K. J. Somaiya College of Engineering, Mumbai-77

Batch: B2 Roll No.: 16010121194

Experiment No. 05

Grade: A / AB / BB / BC / CC / CD /DD

Signature of the Staff In-charge with date

Title: To Study and implement HEBBS learning rule

Objective: To write a program to implement HEBBS learning rule for the given data.

Expected Outcome of Experiment:

CO3: Understand perceptron's and counter propagation networks

Books/ Journals/ Websites referred:

Pre Lab/ Prior Concepts:

Hebbian learning rule and algorithm

Hebbian learning is a fundamental concept in neuroscience and artificial intelligence that explains how neural connections strengthen through experience. It's named after the Canadian psychologist Donald Hebb, who proposed this idea in his book "The Organization of Behavior" in 1949. Hebbian learning is often summarized by the phrase: "Cells that fire together, wire together."

The Hebbian learning rule is a simple and biologically inspired learning principle that suggests that when two neurons on either side of a synapse are activated simultaneously, the strength of that synapse should be increased. Conversely, if they are not activated together, the synaptic

connection should weaken. This rule helps to explain how neural networks can adapt and learn from their inputs.

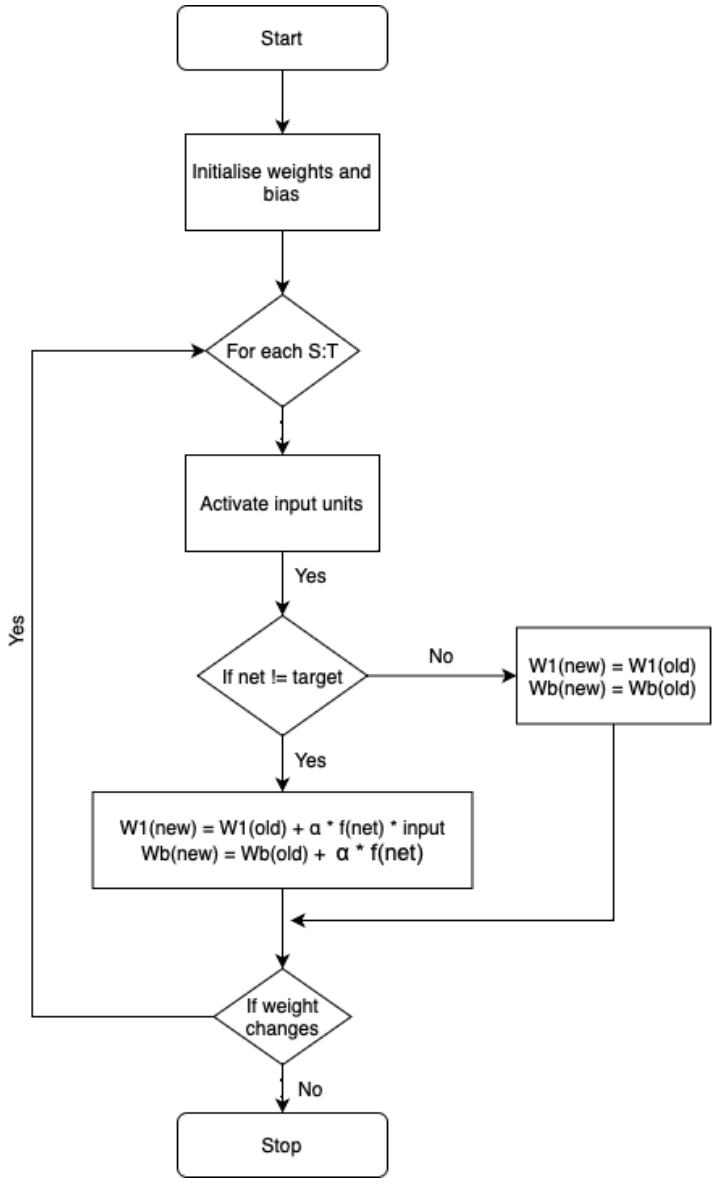
The basic Hebbian learning algorithm can be described as follows:

1. Start with an initial set of synaptic weights, typically small random values.
2. Present a pattern of input to the network.
3. If the input neurons are active and cause the output neuron to fire, increase the strength of the synapses connecting those neurons.
4. If the input neurons are active, but the output neuron does not fire, leave the synaptic weights unchanged.
5. If the input neurons are not active, do nothing to the synapses.
6. Repeat steps 2-5 for many input patterns and iterations.

Over time, this simple rule causes synapses to strengthen for patterns that are frequently encountered, leading to the network's ability to recognize and respond to specific input patterns. However, it's important to note that the basic Hebbian learning rule has limitations, such as the potential for unstable learning and a lack of error correction.

In practice, more sophisticated learning algorithms, like the delta rule or backpropagation, are often used in artificial neural networks. These algorithms provide a more stable and efficient way to adjust synaptic weights and achieve specific learning objectives, including error minimization. Nonetheless, Hebbian learning remains a foundational concept in understanding how the brain processes and learns from information.

Flowchart:



Implementation Details:

```
import math

#libraries used only for data visualization:
import numpy as np
from tabulate import tabulate

#inputs given in the question
```

```
inputs = [
            [1, -2, 1.5, 0],
            [1, -0.5, -2, -1.5],
            [0, 1, -1, 1.5]
          ]

#choosing learning rate to be 1
learning_rate = 1

#initial weights given in the question
weights = [1, -1, 0, 0.5]

#function defined for calculating activation function
#using bipolar sigmoidal function
def activation_function(net):
    return (1 - math.exp(-net)) / (1 + math.exp(-net))

#creating loop for iteration for each input array
#thus, there will be 3 epochs, because there are 3 input arrays

for j in range(len(inputs)):

    #initializing the value of net to be 0 before every
    #input array's calculations are done
    net = 0

    print("\n\n------------------Epoch ", (j + 1), "------------------")

    #taking dot product of weights array and inputs array to find the value of net
    for i in range(len(weights)):
        net = net + weights[i] * inputs[j][i]

    #actual output will be the activation function applied on the value of fnet
    fnet = activation_function(net)


    #making a variable for storing the value of old weights for data visualization
    #using the round function that comes with numpy to show only the first 5 digits
after the decimal
    old = list(np.round(weights, 5))

    #declaring a list to store the value of change in weights
```

```
    delta_w = []

    #for loop for calculating the value of new weights
    #purpose of for loop is matrix multiplication
    for k in range(len(weights)):

        #storing the value of change in weight given by learning_rate * output *
inputs[j][k]
        #for data visualization
        change_in_weight = learning_rate * fnet * inputs[j][k]
        delta_w.append(round(change_in_weight , 5))
        weights[k] += change_in_weight

    #making a variable for storing the value of new weights after this epoch for data
visualization
    new_weights = list(np.round(weights, 3))

    #making a list of numpy arrays to pass in the tabulate function
    #as passing numpy arrays directly to tabulate gives warnings and causes conflicts
    table_data = [old, new_weights, delta_w, fnet]

    #using tabulate functions for data visualization
    headers = ["Old Weights", "New Weights", "Change in Weight", "FNet"]
    table = tabulate([table_data], headers=headers, tablefmt="fancy_grid")
    print(table)
```

Output:

```
------------------Epoch  1 ------------------
```

Old Weights	New Weights	Change in Weight	FNet
[1.0, -1.0, 0.0, 0.5]	[1.905, -2.81, 1.358, 0.5]	[0.90515, -1.8103, 1.35772, 0.0]	0.905148

```
------------------Epoch  2 ------------------
```

