Rajshahi University of Engineering & Technology
Department of Computer Science & Engineering

Course No : CSE 4204
Course title : Sessional based on CSE 4203
Date : 18-12-2023

Experiment No. : 04

Name of the experiment : a. Design and implementation of Kohonen Self-organizing Neural Networks algorithm.
b. Design and implementation of Hopfield Neural Networks algorithm

Submitted By :

Tanmoy Sarkar Pranto
Roll : 1803166
Section : C
Dept. of Computer Science and Eengieering
Rajshahi University of Engineering & Technology

Submitted to :

Rizoan Toufiq
Assistant Professor
Dept. of Computer Science and Eengieering
Rajshahi University of Engineering & Technology

Contents

1 Kohonen Algorithm

1.1 Theory

The Kohonen Network or Kohonen Self Organizing Map(SOM) is an unsupervised learning model to cluster data points based on the distance. This algorithm is also used for dimensionality reduction as implementation of Kohonen Algorithm is predominently two dimensional. [?]
The main difference between a regular network and a Kohonen network is that a regular network has nodes placed in different layers, like, input layer, one or multiple hidden layer and output layer. But for Kohonen Network, the nodes are placed in a flat grid. Each of the node in itself is a output node. Every input is connected to all the nodes.
There is no derivative approach involved in adapting the weights of Kohonen Network. Learning rate is decreased over time or epoch. The learning rate is kept high (q $>$ 0.5) to allow large weight modifications and hopefully settle into an approximate mapping as quickly as possible. This learning rate is decreased over time.
Caution should be taken when initializing the weight matrix. If the weight values are truly random, the network may suffer non-convergent or very slow training cycles. One method is to initialise all the weights so that they are normal and coincident (i.e. with the same value).
There is also a concept of neighborhood which is taken around each of the nodes. When the winning node is selected, weight is updated not only of that node but also of the other nodes that falls in the neighborhood of that winning node. This radius of neighborhood is decreased over time.

1.2 Algorithm

1. Initialize network

   Define $w_{ij}(t)(0 \leq i \leq n-1)$ to be the weight from input i to node j at time t. Initialize weights from the inputs to the nodes to small random values.

2. Present input

   Present input $x_0(t), x_1(t), x_2(t), ..., x_{n-1}(t)$ where $x_i(t)$ is the input to node i at time t.

3. Calculate distances

   Compute the distance d_j between the input and each output node j, given by

$$d_j = \sum_{i=0}^{n-1} (x_i(t) - w_{ij}(t))^2$$

4. Select minimum distance

   Designate the output node with minimum d_j to be j^*

5. Update weights

   Update weights for node j^*. New weights are

$$w_{ij}(t+1) = w_{ij}(t) + \eta(t)(x_i(t) - w_{ij}(t))$$

The term $\eta(t)$ is a gain term $(0 < \eta(t) < 1)$ that decreases in time, so slowing the weight adaption.

6. Repeat by going to 2.

1.3 Code

```
import numpy as np
import matplotlib.pyplot as plt
import random as rand

no_of_inputs = 20
no_of_nodes = 10
learning_rate = 0.6

# Random weight initialization,each row will have weights for one node
weights = np.random.random([no_of_nodes, no_of_inputs ])
# print(weights)

# Inputs

inputs = np.random.randint(2,size=(no_of_inputs,no_of_inputs))
# print(inputs)

# Calculate Distances
new_res = [0]*no_of_inputs

for i in range(10):
    for input in range(no_of_inputs):
        distances = []
        for node in range(no_of_nodes):
            distance = np.sum((inputs[input]-weights[node])**2)
            distances.append(distance)
        # print(f"Distances = {distances} for input = {inputs[input]}")
        index_of_min_distance = np.argmin(distances)
        new_res[input] = index_of_min_distance + 1
        # print(f"Min Distance index = {index_of_min_distance}")

        # update weight
        weights[index_of_min_distance] = weights[index_of_min_distance] + (learning_rate*(inputs[input]-
        # print(f"Updated weights = {weights}")

        # print(f"{res} and {new_res}")
    if learning_rate<0.05:
        learning_rate -= 0.05
print(f"Final Cluster = {new_res}")
# print(f"Final weights: {weights}")

test_input = [1,1,1,1,1,0,0,0,0,1,1,1,1,1,1,0,0,0,0,1,1]
distances = []
for node in range(no_of_nodes):
```

```
    distance = np.sum((test_input-weights[node])**2)
    distances.append(distance)
class_of_test_input = np.argmin(distances)
print(f"Test Class output: {class_of_test_input}")
```

