# Assignment 2

# Experiment 4 and 5

### Title: ANN Learning Rules

Name of Student: Sangeet Agrawal PRN No. 21070122140

DoP4: 5 Aug

DoP5: 12 Aug DoS: 19 Aug

**Title:** ANN Learning Rules

**Aim:** Implement a Program to Train XOR Gate Using Hebbian Learning with Bipolar Input and Targets

**Objective:** Students will learn and implement
- Unsupervised Learning model
- Hebbian learning

* Theory :

→ Hebbian Learning Rule:

It is an unsupervised learning rule that adjusts weights in neural networks based on the principle that simultaneous activation of neurons strengthens their connection. In this experiment, the XOR gate, which outputs true only when inputs differ, is trained using Hebbian learning with bipolar inputs (-1 & 1) and targets. The weights and bias are iteratively updated based on the product of the input and target, continuing over multiple epochs until the network correctly learns to produce the correct XOR output.

* Procedure:

1) Initialize Inputs and Targets for the XOR in Bipolar form.

2) Initialize the weights and bias with random values.

3) Define the learning rate for Hebbian learning rule..

4) Train the network for a specified number of epochs.

5) Compute the output using the trained weights and bias & apply sign function to determine the output.

6) Print the final outputs of the trained network.

**Code:**

```
inputs = [-1 -1; -1 1; 1 -1; 1 1];
targets = [-1; 1; 1; -1];

weights = randn(2, 1);
bias = randn();

learning_rate = 0.1;
```

```
% Hebbian learning rule implementation
epochs = 100;
for epoch = 1:epochs
   for i = 1:size(inputs, 1)
       x = inputs(i, :)';
       t = targets(i);
       y = dot(weights, x) + bias;
       weights = weights + learning_rate * x * t;
       bias = bias + learning_rate * t;
   end
end
% Compute the output
outputs = zeros(size(targets));
for i = 1:size(inputs, 1)
   y = dot(weights, inputs(i, :)') + bias;
   outputs(i) = sign(y);
end

disp('Final output:');
disp(targets');
```

**Output:**

Command Window

```
>> Assignment_4
Final output:
    -1     1     1    -1
```

* Conclusion: The Hebbian learning implementation for the XOR gate effectively trained the neural network to produce correct outputs. This experiment demonstrates the potential of Hebbian learning in neural network training

## Experiment 5

**Title:** ANN Learning Rules

**Aim:** Design and simulate program for implementation of Single discrete single layer Perceptron Learning Algorithm (SDPTA). To Simulate Recall for SDPTA.

**Objective:** Students will learn and implement

- Perceptron Model
- Neural Networks Fundamentals

**Explanation/Stepwise Procedure/ Algorithm:**

**Problem Statement:** Implement AND function using perceptron model

Truth table for AND function is:

| X1 | X2 | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* Theory :

→ Single Discrete Perceptron Learning Algorithm (SDPTA):

It is used for binary classification tasks in neural networks. It updates weights based on input features and target outputs, iteratively minimizing classification error. The algorithm adjusts weights proportionally to the error until all samples are correctly classified or a set of number of iterations is reached. SDPTA is fundamental for understanding more complex neural networks and learning algorithms.

*. Procedure:

1) Define the input matrix & desired outputs & initialize the weights and set the learning rule.

2) Set total error to infinity & initialize iterations count.

3) Calculate the net input, determine the output, compute the error, update weights based on error & accumulate total error.

4) Increment iteration count.

5) Print the final weight vector, total error & no. of iterations

6) Simulate Recall for each input vector.

**Input & Output:**

```
Enter the x1 value
[1 -1 1 -1]
Enter the x2 value
[1 1 -1 -1]
x1=1     x2=1     ta=1     yin=0    y=0     w1=1     w2=1     b=1
x1=-1    x2=1     ta=-1    yin=1    y=1     w1=2     w2=0     b=0
x1=1     x2=-1    ta=-1    yin=2    y=1     w1=1     w2=1     b=-1
x1=-1    x2=-1    ta=-1    yin=-3   y=-1    w1=1     w2=1     b=-1
>>
```

**Code:**

```
import numpy as np

inputs = np.array([[1, -2, 0, -1],
                   [0, 1.5, -0.5, -1],
                   [-1, 1, 0.5, -1]])
```

```python
desired_outputs = np.array([-1, -1, 1])

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

total_error = float('inf')
while total_error != 0:
    total_error = 0
    for i in range(len(inputs)):
        net_input = np.dot(weights, inputs[i, :])
        output = 1 if net_input >= 0 else -1
        error = desired_outputs[i] - output
        total_error += abs(error)
        weights = weights + learning_rate * error * inputs[i, :]
    iterations+=1

print("Final weight vector:", weights)
print("Total error:", total_error)
print("Number of iterations:", iterations)

for i in range(len(inputs)):
    net_input = np.dot(weights, inputs[i, :])
    output = 1 if net_input >= 0 else -1
    print(f"Input: {inputs[i]} -> Output: {output}")
```

**Output:**

```
Final weight vector: [-3.   2.   2.   2.5]
Total error: 0
Number of iterations: 2
Input: [ 1. -2.  0. -1.] -> Output: -1
Input: [ 0.   1.5 -0.5 -1. ] -> Output: -1
Input: [-1.   1.   0.5 -1. ] -> Output: 1
```

**Conclusion:**

* Conclusion: The SDPTA implementation effectively adjusted weights to classify inputs correctly. This experiment demonstrates the basics of perceptron learning and effectiveness of iterative weight updates.