LAB REPORT 4

November 18, 2020

Submitted by:

Kazi Amit Hasan
Roll: 1503089
Department of Computer Science & Engineering
Rajshahi University of Engineering & Technology

November 18, 2020

0.1 Title

Implementation of **Kohonen Self-Organizing Neural Network** algorithm.

0.2 Objectives

1. Learning the concept of kohonen self-organizing neural network algorithm.
2. Learning the maths behind the algorithm.
3. Learning and implementing in a dataset.

0.3 Methodology

The KSOM (also called a feature map or Kohonen map) is an unsupervised ANN algorithm (Kohonen et al., 1996). It is usually presented as a dimensional grid or map whose units (nodes or neurons) become tuned to different input data patterns. The principal goal of the KSOM is to transform an incoming signal pattern of arbitrary dimension into a two-dimensional discrete map. This mapping roughly preserves the most important topological and metric relationship of the original data elements, implying that not much information is lost during the mapping.

To train the KSOM, the multi-dimensional input data are first standardised by deducting the mean and then dividing the result by the standard deviation. Then a standardised input vector is chosen at random and applied to each of the individual neurons, which are initially seeded with weights or code vectors. The elements of the code vectors also have a mean of zero and variance of unity. Comparison between the input vector and the code vectors is then made to identify the code vector most similar to the applied input vector. The identification uses the Euclidian distance, which is defined as:

$$d(x, x') = \sqrt{\left(x_1 - x'_1\right)^2 + + \left(x_n - x'_n\right)^2}$$

The neuron whose vector most closely matches the input data vector, i.e. for which the Di is a minimum, is chosen as a winning node or the best matching unit (BMU). The vector weights of this winning neuron and those of its adjacent

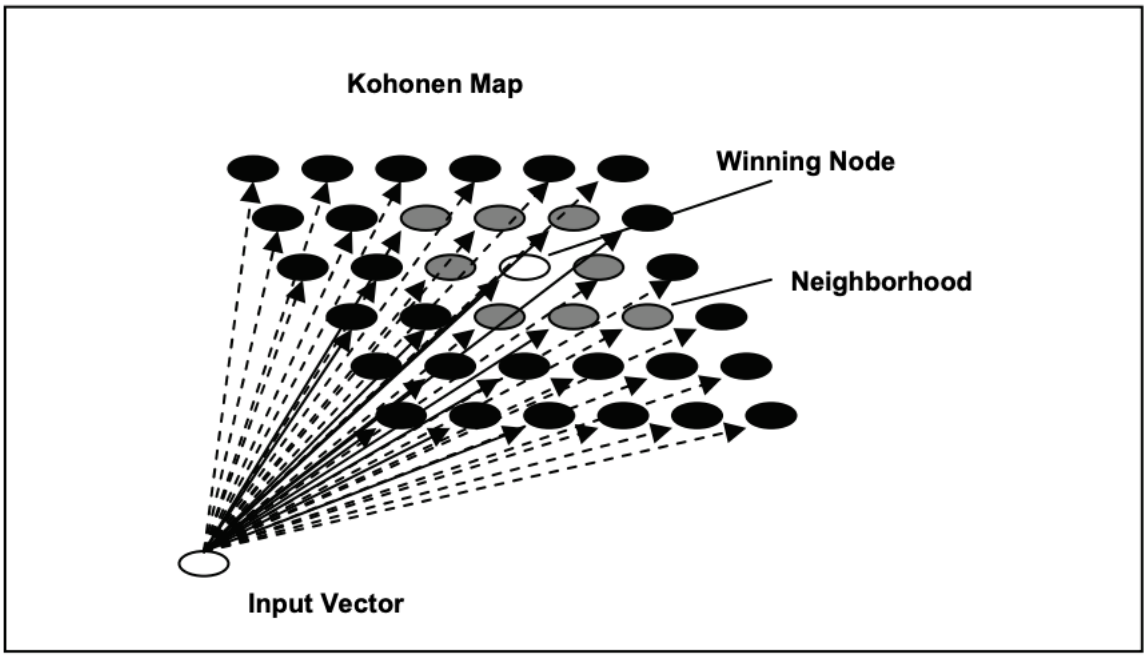

Figure 1: Kohonen self organizing algorithm

neurons are then adjusted to match the input vector using an appropriate algorithm, thus bringing the two vectors further into agreement.

We can summarize the operation of the perceptron as follows it:

- Step 1: Initialise each node's weight w_ij to a random value
- Step 2: Select a random input vector x_k
- Step 3: Compute Euclidean distance between the input vector x(t) and the weight vector
- Step 4: Track the node that produces the smallest distance t
- Step 5: Find the overall Best Matching Unit (BMU)
- Step 6: Determine topological neighbourhood ij(t) its radius (t) of BMU in the Kohonen Map
- Step 7: Repeat for all nodes in the BMU neighbourhood
- Step 8: Repeat this whole iteration until reaching the chosen iteration limit

0.4 Implementation

The tools that I used for the implementation purpose of this lab are Jupyter notebook. The codes are written in Python language. The algorithm steps were discussed in Methodology section.

