LAB EXERCISE - 1

Perceptron (without using any ML/DL libraries)

1. Aim of the Experiment:

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

Listing 1: Figure 1 - Artificial Neural Networks

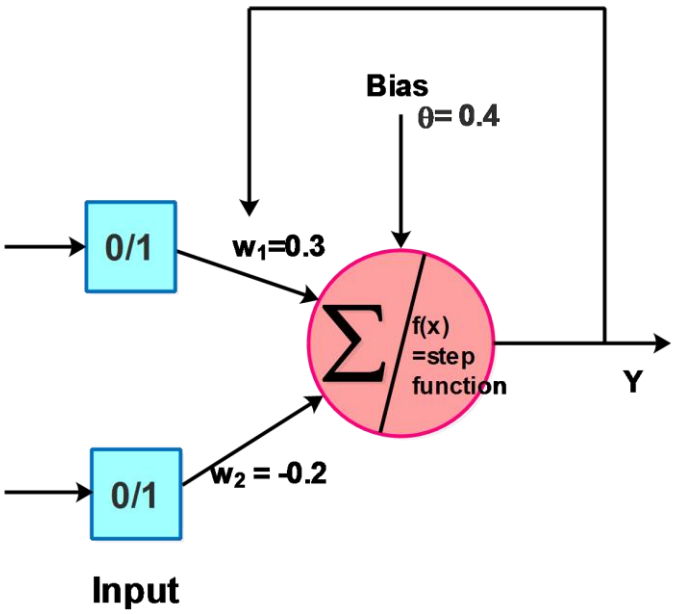


Figure 1: Perceptron for Boolean Function AND

Desired output for Boolean function AND is shown in Table 1

Table 1: AND Truth Table

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

Consider the perceptron to represent the Boolean function AND with the initial weights W_1 = 0.3, W_2 = -0.2, learning rate η = 0.2 and bias θ = 0.4 as shown in Figure 1. 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 AND. The weights are updated until the Boolean function gives the desired output.

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, 0, 0, 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)
```

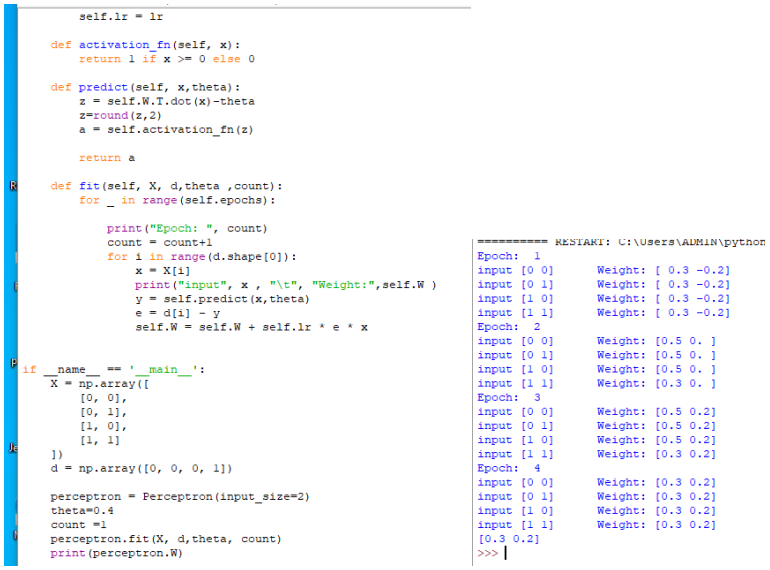
Output:

```
========== 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: [0.5 0. ]
input [0 1]     Weight: [0.5 0. ]
input [1 0]     Weight: [0.5 0. ]
input [1 1]     Weight: [0.3 0. ]
Epoch:  3
input [0 0]     Weight: [0.5 0.2]
input [0 1]     Weight: [0.5 0.2]
input [1 0]     Weight: [0.5 0.2]
input [1 1]     Weight: [0.3 0.2]
Epoch:  4
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]
[0.3 0.2]
>>>
```

It is observed that with 4 epochs, the perceptron learns, and the weights have been updated to 0.3 and 0.2 with which the perceptron gives the desired output of a Boolean AND function.

Screenshot of the Output:



```
        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)
```

```
========== RESTART: C:\Users\ADMIN\python
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: [0.5 0. ]
input [0 1]     Weight: [0.5 0. ]
input [1 0]     Weight: [0.5 0. ]
input [1 1]     Weight: [0.3 0. ]
Epoch:  3
input [0 0]     Weight: [0.5 0.2]
input [0 1]     Weight: [0.5 0.2]
input [1 0]     Weight: [0.5 0.2]
input [1 1]     Weight: [0.3 0.2]
Epoch:  4
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]
[0.3 0.2]
>>>
```

Perceptron implementation using SCIKIT LEARN:

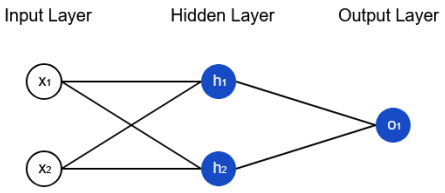
```
from sklearn.linear_model import Perceptron # for and logic
********************************************************************************
import numpy as np
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score

# 1. Define the input data (features) and target labels for the AND gate
# X: Input features (two inputs for AND gate)
# y: Target labels (output of AND gate)
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([0,
              0,
              0,
              1])

# 2. Create and train the Perceptron model
# Set a random_state for reproducibility and max_iter to ensure convergence
# eta0 is the learning rate, the default is 1.0
perceptron_model = Perceptron(random_state=42, max_iter=10, eta0=0.1)
perceptron_model.fit(X, y)

# 3. Evaluate the model on the training data
predictions = perceptron_model.predict(X)
print(f"Predictions: {predictions}")
print(f"Actual Labels: {y}")
accuracy = accuracy_score(y, predictions)
print(f"Accuracy: {accuracy * 100}%")

# 4. Access the learned weights and bias (optional)
# The coef_ attribute holds the weights, and intercept_ holds the bias
print(f"Learned weights (coef_): {perceptron_model.coef_}")
print(f"Learned bias (intercept_): {perceptron_model.intercept_}")
********************************************************************************************************
```

Perceptron on Or-, And- and Xor data

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt import seaborn as sns
or_data = pd.DataFrame() and_data = pd.DataFrame() xor_data =
pd.DataFrame()
or_data['input1']=[1,1,0,0]
or_data['input2']=[1,0,1,0]
or_data['ouput']=[1,1,1,0]
and_data['input1']=[1,1,0,0]
and_data['input2']=[1,0,1,0]
and_data['ouput']=[1,0,0,0]
xor_data['input1']=[1,1,0,0]
xor_data['input2']=[1,0,1,0]
xor_data['ouput']=[0,1,1,0]
from sklearn.linear_model import Perceptron clf1=Perceptron()
clf2=Perceptron() clf3=Perceptron()
clf1.fit(and_data.iloc[:,0:2].values,and_data.iloc[:,-1].values)
print(clf1.coef_)
print(clf1.intercept_) x=np.linspace(-1,1,5) y=-x+1
plt.plot(x,y)
#sns.scatterplot(and_data['input1'],and_data['input2'],hue=and_data['ouput'
],s=200) clf2.fit(or_data.iloc[:,0:2].values,or_data.iloc[:,-1].values)
print(clf2.coef_) print(clf2.intercept_) x1=np.linspace(-1,1,5) y1=-x+0.5
plt.plot(x1,y1)
#sns.scatterplot(or_data['input1'],or_data['input2'],hue=or_data['ouput'],s
=200) clf3.fit(xor_data.iloc[:,0:2].values,xor_data.iloc[:,-1].values)
print(clf3.coef_)
print(clf3.intercept_)
plot_decision_regions(xor_data.iloc[:,0:2].values,xor_data.iloc[:,-
1].values, clf=clf3, legend=2)
**********************************************************************
```

```
from sklearn.neural_network import MLPClassifier # for XOR
```

****




```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score

# 1. Define the XOR data
# Features (X): 4 input combinations for XOR
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Labels (y): Corresponding XOR outputs
y = np.array([0, 1, 1, 0])

# 2. Initialize the MLPClassifier
```

```
# We use one hidden layer with 2 neurons (the minimum required to solve
XOR)
# 'logistic' activation is commonly used for this problem.
# 'stochastic gradient descent' (sgd) is the optimizer.
# random_state for reproducibility.
mlp = MLPClassifier(hidden_layer_sizes=(2,),
                    activation='logistic',
                    solver='sgd',
                    learning_rate_init=0.1,
                    max_iter=10000,
                    random_state=1)

# 3. Train the model
# The model learns the complex, non-linear relationship
mlp.fit(X, y)

# 4. Make predictions
predictions = mlp.predict(X)

# 5. Evaluate the model
print(f"Predictions: {predictions}")
print(f"Actual Labels: {y}")
print(f"Accuracy: {accuracy_score(y, predictions) * 100:.2f}%")

# Test with new data (optional)
test_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
test_predictions = mlp.predict(test_data)
print(f"\nTest Predictions: {test_predictions}")
**************************************************************************
```