = -0.2, learning rate $\propto$ = 0.2 and bias W_0 = 0.4 as shown in Figure 7. The activation function used is the Step function f(x) which gives the output value as binary i.e., 0 or 1. If value of f(x) is greater than or equal to 0, it outputs 1 or else it outputs 0.

We design a perceptron that performs the Boolean function OR. The weights are updated until the Boolean function gives the desired output.

3. Python Program with Explanation:

1. Import numpy, array-processing package to work with the arrays.

```
import numpy as np
```

2. Create a Perceptron class to implement a perceptron network. Define the built-in __init__() function that takes learning rate of 0.2 and number of epochs of 4 to initialize the object. The initial weight vector is set as [0.3, -0.2].

```
class Perceptron(object):
    def __init__(self, input_size, lr=0.2, epochs=4):
        self.W = np.array([0.3,-0.2])
        self.epochs = epochs
        self.lr = lr
```

3. Define the activation function as Step function f(x) which gives the output value as binary i.e., 0 or 1. If value of f(x) is greater than or equal to 0, it outputs 1 or else it outputs 0.

```
    def activation_fn(self, x):
        return 1 if x >= 0 else 0
```

4. Define the predict function to compute the weighted sum ‘z’ by multiplying the inputs with the weights and add the products. Then subtract θ. Round the value to 2 decimals. Then call the activation function.

```
    def predict(self, x, theta):
        z = self.W.T.dot(x)-theta
        z=round(z,2)
        a = self.activation_fn(z)
         return a
```

5. Define the learning function fit() passing all inputs X, the desired output d, bias θ and count.

Update the weights for epochs, until the perceptron can correctly classify all inputs.

```
def fit(self, X, d,theta ,count):
    for _ in range(self.epochs):

        print("Epoch: ", count, "\n")
        count = count+1
        for i in range(d.shape[0]):
            x = X[i]
            print("input", x , "\t", "Weight:",self.W )
            print("\n")
```

Call the predict function, passing the input value x and theta. The function returns the predicted output value ‘y’.

```
            y = self.predict(x,theta)
```

Calculate error as the difference between the desired output d[i] and the predicted output y.

```
            e = d[i] – y
```

Update the weight vector.

```
            self.W = self.W + self.lr * e * x
```

6. Define the main function with input array X, desired output array d. This function is the entry point of the program.

```
if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
```

```
d = np.array([0, 1, 1, 1])
```

Create perceptron object. When the object is created, the __init__() function is called and the object is initialized.

```
perceptron = Perceptron(input_size=2)
theta=0.4
count =1
```

Call the learning function of the perceptron passing training input X, desired output d, theta and count.

```
perceptron.fit(X, d, theta, count)
```

Finally print the learned weights for the AND gate which gives the desired output.

```
print(perceptron.W)
```

Complete Program:

```
import numpy as np

class Perceptron(object):

    def __init__(self, input_size, lr=0.2, epochs=4):
        self.W = np.array([0.3,-0.2])
        self.epochs = epochs
        self.lr = lr

    def activation_fn(self, x):
        return 1 if x >= 0 else 0

    def predict(self, x,theta):
        z = self.W.T.dot(x)-theta
        z=round(z,2)
        a = self.activation_fn(z)
        return a

    def fit(self, X, d,theta ,count):
        for _ in range(self.epochs):
```

```
            print("Epoch: ", count)
            count = count+1
            for i in range(d.shape[0]):
                x = X[i]
                print("input", x , "\t", "Weight:",self.W )
                y = self.predict(x,theta)
                e = d[i] - y
                self.W = self.W + self.lr * e * x

if __name__ == '__main__':
    X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
    ])
    d = np.array([0, 0, 0, 1])

    perceptron = Perceptron(input_size=2)
    theta=0.4
    count =1
    perceptron.fit(X, d,theta, count)
    print(perceptron.W)
```

