

| **Title: Implement Hebbian learning Rule** |
| --- |

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Objective:** To implement Hebbian learning with binary bipolar activation functions..

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Expected Outcome of Experiment:**

CO3: Understand perceptron’s and counter-propagation networks

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Books/ Journals/ Websites referred:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**Pre Lab/ Prior Concepts:**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Implementation Details:**

**Code:**

import numpy as np

*# Define the binary step function*

def binary\_step(x, threshold=0):

    return 1 if x >= threshold else 0

*# Define the bipolar step function*

def bipolar\_step(x, threshold=0):

    return 1 if x >= threshold else -1

*# Initialize weights and bias randomly*

np.random.seed(42)  *# For reproducibility*

weights = np.random.uniform(-1, 1, 3)

bias = np.random.uniform(-1, 1)

learning\_rate = 0.7

*# Define input vectors and expected outputs for AND gate*

inputs = np.array([

    [0, 0, 0],

    [0, 0, 1],

    [0, 1, 0],

    [0, 1, 1],

    [1, 0, 0],

    [1, 0, 1],

    [1, 1, 0],

    [1, 1, 1]

])

expected\_outputs\_binary = np.array([0, 0, 0, 0, 0, 0, 0, 1])  *# Binary AND*

expected\_outputs\_bipolar = np.array([-1, -1, -1, -1, -1, -1, -1, 1])  *# Bipolar AND*

*# Hebbian learning with binary step function*

print("Hebbian Learning with Binary Step Function")

for i, input\_vector in enumerate(inputs):

    net\_input = np.dot(weights, input\_vector) + bias

    output = binary\_step(net\_input, threshold=0.5)

    print(f"Input: {input\_vector}, Net input: {net\_input:.3f}, Output: {output}")

*# Update weights and bias*

    weights += learning\_rate \* input\_vector \* output

    bias += learning\_rate \* output

    print(f"Updated Weights: {weights}, Updated Bias: {bias}")

    print()

*# Reset weights and bias for bipolar step function*

weights = np.random.uniform(-1, 1, 3)

bias = np.random.uniform(-1, 1)

*# Hebbian learning with bipolar step function*

print("Hebbian Learning with Bipolar Step Function")

for i, input\_vector in enumerate(inputs):

    net\_input = np.dot(weights, input\_vector) + bias

    output = bipolar\_step(net\_input, threshold=0)

    print(f"Input: {input\_vector}, Net input: {net\_input:.3f}, Output: {output}")