1.4 Result

Final Cluster = [5, 4, 10, 1, 5, 10, 6, 8, 8, 7, 1, 2, 3, 9, 3, 3, 2, 3, 9, 1]
Test Class output: 2

1.5 Discussion

The Kohonen Self Organizing Network is implemented with 20 inputs and 10 nodes on the network, which means 10 clusters. Learning rate was taken 0.6 initially, which was lowered by 0.05 after each epoch till its higher than 0.05. Neighborhood was not taken here as I didn't find any proper way of dealing with the reduction in radius of neighborhood. After running the weight updation of weights, the final cluster is given in the result section.

2 Hopfield Network

2.1 Theory

Hopfield Network is a special type of neural network algorithm which is used to store and recall patterns based on given input. The Hopfield net consists of a number of nodes, each connected to every other node, it is fully-connected network.It is also a symmetricully weighted network, since the weights on the link from one node to another are the same in both directions. Each node has, like the single-layer perceptron, a threshold and a step-function, and the nodes calculate the weighted sum of their inputs minus the threshold value, passing that through the step function to determine their output state. In the Hopfield net, this first output is taken as the new input, which produces a new output, and so on; the solution occurs when there is no change from cycle to cycle.

2.2 Algorithm

1. Assign connection weights

$$w_{ij} = \begin{cases} \sum_{s=0}^{M-1} x_i^s x_j^s & i \neq j \\ 0 & i = j, 0 \leq i, j \leq M-1 \end{cases}$$

where w_{ij} is the connection weight between node i and node j, and x_i^s is element i of the exemplar pattern for class s, and is either $+1$ or -1. There are M patterns, from 0 to $M-1$, in total. The thresholds of the units are zero.

2. Initialise with unknown pattern

$$\mu_i(0) = x_i \qquad 0 \leq i \leq N-1$$

where $\mu_i(t)$ is the output of node i at time t.

3. Iterate until convergence

$$\mu_i(t+1) = f_h \left[\sum_{i=0}^{N-1} w_{ij} \mu_j(t) \right] \qquad 0 \leq j \leq N-1$$

The function f_h is the hard-limiting non-linearity, the step function, as in figure 3.3. Repeat the iteration until the outputs from the nodes remain unchanged.

2.3 Code

```
import numpy as np


stored_patterns = np.array([[1,-1,1],[1,-1,-1],[-1,-1,1],[1,1,1],[-1,-1,-1]])
no_of_neurons = len(stored_patterns[0])
# print(no_of_neurons)
# Weights Initialization
W = np.zeros([no_of_neurons,no_of_neurons])


# Update weights for storing
for i in range(len(stored_patterns)):
    for row in range(no_of_neurons):
        for col in range(no_of_neurons):
            if row != col:
                W[row][col] += stored_patterns[i][row]*stored_patterns[i][col]

print(f"Weights after storing patterns = {W}")
input_pattern = np.array([1,1,-1])

# Recall Pattern
print(f"Input Pattern = {input_pattern}")
for i in range(10):
    temp = np.matmul(input_pattern,W)
    temp = [1 if num>=0 else -1 for num in temp]
    input_pattern = temp.copy()
    # print(f"Epoch {i} = {temp}")

print(f"Recalled Pattern = {temp}")
```

2.4 Result

Weights after storing patterns = [[0. 1. 1.] [1. 0. 1.] [1. 1. 0.]]
Input Pattern = [1 1 -1]
Recalled Pattern = [1, 1, 1]

2.5 Discussion

The Kohonen Self Organizing Network is implemented with 20 inputs and 10 nodes on the network, which means 10 clusters. Learning rate was taken 0.6 initially, which was lowered by 0.05 after each epoch till its higher than 0.05. Neighborhood was not taken here as I didn't find any proper way of dealing with the reduction in radius of neighborhood. After running the weight updation of weights, the final cluster is given in the result section.