Heaven's Light is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science and Engineering

Course no: CSE 4204
Course Title: Sessional Based on CSE 4203
Experiment no: 4
Name of the experiment:
Design and implementation of Kohonen Self-organizing neural network and Hopfield neural network.

Submitted by
Partho Kumar Rajvor
Roll: 1803119
Section: B
Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology

Submitted to
Rizoan Toufiq
Assistant Professor
Department of Computer Science and Engineering
Rajshahi University of Engineering and Technology

Contents

Chapter 3

Design and implementation of Kohonen Self-organizing neural network

3.1 Introduction

Kohonon Self-organizing neural network is a type of unsupervised learning algorithm. It is also known as Kohonen map or Self-organizing map. It is a type of artificial neural network that is trained using unsupervised learning to produce a low-dimensional, discretized representation of the input space of the training samples, called a map, and is therefore a method to do dimensionality reduction.

3.2 About the dataset

In this lab, we will be using the same dataset as we used in the previous lab.

3.2.1 Foreword

We will be using the following libraries in this lab:

- *pandas* for loading and preprocessing the dataset.
- *scikit-learn* for splitting the dataset into training and test sets and measuring performance metrics of the model.

3.2.2 Preprocessing the dataset

To ease the process of working with the dataset, we will specifically preprocess the *disagnosis* column of the dataset. We will replace the values M and B with 1 and 0 respectively. This will help us to work with the dataset more easily.

```
df['diagnosis'] = df['diagnosis']
                    .replace('M', 1)
df['diagnosis'] = df['diagnosis']
                    .replace('B', 0)
```

3.2.3 Selecting the features and the output

We will be using the first 30 columns of the dataset as the features and the last column as the output. We will use the following code snippet to select the features and the output:

```
x = df.iloc[:, 2:32]
y = df.iloc[:, 1]
```

3.3 Implementing the algorithm

3.3.1 Algotihm for Kohonen Self-organizing neural network

1. Initialize network weights. Define w_{ij} as the weight of the connection between the i^{th} and the j^{th} node.

2. Present input Preset input $x_0(t)$, $x_1(t)$, $x_2(t)$, ..., $x_n(t)$ to the network.

3. Calculate distances Compute the distance d_j between the input vector and the weight vector of each neuron.
   $d_j = \sum_{i=0}^{n}(x_i(t) - w_{ij}(t))^2$

4. Select minimum distance Designate the output node with minimum d_j to be j*.

5. Update weights Update weights for node j* and its neighbors, defined by the neighborhood size N_{j*}.
   New weights are
   $w_{ij}(t+1) = w_{ij}(t) + \eta(t)N_{j*}(t)[x_i(t) - w_{ij}(t)]$
   For all i and j where $\eta(t)$ is the learning rate.

3.3.2 Necessary imports

```
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix,
                            accuracy_score,
                            f1_score
```

3.3.3 Implementing the algorithm

```
class KSOFM:
    def __init__(
        self,
```

```
            num_input,
            num_output,
            learning_rate=0.1,
            epochs=100):
            self.num_input = num_input
            self.num_output = num_output
            self.learning_rate = learning_rate
            self.weights = np.random.rand(num_output, num_input)
            self.epochs = epochs

        def train(self, X):
            for i in range(self.epochs):
                for _, x in enumerate(X):
                    # print(x)
                    # print(self.weights)
                    # Calculate the distance between the
                    # input vector and each weight vector
                    distances =
                    np.linalg.norm(self.weights - x, axis=1)
                    # print(distances)
                    # Find the index of the winning neuron
                    winner =
                    np.argmin(distances)
                    # Update the weights of the winning neuron
                    self.weights[winner] +=
                    self.learning_rate *
                        (x - self.weights[winner])
                print(f'Epoch {i+1}, Weights {self.weights}')
            return self.weights

        def predict(self, X):
            y_pred = []
            for _, x in enumerate(X):
```

```
            distances =
            np.linalg.norm(self.weights - x, axis=1)
            winner = np.argmin(distances)
            y_pred.append(winner)
        return y_pred
```