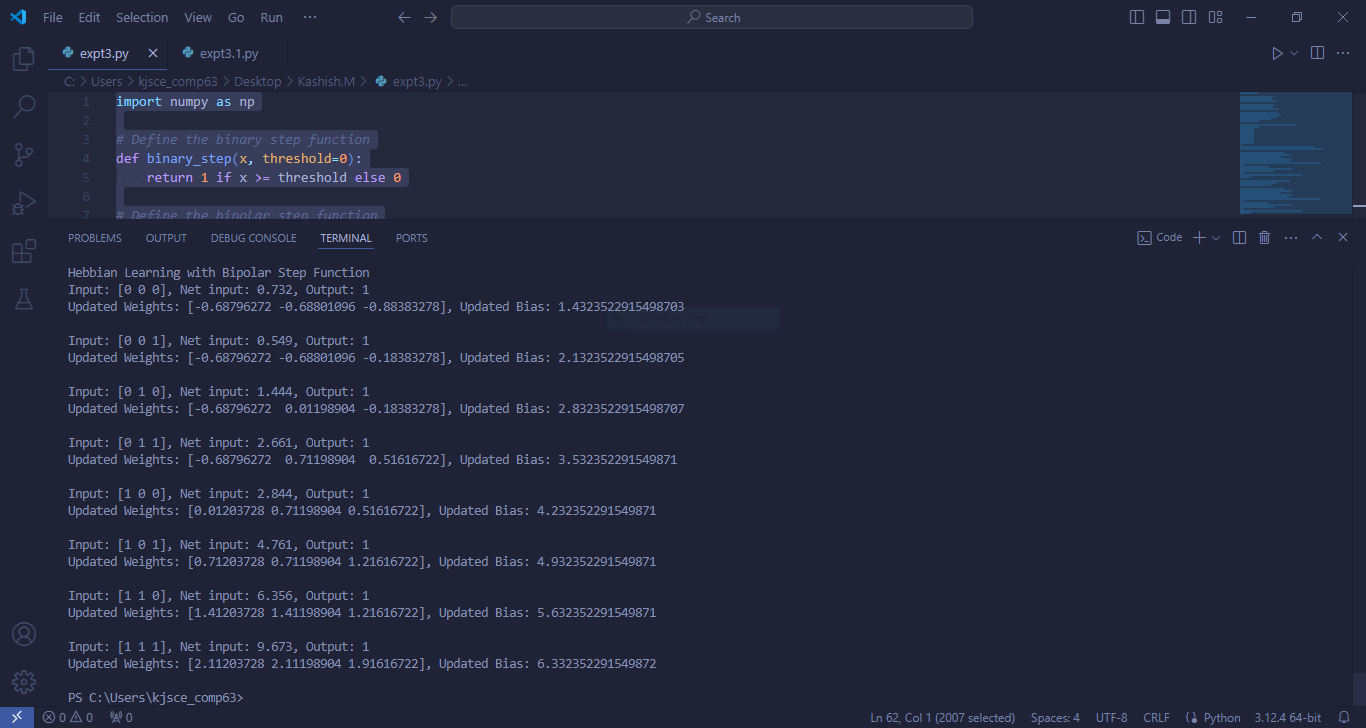
*# Update weights and bias*

    weights += learning\_rate \* input\_vector \* output

    bias += learning\_rate \* output

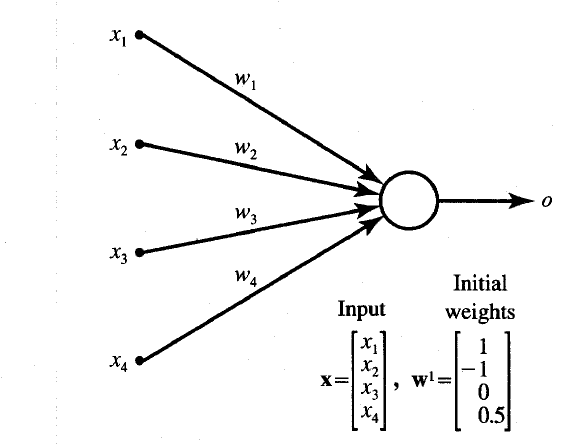
    print(f"Updated Weights: {weights}, Updated Bias: {bias}")

    print()



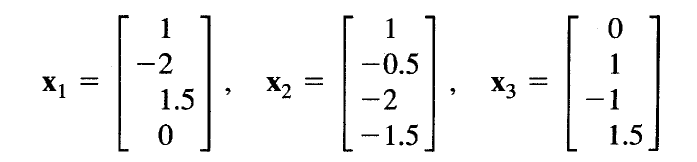
**#** Implement Hebbian learning for the given network

#Compare the learning using Binary and Bipolar activation functions.



**Network for implementation**

**Input vector:**

****

**Code:**

import numpy as np

*# Define the initial weights and the learning rate*

weights = np.array([1, -1, 0, 0.5])

learning\_rate = 1  *# You can set it to any value as needed*

*# Define the input vectors*

X1 = np.array([1, -2, 1.5, 0])

X2 = np.array([1, -0.5, -2, -1.5])

X3 = np.array([0, 1, -1, 1.5])

*# List of all input vectors*

inputs = [X1, X2, X3]

*# Hebbian learning process*

print("Initial Weights:", weights)

for i, input\_vector in enumerate(inputs):