Output:

```
Python 3.8.3 (tags/v3.8.3:6f8c832, May 13 2020, 22:37:02) [MSC v.1924 64 bit (AMD64)] on win32
>>>
========== RESTART: C:\Users\ADMIN\pythonpgms\final\jnf perceptron.py ==========
Epoch:  1
input [0 0]     Weight: [ 0.3 -0.2]
input [0 1]     Weight: [ 0.3 -0.2]
input [1 0]     Weight: [ 0.3 -0.2]
input [1 1]     Weight: [ 0.3 -0.2]
Epoch:  2
```

input [0 0]   Weight: [ ]

input [0 1]   Weight: [ ]

input [1 0]   Weight: []

input [1 1]   Weight: []

Epoch: 3

input [0 0]   Weight: []

input [0 1]   Weight: []

input [1 0]   Weight: []

input [1 1]   Weight: []

Epoch: 4

input [0 0]   Weight: []

input [0 1]   Weight: []

input [1 0]   Weight: []

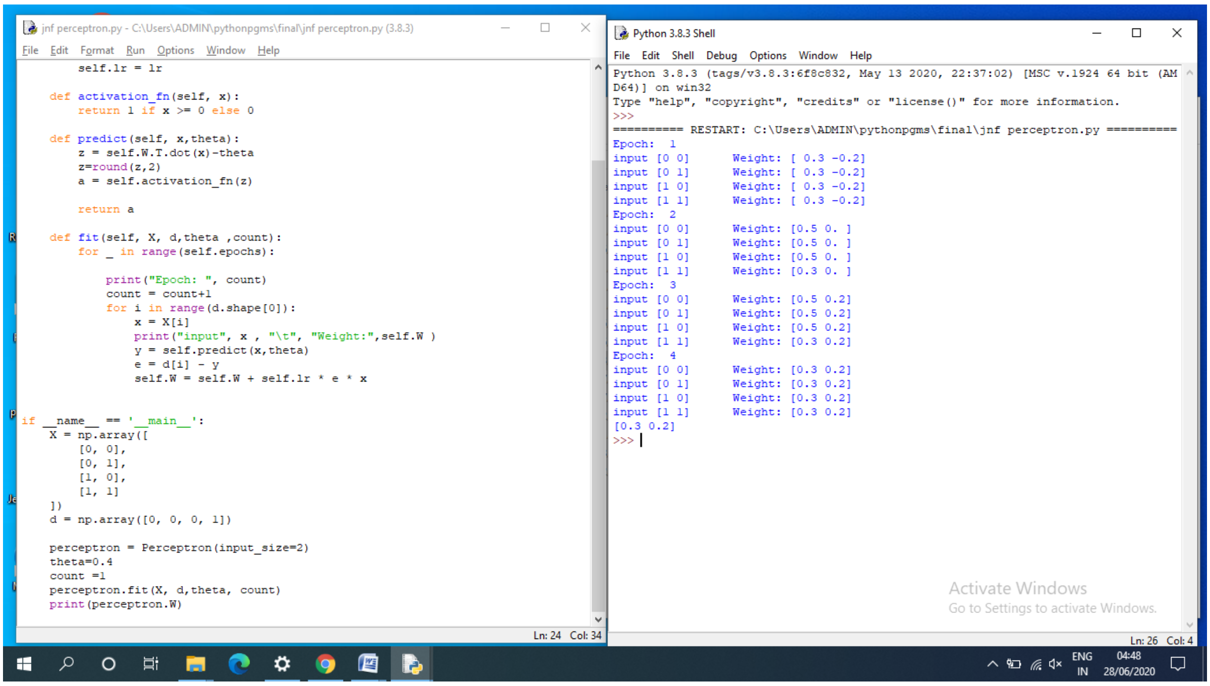
input [1 1]   Weight: []

[0.3 0.2]

>>>

It is observed that with 4 epochs, the perceptron learns, and the weights have been updated to new weights[ ] and with which the perceptron gives the desired output of a Boolean OR function.

Screenshot of the Output:



Programming Exercises:

1. Consider the perceptron taking two inputs x_1 and x_2 with weights $w_1 = 1.0$, $w_2 = 1.0$ and $w_0 = 1.5$. Determine the outputs for different combination of the inputs and plot them in graph x_1 vs. x_2.