Aim: Write a Python program to design a Hopfield network which stores 4 vectors.

Software requirement for Python: Jupyter Notebook

Theory:

Hopfield neural network was invented by Dr. John J. Hopfield in 1982. It consists of a single layer which contains one or more fully connected recurrent neurons. The Hopfield network is commonly used for auto-association and optimization tasks.

Discrete Hopfield Network

A Hopfield network which operates in a discrete line fashion or in other words, it can be said the input and output patterns are discrete vector, which can be either binary 0,1 or bipolar +1,−1 in nature. The network has symmetrical weights with no self-connections i.e., wij = wji and wii = 0 (Fig.1).

Following are some important points to keep in mind about discrete Hopfield network –

1. This model consists of neurons with one inverting and one non-inverting output.
2. The output of each neuron should be the input of other neurons but not the input of self.
3. Weight/connection strength is represented by wij.
4. Connections can be excitatory as well as inhibitory. It would be excitatory, if the output of the neuron is same as the input, otherwise inhibitory.
5. Weights should be symmetrical, i.e. wij = wji

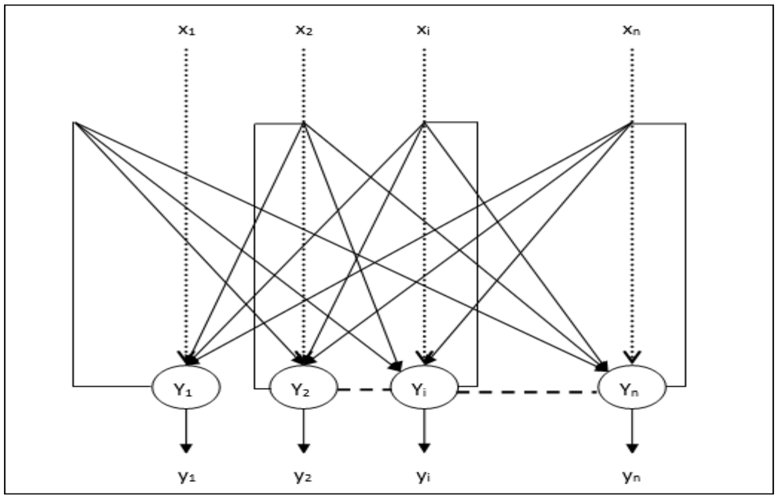


Fig.1- Discrete Hopfield Network

The output from Y1 going to Y2, Yi and Yn have the weights w12, w1i and w1n respectively. Similarly, other arcs have the weights on them.

Training Algorithm

During training of discrete Hopfield network, weights will be updated. As we know that we can have the binary input vectors as well as bipolar input vectors.

