## Aim:

LAB Manual PART A

(PART A : TO BE REFFERED BY STUDENTS)

**Experiment No.03**

Implementation of logic gate (AND, OR, NOT, NAND, NOR ) using Hebb Net. Implementation of Hebb Net to classify two-dimensional two input patterns.

## Prerequisite:

* + 1. Theoretical knowledge of Hebb model of neural network.
    2. Knowledge of Bipolar Inputs.
    3. Different programming language structure overview.

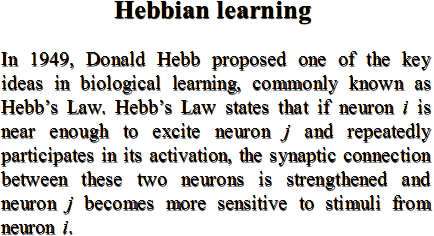
## Outcome:

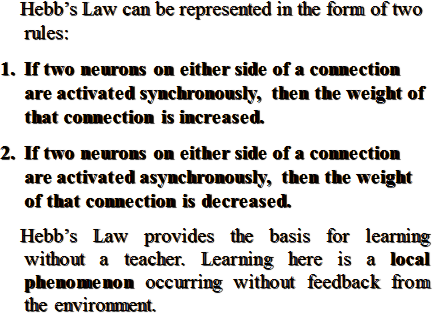
### After successful completion of this experiment students will be able to

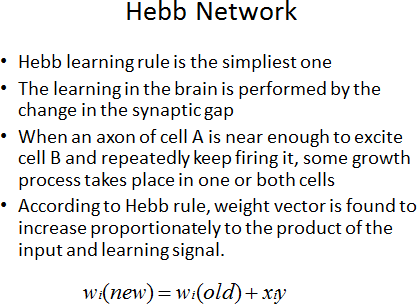
* + 1. Design different logical functions using Hebb Net.
    2. Apply Hebb Net to classify input patterns.
    3. Design neural network by making use of Hebb Net.

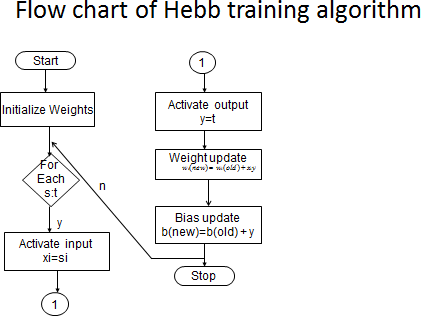
## Theory:

**A.4.1. Hebb Net**:









### The Training Algorithm for Hebb network is as given below:

Step 0: Initialize the weights. It may be initialized to zero i.e. wi = 0; for i = 1 to n where ‘n’ is the total number of input neurons.

Step 1: Step 2 – 4 have to be performed for each input training vector and targer output pair s:t.

Step 2: Input units activations are set. Generally, the activation function of input layer is identity function: xi = si for i= 1 to n.

Step 3: Output unit activations are set: y = t.

Step 4: Weight adjustment and bias adjustments are performed wi(new) = wi(old) + xiy

b(new) = b(old)+ y

## Procedure/Algorithm:

* + 1. **:**

### Read the input patterns

* + - 1. **Initialize the weights**
      2. **Apply the Hebb Learning algorithm.**
      3. **Generate the output.**
      4. **Repeat the same procedure (Step 1 to 4) for all input patterns.**
      5. **Realize the logic.**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# PART B

(PART B : TO BE COMPLETED BY STUDENTS)

***(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)***

|  |  |
| --- | --- |
| Roll No.: C026 | Name: Anirbaan Ghatak |
| Class : B | Batch: B1 |
| Date of Experiment: | Date of Submission: |
| Grade : | Time of Submission: |
| Date of Grading: |  |

## Software Code written by student:

import numpy as np

# Function to train the Hebbian network

def train\_hebbian(X, epochs, learning\_rate):

    # Initialize weights and bias to zero

    n\_samples, n\_features = X.shape

    weights = np.zeros(n\_features)