*# Hebbian learning update rule: Δw = η \* x \* y*

*# Assuming the output y = x for simplicity in Hebbian learning*

    output = input\_vector  *# For Hebbian, output = input*

    delta\_w = learning\_rate \* output \* output  *# Δw = η \* x \* y (y = x)*

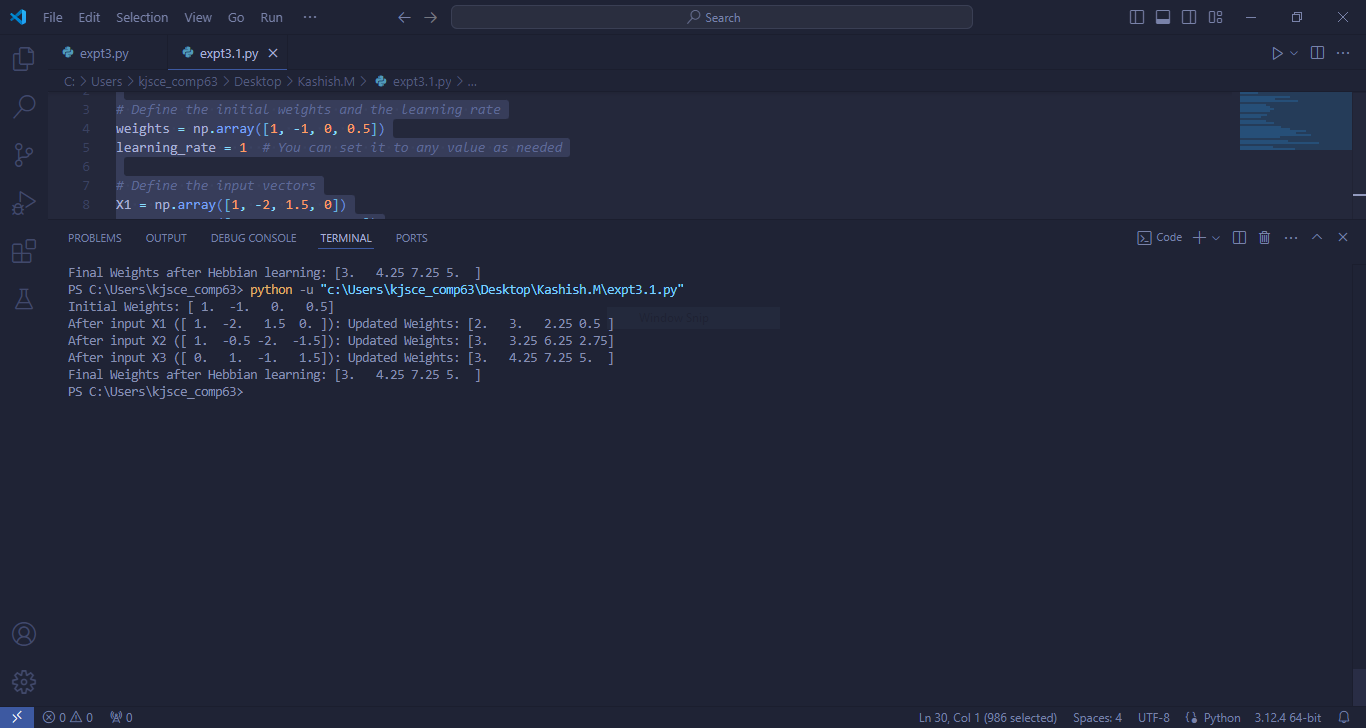
    weights += delta\_w  *# Update weights*

*# Output the updated weights after each step*

    print(f"After input X{i + 1} ({input\_vector}): Updated Weights: {weights}")

*# Final weights after all updates*

print("Final Weights after Hebbian learning:", weights)



**Conclusion:** *#write the conclusion w.r.t the comparison between the Hebbian learning using Binary and Bipolar activation function*

**Thus, we have successfully implemented Hebbian learning algorithm of Neural Network.**

**Post Lab Descriptive Questions :**

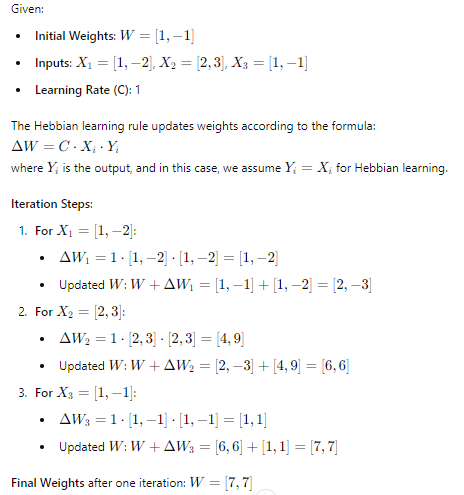
1. Compare the Hebbian learning and competitive learning.

**Ans:**

| **Aspect** | **Hebbian Learning** | **Competitive Learning** |
| --- | --- | --- |
| **Basic Concept** | Strengthens connections between simultaneously activated neurons. | Neurons compete to become the sole "winner" for an input. Only the winning neuron updates its weights. |
| **Weight Update Rule** | All active neurons update their weights based on the correlation between input and output. | Only the winning neuron's weights are adjusted; others remain unchanged. |
| **Activation Function** | Often uses binary or bipolar activation functions. | Typically uses an activation function to determine the winner. |
| **Specialization** | Does not inherently specialize neurons; reinforces all frequently occurring patterns. | Specializes neurons to distinct input patterns, often leading to partitioning of the input space. |
| **Output** | Correlation reinforcement; patterns are strengthened. | Neurons develop specificity to different input patterns. |
| **Convergence** | May reinforce all patterns without competition, which can lead to redundancy. | Tends to converge to a state where each neuron specializes in different aspects of the input space. |
| **Applications** | Used in associative memory, feature detection, auto-associative networks. | Common in clustering algorithms like Self-Organizing Maps (SOMs) and feature maps. |
| **Learning Mechanism** | Unsupervised, correlation-based. | Unsupervised, competition-based. |
| **Output Representation** | Can represent patterns but may not distinguish as distinctly as competitive learning. | Can distinctly represent and separate patterns due to neuron competition. |
| **Network Behavior** | All neurons can potentially learn from the same inputs. | Neurons compete, leading to a unique representation per neuron. |
| **Example Use Cases** | Pattern recognition, associative memory tasks. | Clustering, categorization tasks like in self-organizing maps. |

1. Find the weights after one iteration for hebbian learning of a single neuron network. Start with initial weights W=[1,-1] and Inputs as X1=[1,-2] X2=[2,3], x3[1,-1] and C=1.

**Ans:**



.

**Date: 06/08/2024 Signature of faculty in-charge**