Lab 5: Implementation of Hebbian Learning and Perceptron

Hebbian Learning: Hebbian Learning is based upon Hebb's postulate of learning "If two interconnected neurons are both "on" at the same time, then the weight between them should be increased!" 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.

Learning Rule

Step 0: Initialize all weights to 0

Step 1: Given a training input, s, with its target output, t, set the activation of the input units: $x_i = s_i$

Step 2: Set the activation of the output unit to the target value: y = t

Step 3: Adjust the weights: w_i (new) = w_i (old) + x_i y

Step 4: Adjust the bias (just like the weights): b(new) = b(old) + y

In this lab, we will implement Hebbian Learning on a network, which works just like an AND function.

X_1	X_2	b	y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

Source Code:

```
import numpy as np

#Initialize x1, x2 and y
bias = [1, 1, 1, 1]
x1 = [1, 1, -1, -1]
x2 = [1, -1, 1, -1]
y = [1, -1, -1, -1]

#Initialize bias and weights
theta = [0, 0]
bias = 0

#Use Hebian Learning Rule
for i in range(0,len(x1)):
  theta[0] = theta[0] + x1[i]*y[i]
  theta[1] = theta[1] + x2[i]*y[i]
  bias = bias + y[i]

print("Final w0 = ",theta[0])
print("Final w1 = ",theta[1])
print("Final b = ",bias)
```

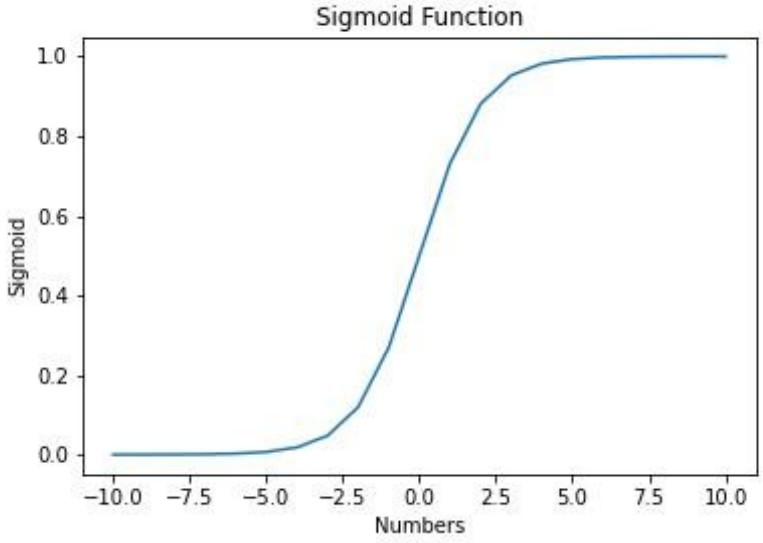
Output:

```
Final w0 = 2
Final w1 = 2
Final b = -2
```

```
def sigmoid(Z):
  return 1/(1 + np.exp(-Z))

numList = [i for i in range(-10, 11)]
numListSigmoid = [sigmoid(number) for number in numList]
import matplotlib.pyplot as plt
plt.figure()
plt.plot(numList, numListSigmoid)
plt.title("Sigmoid Function")
plt.xlabel("Numbers")
plt.ylabel("Sigmoid")
```

Output:




```
def predict(X,theta):
  Y = X[0]*theta[0] + X[1]*theta[1] + bias
  print(sigmoid(Y))
  if sigmoid(Y) >= 0.5:
    return 1
  else:
    return -1

X = np.array([1, -1])
print("The Prediction is ",predict(X,theta))
```

Output:

```
0.11920292202211755
The Prediction is -1
```

Perceptron: The perceptron was first introduced by American psychologist, Frank Rosenblatt in 1957 at Cornell Aeronautical Laboratory. Rosenblatt was heavily inspired by the biological neuron and its ability to learn. Rosenblatt's perceptron consists of one or more inputs, a processor, and only one output.
Perceptron is an algorithm for Supervised Learning of single layer binary linear classifier. Optimal weight coefficients are automatically learned. Weights are multiplied with the input features and decision is made if the neuron is fired or not.

Activation function applies a step rule to check if the output of the weighting function is greater than zero. Linear decision boundary is drawn enabling the distinction between the two linearly separable classes +1 and -1. If the sum of the input signals exceeds a certain threshold, it outputs a signal; otherwise, there is no output.