0.5 Code

0.5.1 Implementation the Kohonen self organizing NN algorithm in Python

```
import math
import sys

MAX_CLUSTERS = 2
VECTORS = 4
VEC_LEN = 4
DECAY_RATE = 0.96
MIN_ALPHA = 0.001

# Dataset
training_patterns = [[1, 1, 0, 0],
                     [0, 0, 0, 1],
                     [1, 0, 0, 0],
                     [0, 0, 1, 1]]

tests = [[1, 0, 0, 1],
         [0, 1, 1, 0],
         [1, 0, 1, 0],
         [0, 1, 0, 0]]


# Main Class
class SOM_Class1:
```

```
def __init__(self, numVectors, maxClusters,
alphaStart, minimumAlpha, decayRate, vectorLength):
    self.mVectors = numVectors
    self.mVecLen = vectorLength
    self.mAlpha = alphaStart
    self.minAlpha = minimumAlpha
    self.decayRate = decayRate
    self.mIterations = 0
    self.maxClusters = maxClusters
    self.mD = [[]] * maxClusters
    self.w = [[0.2, 0.6, 0.5, 0.9],
              [0.8, 0.4, 0.7, 0.3]]

def compute_input(self, vectorNumber, trainingTests):
    self.mD[0] = 0.0
    self.mD[1] = 0.0

    for i in range(self.maxClusters):
        for j in range(self.mVectors):
            self.mD[i] += math.pow((self.w[i][j] - trainingTests
            [vectorNumber][j]), 2)

    return

def train(self, patterns, trainingTests):
    self.mIterations = 0

    while self.mAlpha > self.minAlpha:
        self.mIterations += 1
        for i in range(self.mVectors):
            self.compute_input(i, trainingTests)

            # See which is smaller, mD(0) or mD(1)?
            dMin = 1 if self.mD[0] > self.mD[1] else 0
```

```
            # Update the weights on the winning unit.
            for j in range(self.mVectors):
                self.w[dMin][j] = self.w[dMin][j] + (self.mAlpha *
                (patterns[i][j] - self.w[dMin][j]))

        # Reduce the learning rate.
        self.mAlpha = self.decayRate * self.mAlpha

    return

def test(self, patterns, trainingTests):
    # Print clusters created.
    sys.stdout.write("Clusters for training input:\n")

    for i in range(self.mVectors):
        self.compute_input(i, trainingTests)

        dMin = 1 if self.mD[0] > self.mD[1] else 0

        sys.stdout.write("\nVector ( ")

        for j in range(self.mVectors):
            sys.stdout.write(str(patterns[i][j]) + ", ")

        sys.stdout.write(") falls into category " + str(dMin) + "\n")

    # Print weight matrix.
    sys.stdout.write("\n")
    for i in range(self.maxClusters):
        sys.stdout.write("Weights for Node " + str(i) + " connections:\n")

        for j in range(self.mVecLen):
            sys.stdout.write("{:03.3f}".format(self.w[i][j]) + ", ")

        sys.stdout.write("\n\n")
```

```
        # Print post-training tests.
        sys.stdout.write("Categorized test input:\n")
        for i in range(self.mVectors):
            self.compute_input(i, trainingTests)

            dMin = 1 if self.mD[0] > self.mD[1] else 0

            sys.stdout.write("\nVector ( ")

            for j in range(self.mVectors):
                sys.stdout.write(str(trainingTests[i][j]) + ", ")

            sys.stdout.write(") falls into category " + str(dMin) + "\n")

        return

    def get_iterations(self):
        return self.mIterations


if __name__ == '__main__':
    Alpha = 0.9
    som = SOM_Class1(VECTORS, MAX_CLUSTERS, Alpha,
    MIN_ALPHA, DECAY_RATE, VEC_LEN)
    som.train(training_patterns, tests)
    som.test(training_patterns, tests)

    sys.stdout.write("\nIterations: " + str(som.get_iterations()) + "\n")
```

0.6 Results and Performance Analysis

In the code, I already defined the maximum number of clusters, nodes, decay rate. A self made small dataset was used for this lab which have four arrays. I used training and testing dataset. As Kohonen is an unsupervised algorithm, I

```
Clusters for training input:

Vector ( 1, 1, 0, 0, ) falls into category 0

Vector ( 0, 0, 0, 1, ) falls into category 1

Vector ( 1, 0, 0, 0, ) falls into category 0

Vector ( 0, 0, 1, 1, ) falls into category 1

Weights for Node 0 connections:
0.495, 0.244, 0.255, 0.505,

Weights for Node 1 connections:
0.494, 0.245, 0.256, 0.506,

Categorized test input:

Vector ( 1, 0, 0, 1, ) falls into category 0

Vector ( 0, 1, 1, 0, ) falls into category 1

Vector ( 1, 0, 1, 0, ) falls into category 0

Vector ( 0, 1, 0, 0, ) falls into category 1

Iterations: 167
```

Figure 2: Results

didn't define any type of class here neither in test dataset. I defined the weights in my code. In the result figure, we can see that our algorithm selected clusters according to the algorithm and I showed all the final weights for nodes. Then I tested my dataset and also showed the results in the result figure. This took 167 iterations in totoal.

0.7 Conclusion

By implementing this, we knew about the fundamentals of one of the most basic machine learning algorithms. It has some basic advantages. In this lab, I tried to implement the kohonen self organizing neural network algorithm without using any automated library functions All the codes and neccesary files are available in my github profile. The codes will be publicly available after my finals grades. My Github profile: https://github.com/AmitHasanShuvo/