Hence, in both the cases, weight updates can be done with the following relation

Case 1 − Binary input patterns

For a set of binary patterns sp, p = 1 to P

Here, sp = s1p, s2p,..., sip,..., snp

Weight Matrix is given by

$$w_{ij} = \sum_{p=1}^{P}[2s_i(p) - 1][2s_j(p) - 1] \quad for\ i \neq j$$

Case 2 − Bipolar input patterns

For a set of binary patterns sp, p = 1 to P

Here, sp = s1p, s2p,..., sip,..., snp

Weight Matrix is given by

$$w_{ij} = \sum_{p=1}^{P}[s_i(p)][s_j(p)] \quad for\ i \neq j$$

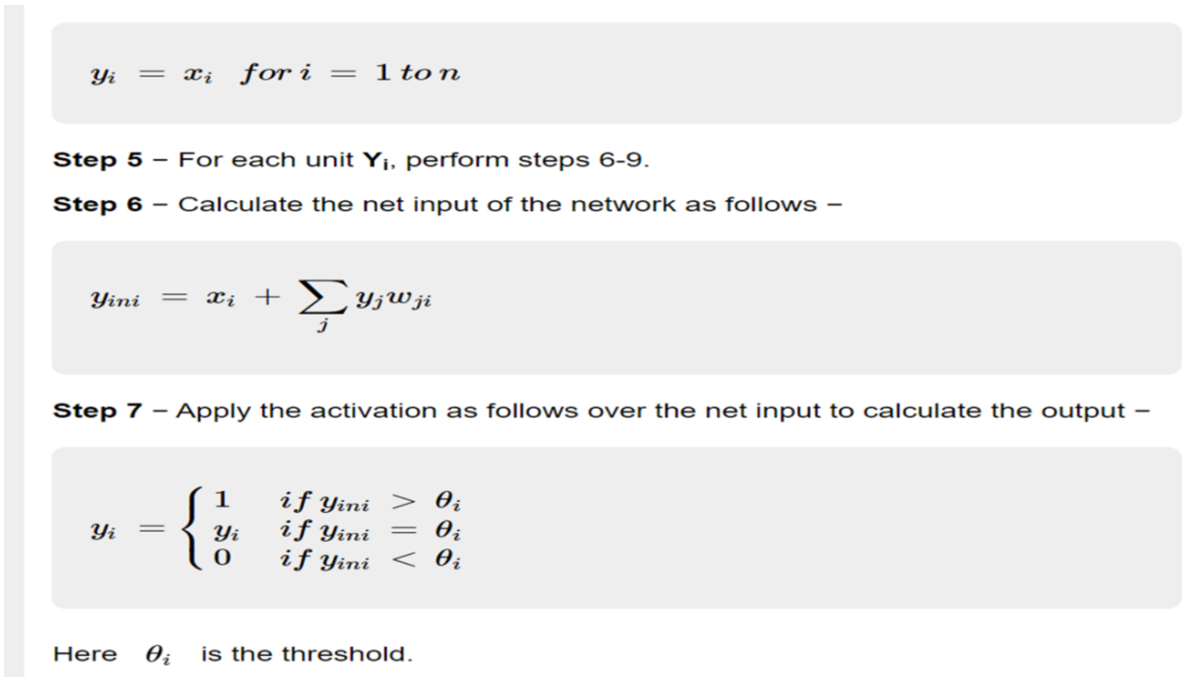
Testing Algorithm

Step 1 − Initialize the weights, which are obtained from training algorithm by using Hebbian principle.

Step 2 − Perform steps 3-9, if the activations of the network is not consolidated.

Step 3 − For each input vector X, perform steps 4-8.

Step 4 − Make initial activation of the network equal to the external input vector X as follows −



$$y_i = x_i \quad for\, i = 1\, to\, n$$

Step 5 – For each unit $\mathbf{Y_i}$, perform steps 6-9.

Step 6 – Calculate the net input of the network as follows –

$$y_{ini} = x_i + \sum_j y_j w_{ji}$$

Step 7 – Apply the activation as follows over the net input to calculate the output –

$$y_i = \begin{cases} 1 & if\, y_{ini} > \theta_i \\ y_i & if\, y_{ini} = \theta_i \\ 0 & if\, y_{ini} < \theta_i \end{cases}$$

Here θ_i is the threshold.

Step 8 – Broadcast this output yi to all other units.

Step 9 – Test the network for conjunction.

Procedure:

This code implements a Hopfield network, a type of recurrent neural network used for content-addressable memory and pattern recognition tasks.

The HopfieldNetwork class is defined with an init method that initializes the network with the number of neurons (n) and weights initialized as an n x n matrix of zeros. The train method takes in a set of patterns and updates the weights matrix based on the outer product of each pattern with itself. The diagonal elements of the weights matrix are set to zero to prevent self-connections.

The recall method takes in a pattern and uses it to update the activation of the network until it converges to a stable state. The energy of the system is calculated at each iteration, and the activation is updated using the sign function applied to the dot product of the weights matrix and the current activation. If the new activation is the same as the previous one, the iteration stops and the output is returned. If the maximum number of iterations is reached before convergence, the current activation is returned as the output.

The main code initializes the Hopfield network with 4 neurons, trains it on a set of 4 input patterns, and then recalls each pattern to check if it is correctly recognized by the network.

Conclusion: Thus, we have studied to design a Hopfield network which stores 4 vectors.

Program Code and Output

```
import numpy as np
class HopfieldNetwork:
    def __init__(self, n):
        self.n = n
        self.weights = np.zeros((n, n))

    def train(self, patterns):
        self.weights = np.zeros((self.n, self.n))
        for p in patterns:
            self.weights += np.outer(p, p)
        np.fill_diagonal(self.weights, 0)

    def recall(self, pattern, max_iters=100):
        for i in range(max_iters):
            energy = -0.5 * np.dot(np.dot(pattern, self.weights), pattern)
            new_pattern = np.sign(np.dot(self.weights, pattern))
            if np.array_equal(new_pattern, pattern):
                return new_pattern
            pattern = new_pattern
        return pattern

# Define the input patterns
patterns = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 0, 0], [0, 0, 1, 1]])
# Initialize the Hopfield network
hopfield = HopfieldNetwork(n=4)
```

```
# Train the network

hopfield.train(patterns)

# Recall the patterns

for pattern in patterns:

    recalled_pattern = hopfield.recall(pattern)

    print("Input pattern:", pattern)

    print("Recalled pattern:", recalled_pattern)
```

OUTPUT OF THE CODE

```
Input pattern: [1 0 1 0]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [0 1 0 1]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [1 1 0 0]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [0 0 1 1]

Recalled pattern: [1. 1. 1. 1.]
```