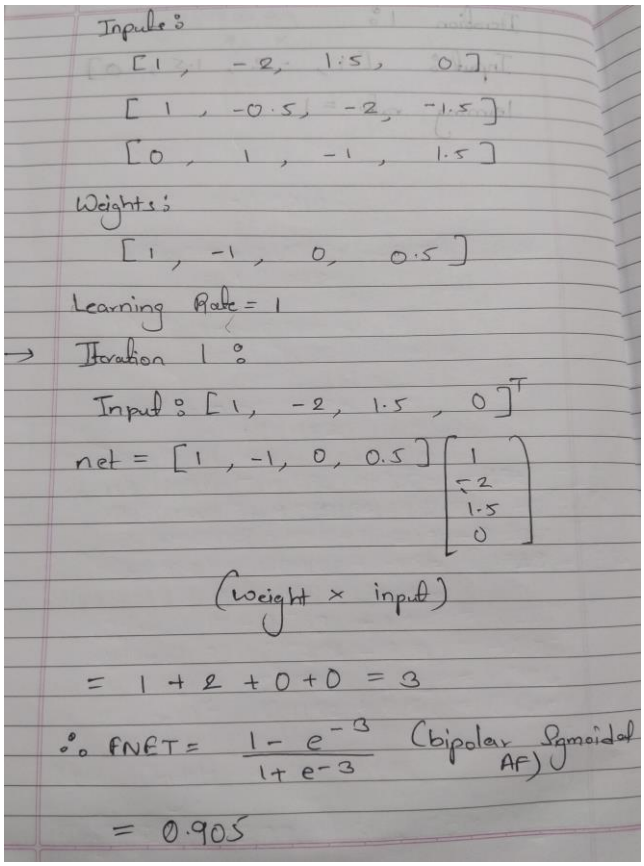
Old Weights	New Weights	Change in Weight	FNet
[1.90515, -2.8103, 1.35772, 0.5]	[1.828, -2.772, 1.513, 0.616]	[-0.07742, 0.03871, 0.15484, 0.11613]	-0.0774189

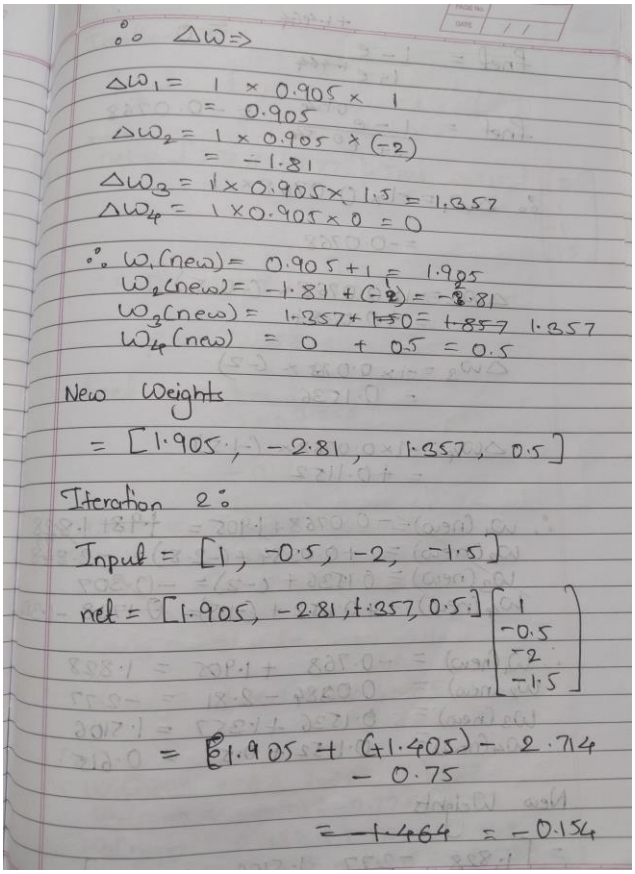
```
------------------Epoch  3 ------------------
```