3.3.4 Splitting the dataset into training and test sets

We will be using the *scikit-learn* library to split the dataset into training and test sets. We will use the following code snippet to split the dataset into training and test sets:

```
df = pd
    .read_csv('breast-cancer-wisconsin-data_data.csv')
df['diagnosis'].replace('M', 1, inplace=True)
df['diagnosis'].replace('B', 0, inplace=True)

X = df.iloc[:, 2:32]
y = df.iloc[:, 1]
X = np.array(X)
y = np.array(y)

X_train,
X_test,
y_train,
y_test = train_test_split(X, y, test_size=0.2)
```

3.4 Training the model and evaluating the performance

3.4.1 Training the model

We will use the following code snippet to train the model:

```
ksofm = KSOFM(30, 2, learning_rate=0.1, epochs=3000)
w = ksofm.train(X_train)
pred = ksofm.predict(X_test)
```

3.4.2 Accuracy of the model

We will use the following code snippet to calculate the accuracy of the model:

```
acc = accuracy_score(y_test, pred)
f1 = f1_score(y_test, pred)
conf_matrix = confusion_matrix(y_test, pred)
print(f'Accuracy: {acc}')
print(f'F1 Score: {f1}')
print(f'Confusion Matrix: \n{conf_matrix}')
```

3.4.3 Output

Accuracy: 0.9122807017543859
Confusion Matrix:

$$\begin{bmatrix} 78 & 1 \\ 9 & 26 \end{bmatrix}$$

F1 Score: 0.8387096774193549

3.5 Visualization of the results

3.5.1 Scatter plot of predicted and test data

We will be using two features to plot the scatter plot.

```
# scatter plot of the test data
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test)
plt.show()
# scatter plot of the predicted data
```

```
plt.scatter(X_test[:, 0], X_test[:, 1], c=pred)
plt.show()
```