    # Training loop

    for epoch in range(epochs):

        for i in range(n\_samples):

            for j in range(n\_features):

                # Apply the Hebbian learning rule

                weights[j] += learning\_rate \* X[i, j] \* X[i, j]

    return weights

# Function to make predictions using the trained Hebbian network

def predict\_hebbian(X, weights):

    return np.dot(X, weights)

# Define input for AND operation

X\_and = np.array([[0, 0],

                    [0, 1],

                    [1, 0],

                    [1, 1]])

# Add a bias

X\_and\_bias = np.hstack([X\_and, np.ones((X\_and.shape[0], 1))])

# Training for AND operation

weights\_and = train\_hebbian(X\_and\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_and = predict\_hebbian(X\_and\_bias, weights\_and)

weights\_and, predictions\_and

# Define input for OR operation

X\_or = np.array([[0, 0],

                  [0, 1],

                  [1, 0],

                  [1, 1]])

# Add a bias

X\_or\_bias = np.hstack([X\_or, np.ones((X\_or.shape[0], 1))])

# Train for OR operation

weights\_or = train\_hebbian(X\_or\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_or = predict\_hebbian(X\_or\_bias, weights\_or)

weights\_or, predictions\_or

# Define input for NOT operation (single input)

X\_not = np.array([[0],

                  [1]])

# Add a bias

X\_not\_bias = np.hstack([X\_not, np.ones((X\_not.shape[0], 1))])

# Train for NOT operation

weights\_not = train\_hebbian(X\_not\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_not = predict\_hebbian(X\_not\_bias, weights\_not)

weights\_not, predictions\_not

# Define input for NAND operation

X\_nand = np.array([[0, 0],

                    [0, 1],

                    [1, 0],

                    [1, 1]])

# Add a bias term to the input

X\_nand\_bias = np.hstack([X\_nand, np.ones((X\_nand.shape[0], 1))])

# Train for NAND operation

weights\_nand = train\_hebbian(X\_nand\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_nand = predict\_hebbian(X\_nand\_bias, weights\_nand)

weights\_nand, predictions\_nand

X\_nor = np.array([[0, 0],

                  [0, 1],

                  [1, 0],

                  [1, 1]])

# Add a bias

X\_nor\_bias = np.hstack([X\_nor, np.ones((X\_nor.shape[0], 1))])

# Train for NOR operation

weights\_nor = train\_hebbian(X\_nor\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_nor = predict\_hebbian(X\_nor\_bias, weights\_nor)

weights\_nor, predictions\_nor

# Define input for ANDNOT operation (AND with the NOT of X2)

X\_andnot = np.array([[0, 1],

                     [0, 0],

                     [1, 1],

                     [1, 0]])

# Add a bias

X\_andnot\_bias = np.hstack([X\_andnot, np.ones((X\_andnot.shape[0], 1))])

# Train for ANDNOT operation

weights\_andnot = train\_hebbian(X\_andnot\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_andnot = predict\_hebbian(X\_andnot\_bias, weights\_andnot)

weights\_andnot, predictions\_andnot

# Define input for XOR operation

X\_xor = np.array([[0, 0],

                  [0, 1],

                  [1, 0],

                  [1, 1]])

# Add a bias

X\_xor\_bias = np.hstack([X\_xor, np.ones((X\_xor.shape[0], 1))])

# Train for XOR operation

weights\_xor = train\_hebbian(X\_xor\_bias, epochs=100, learning\_rate=0.1)

# Make predictions

predictions\_xor = predict\_hebbian(X\_xor\_bias, weights\_xor)

weights\_xor, predictions\_xor

## Output:

## 

## 

## 

## 

* 1. **Observations and learning:**

The Hebbian neural network experiment simulated logic gates (like AND, OR, NOT). By strengthening connections when inputs fired together, the network displayed how it could learn and mimic gate behaviors.

## Conclusion:

## The logic gates mentioned in the experiment were implemented and the output was verified.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***