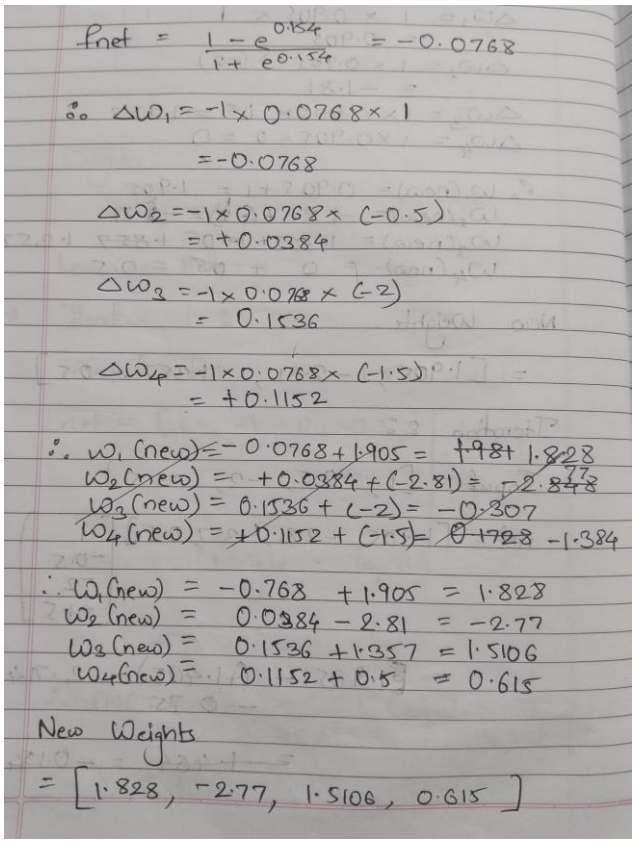
Old Weights	New Weights	Change in Weight	FNet
[1.82773, -2.77159, 1.51256, 0.61613]	[1.828, -3.704, 2.445, -0.783]	[-0.0, -0.93286, 0.93286, -1.39929]	-0.932859

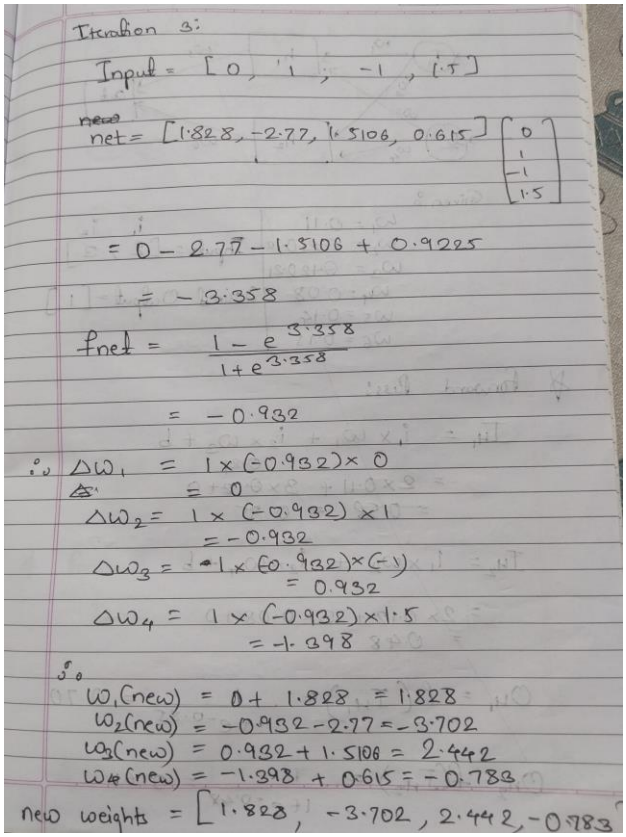
```
PS C:\Users\uditi>
```

Calculations Done:









Conclusion: In this lab, I studied and implemented the Hebbs learning rule, and compared the result with manual calculations.