Algorithm

1. Set initial weights W_1, W_2,....W_n and bias to random numbers in the range [-0.5, 0.5]
2. Activate the perceptron by applying inputs and calculate the y_pred.

$$\text{y\_pred} = \text{step}\left(\sum_{i=1}^{n} X_i * W_i + b \right)$$

3. Update the weights of the perceptron as

$$W_i(p+1) = W_i(p) + \text{alpha} * X_i * e(p)$$

4. Increase iteration p by one, go back to step 2 and repeat until convergence.

In this lab, we will implement a Perceptron that will work just like an OR function.

X_1	X_2	b	y
1	1	1	1
1	-1	1	1
-1	1	1	1
-1	-1	1	-1

Source Code:

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

#Initialize X and y
X = np.array([[1, 1, 1],
      [1, 1, -1],
      [1, -1, 1],
      [1, -1, -1]])
y = np.array([1, 1, 1, -1])
print("Shape of X = ", X.shape)
print("Shape of y = ",y.shape)
```

Output:

```
Shape of X = (4, 3)
Shape of y = (4,)
```

```
#Initialize weights
theta = np.random.uniform(size=(3, 1))*0.5
print("Shape of theta = ",theta.shape)
print(theta)
```

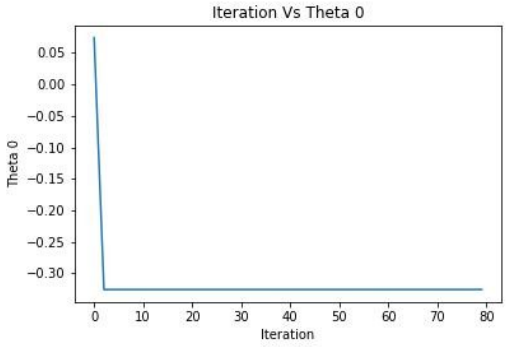
Output:

```
Shape of theta = (3, 1)
[[0.03366092] [0.25935474] [0.05914699]]
```

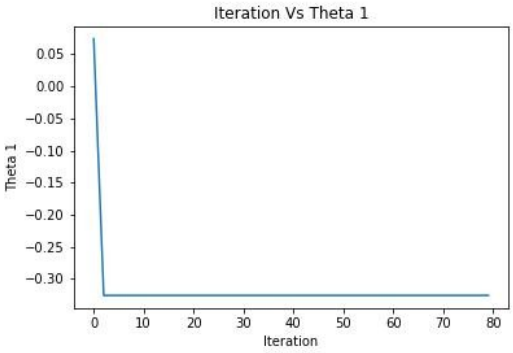
```
def step(Z):
  if Z>0:
    return 1
  else:
    return -1

alpha = 0.1
delta = 1
theta0 = []
theta1 = []
theta2 = []
for i in range(20):
  errors = []
  print("Iteration number : ",i)
  for i in range(0,X.shape[0]):
    y_pred = step(theta[0] + X[i][1]*theta[1] +X[i][2]*theta[2])
    error = y[i] - y_pred
    for j in range(3):
```

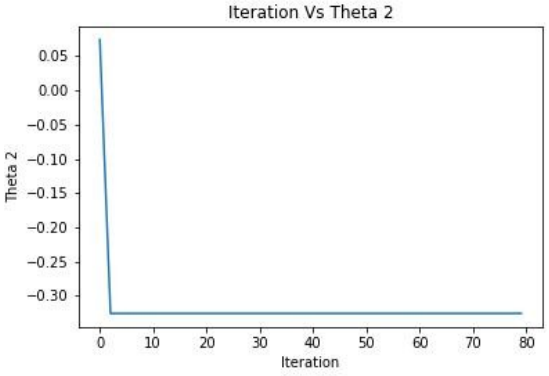
```
      theta[j] = theta[j] + alpha*error*X[i][j]
    theta0.append(theta[0][0])
    theta1.append(theta[1][0])
    theta2.append(theta[2][0])
```




```
import matplotlib.pyplot as plt
plt.figure()
plt.plot(theta0)
plt.title("Iteration Vs Theta 0")
plt.xlabel("Iteration")
plt.ylabel("Theta 0")
plt.savefig("Theta 0.jpg")
```




```
plt.figure()
plt.plot(theta0)
plt.title("Iteration Vs Theta 1")
plt.xlabel("Iteration")
plt.ylabel("Theta 1")
plt.savefig("Theta 1.jpg")
```




```
plt.figure()
plt.plot(theta0)
plt.title("Iteration Vs Theta 2")
plt.xlabel("Iteration")
plt.ylabel("Theta 2")
plt.savefig("Theta 2.jpg")
```

```
def predict(X,theta):
  return step(X[0][0]*theta[0] + X[0][1]*theta[1] + X[0][2]*theta[2])

X = np.array([[1, 1, -1]])
print("The Prediction is ",predict(X,theta))
```

Output:

```
The Prediction is 1
```