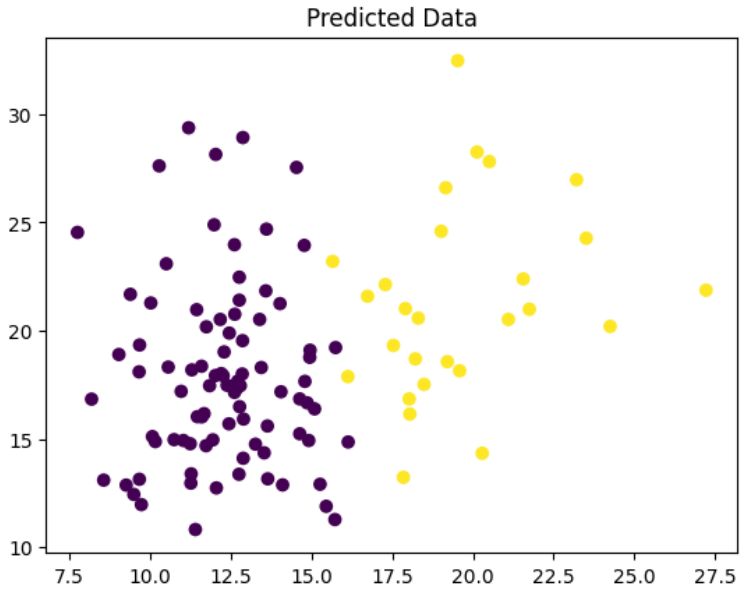

Figure 3.1: Scatter plot of the test data

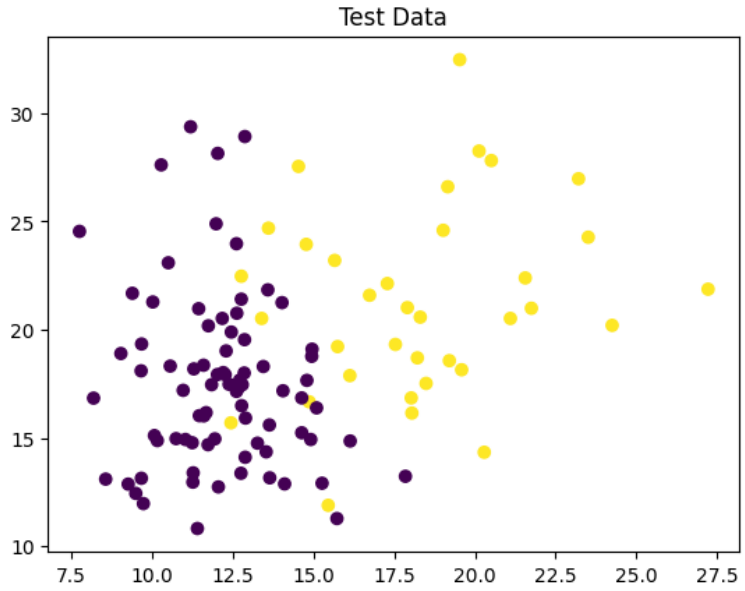

Figure 3.2: Scatter plot of the predicted data

3.5.2 Confusion Matrix

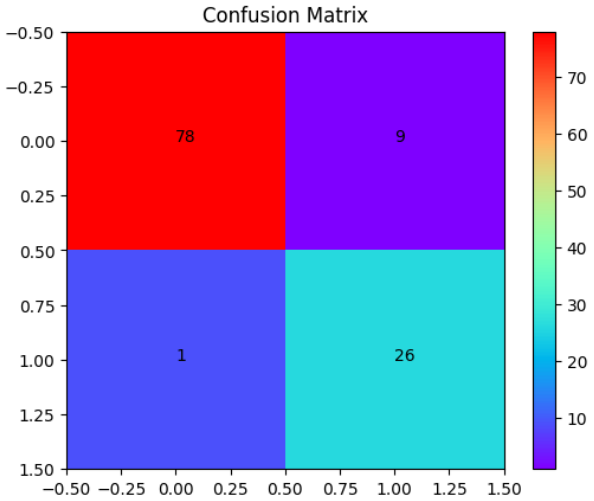

Figure 3.3: Confusion Matrix

3.6 Design and implementation of Hopfield neural network

3.7 Introduction

Hopfield neural network is a type of recurrent neural network. It is a form of associative memory. It is a fully connected network where each neuron is connected to every other neuron. It is a content addressable memory system with binary threshold units.

3.8 Implementing the algorithm

```
class HopfieldNetwork:
def __init__(self, n):
    self.n = n
    self.weights = []
    self.threshold = 0

def fit(self, X):
```

```
        self.weights = np.zeros((self.n, self.n))
        for x in X:
            self.weights += np.outer(x, x)
            print(self.weights)
        # self.weights /= self.n
        np.fill_diagonal(self.weights, 0)
        self.threshold = 0

    def predict(self, x):
        y = np.dot(self.weights, x) - self.threshold
        y[y >= 0] = 1
        y[y < 0] = -1
        return y
```

3.8.1 Training and testing the model

```
from random import randint
X = np.array([[1, -1, 1, -1, 1, 1, -1, 1],
              [1, 1, 1, -1, -1, -1, 1, 1]])
hn = HopfieldNetwork(8)
hn.fit(X)
pred = hn.predict([1, -1, 1, 0, 1, 1, -1, 0])
print(f'predicted pattern: {pred}')
```

3.8.2 Output

predicted pattern: [1. -1. 1. -1. 1. 1. -1. 1.]

3.9 Conclusion

In this lab, we have implemented the Kohonen Self-organizing neural network and the Hopfield neural network. We have also visualized the results of the Kohonen Self-organizing neural network.