Post Lab Descriptive Questions :

1. **Compare the Hebbian learning and competitive learning.**

Aspects	**Hebbian Learning**	**Competitive Learning**
Learning Principle	Cells that fire together, wire together.	Neurons compete to become active and learn from the winner.
Objective	Strengthen synaptic connections between neurons that are often active together.	Select a single neuron (or a group of neurons) as the winner to represent the most

		significant input.
Nature of Learning	Unsupervised learning.	Unsupervised learning.
Weight Update Rule	$\Delta w = \eta$ * (input neuron activity) * (output neuron activity.	$\Delta w = \eta$ * (input neuron activity) for the winner, and $\Delta w = 0$ for non-winners.
Competition Mechanism	No competition among neurons; all connections are updated simultaneously.	Neurons compete via lateral inhibition, where the most active neuron wins and inhibits others.
Use Cases	Basic associative learning; limited applicability in complex tasks.	Feature mapping, dimensionality reduction, and clustering in unsupervised learning tasks.
Robustness	Prone to instability and overfitting; may strengthen irrelevant connections.	Can produce a stable, organized representation of input patterns.
Applications	Theoretical concept in neuroscience; not widely used in modern artificial neural networks.	Used in Self-Organizing Maps (SOMs) and vector quantization neural networks.

2. **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.**
   i. **Use bipolar binary activation function**

Ans) Code:

```
import math

#libraries used only for data visualization:
import numpy as np
from tabulate import tabulate

#inputs given in the question
```

```
inputs = [
            [1, -2],
            [2, 3],
            [1, -1]
          ]

#choosing learning rate to be 1
learning_rate = 1

#initial weights given in the question
weights = [1, -1]

#function defined for calculating activation function
#using bipolar sigmoidal function
def activation_function(net, threshold=0):
    if net <= threshold:
        return -1
    else:
        return 1

#creating loop for iteration for each input array
#thus, there will be 3 epochs, because there are 3 input arrays

for j in range(len(inputs)):

    #initializing the value of net to be 0 before every
    #input array's calculations are done
    net = 0

    print("\n\n------------------Epoch ", (j + 1), "------------------")

    #taking dot product of weights array and inputs array to find the value of net
    for i in range(len(weights)):
        net = net + weights[i] * inputs[j][i]

    #actual output will be the activation function applied on the value of fnet
    fnet = activation_function(net)


    #making a variable for storing the value of old weights for data visualization
    #using the round function that comes with numpy to show only the first 5 digits
after the decimal
```

```
    old = list(np.round(weights, 5))

    #declaring a list to store the value of change in weights
    delta_w = []

    #for loop for calculating the value of new weights
    #purpose of for loop is matrix multiplication
    for k in range(len(weights)):

        #storing the value of change in weight given by learning_rate * output *
inputs[j][k]
        #for data visualization
        change_in_weight = learning_rate * fnet * inputs[j][k]
        delta_w.append(round(change_in_weight , 5))
        weights[k] += change_in_weight

    #making a variable for storing the value of new weights after this epoch for data
visualization
    new_weights = list(np.round(weights, 3))

    #making a list of numpy arrays to pass in the tabulate function
    #as passing numpy arrays directly to tabulate gives warnings and causes conflicts
    table_data = [old, new_weights, delta_w, fnet]

    #using tabulate functions for data visualization
    headers = ["Old Weights", "New Weights", "Change in Weight", "FNet"]
    table = tabulate([table_data], headers=headers, tablefmt="fancy_grid")
    print(table)
```

Output:

----------Epoch 1 ----------

Old Weights	New Weights	Change in Weight	FNet
[1, -1]	[2, -3]	[1, -2]	1

----------Epoch 2 ----------

Old Weights	New Weights	Change in Weight	FNet
[2, -3]	[0, -6]	[-2, -3]	-1

----------Epoch 3 ----------

Old Weights	New Weights	Change in Weight	FNet
[0, -6]	[1, -7]	[1, -1]	1

ii. Use continuous activation function

Ans) Code:
Activation Function used: Sigmoidal

```
import math

#libraries used only for data visualization:
import numpy as np
from tabulate import tabulate

#inputs given in the question
inputs = [
            [1, -2],
            [2, 3],
            [1, -1]
          ]

#choosing learning rate to be 1
learning_rate = 1

#initial weights given in the question
weights = [1, -1]

#function defined for calculating activation function
#using bipolar sigmoidal function
def activation_function(net):
    return 1 / (1 + np.exp(-net))
```

```
#creating loop for iteration for each input array
#thus, there will be 3 epochs, because there are 3 input arrays

for j in range(len(inputs)):

    #initializing the value of net to be 0 before every
    #input array's calculations are done
    net = 0

    print("\n\n------------------Epoch ", (j + 1), "------------------")

    #taking dot product of weights array and inputs array to find the value of net
    for i in range(len(weights)):
        net = net + weights[i] * inputs[j][i]

    #actual output will be the activation function applied on the value of fnet
    fnet = activation_function(net)


    #making a variable for storing the value of old weights for data visualization
    #using the round function that comes with numpy to show only the first 5 digits
after the decimal
    old = list(np.round(weights, 5))

    #declaring a list to store the value of change in weights
    delta_w = []

    #for loop for calculating the value of new weights
    #purpose of for loop is matrix multiplication
    for k in range(len(weights)):

        #storing the value of change in weight given by learning_rate * output *
inputs[j][k]
        #for data visualization
        change_in_weight = learning_rate * fnet * inputs[j][k]
        delta_w.append(round(change_in_weight , 5))
        weights[k] += change_in_weight

    #making a variable for storing the value of new weights after this epoch for data
visualization
    new_weights = list(np.round(weights, 3))
```

```
#making a list of numpy arrays to pass in the tabulate function
#as passing numpy arrays directly to tabulate gives warnings and causes conflicts
table_data = [old, new_weights, delta_w, fnet]

#using tabulate functions for data visualization
headers = ["Old Weights", "New Weights", "Change in Weight", "FNet"]
table = tabulate([table_data], headers=headers, tablefmt="fancy_grid")
print(table)
```

Output:

```
------------------Epoch  1 ------------------
╒═══════════════╤═══════════════╤═══════════════════╤═══════════╕
│ Old Weights   │ New Weights   │ Change in Weight  │      FNet │
╞═══════════════╪═══════════════╪═══════════════════╪═══════════╡
│ [1, -1]       │ [1.953, -2.905] │ [0.95257, -1.90515] │ 0.952574 │
╘═══════════════╧═══════════════╧═══════════════════╧═══════════╛

------------------Epoch  2 ------------------
╒═══════════════════╤═══════════════╤═══════════════════╤═══════════╕
│ Old Weights       │ New Weights   │ Change in Weight  │      FNet │
╞═══════════════════╪═══════════════╪═══════════════════╪═══════════╡
│ [1.95257, -2.90515] │ [1.969, -2.881] │ [0.01616, 0.02424] │ 0.00807963 │
╘═══════════════════╧═══════════════╧═══════════════════╧═══════════╛

------------------Epoch  3 ------------------
╒═══════════════════╤═══════════════╤═══════════════════╤═══════════╕
│ Old Weights       │ New Weights   │ Change in Weight  │      FNet │
╞═══════════════════╪═══════════════╪═══════════════════╪═══════════╡
│ [1.96873, -2.88091] │ [2.961, -3.873] │ [0.99223, -0.99223] │ 0.99223 │
╘═══════════════════╧═══════════════╧═══════════════════╧═══════════╛
PS C:\Users\uditi>
```

Date: **Signature of faculty in-charge**