Heaven's Light is Our Guide

Computer Science and Engineering

Rajshahi University of Engineering and Technology

Course No: CSE 4204

Course Title: Sessional Based on CSE 4203

No of Experiment: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.

Course Outcomes:CO1

Learning Domain with Level: Cognitive (Applying, Analyzing,Evaluating and Creating)

Submitted To

Rizoan Toufiq

Assistant Professor

Department of Computer Science and Engineering

Rajshahi University of Engineering and Technology

Submitted By

Name:Rafi Ahammed Songram

Roll:1803095

Department:Computer Science and Engineering

Rajshahi University of Engineering and Technology

[Implementation of Kohonen Neural Networks algorithm]

&

[Implementation of Hopfield Neural Networks algorithm]

Table 1: **Kohonen Neural Networks Algorithm**

Table 2: **Hopfield Neural Networks Algorithm**

Kohonen Self-organizing Neural Networks algorithm

The Kohonen Self-Organizing Neural Networks algorithm, also known as Kohonen maps or Self-Organizing Feature Maps, is a type of artificial neural network that can be used for unsupervised learning, clustering, and visualization of high-dimensional data. The Kohonen Self Organizing algorithm consists of a grid or lattice of neurons that are connected to each other. Each neuron represents a specific region in the input data space and has a weight vector that is initialized randomly at the beginning of the training process. During the learning process, the algorithm adjusts the weight vectors of the neurons in the grid to match the statistical properties of the input data.

The learning process of the Kohonen Self-Organizing algorithm can be divided into two phases: competitive and cooperative. In the competitive phase, the algorithm finds the neuron with the closest weight vector to the input data and updates its weight vector to be closer to the input data. This process is known as the winner-takes-all rule.

In the cooperative phase, the algorithm updates the weight vectors of neighboring neurons to be closer to the winner neuron. This process is known as the lateral inhibition rule. The idea is that neurons that are close to each other in the grid should have similar weight vectors and represent similar regions in the input data space.

Once the training process is complete, the Kohonen Self-Organizing algorithm can be used for various tasks, such as data visualization, clustering, and pattern recognition.

Steps of Kohonen Self-organizing Neural Networks algorithm:

Step-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 n inputs to the nodes to small random values. Set the initial radius of the neighborhood around node j $N_j(0)$, to be large.
Step-2: Present input x_0(t) ,x_1(t), x_2(t).....x_{n-1}(t) where x_i(t) is the in-

put to node i at time t.

Step-3: Calculate distance: Compute the distance between the input and each output node j, given by

$$d_j = f \sum_{i=0}^{n-1} (x_{it} - w_{ij})^2$$

Step-4: Select minimum distance Designate the output node with minimum d_j to be j*

Step-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) $= w_{ij}$ + η(t)$x_i(t) - w_{ij}(t)$

Step-6: Repeat by going to step 2

Dataset:

The name of the dataset is **Diabetes** and csv file diabetes.csv

Table 3: Attribute of Dataset

Column	Count	Type
1 pregnencies	768	int64
2.Glucose	768	int64
3.BloodPressure	768	int64
4.SkinThickness	768	int64
5.Insulin	768	int64
6.BMI	768	float64
7.Diabetespedigrefunction	768	float64
8.Age	768	int64
9.Outcome	768	int64

Here are the all attribute of full dataset and we apply on this Kohonen neurals Networks algorithm .

Outcome plot of dataset:

Code-snippet

```
ax = sns.countplot(df["diagnosis"],label="Count")
f, ax = plt.subplots(figsize=(2, 2))
sns.countplot(df, x="Outcome", ax=ax)
```

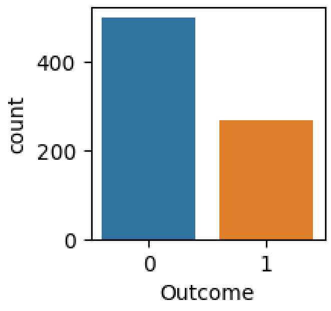

Figure-1: Dataset Outcome Count Plot

Feature-characteristics plot of dataset :

```
fig, ax = plt.subplots(4, 2, figsize=(6, 6))
for i in range(4):
for j in range(2):
feature = df.columns[i*2+j]
ax[i, j].hist(df[feature])
# ax[i, j].set_title(feature)
sk = skew(df[feature], bias=True)
print(sk)
ax[i, j].set_xlabel(f'{feature} [Skewness={sk:.2f}]')
ax[i, j].set_ylabel('Count')
fig.tight_layout()
```

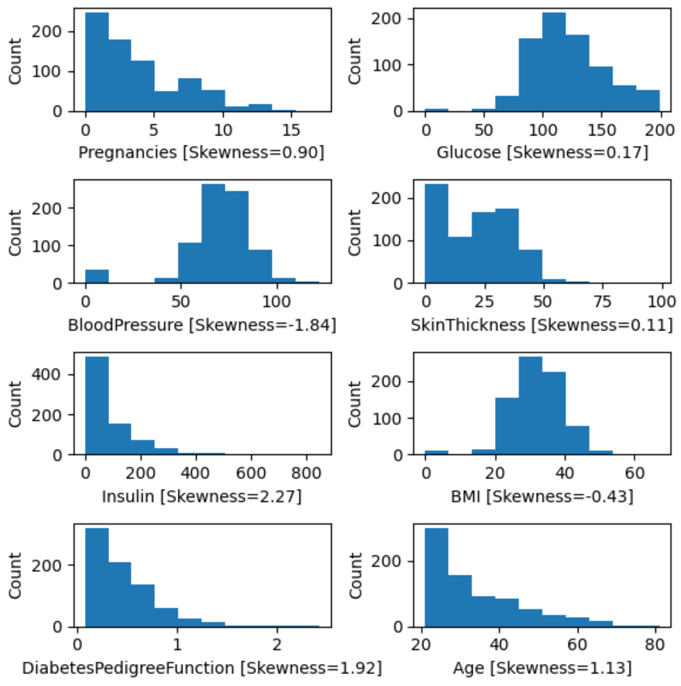

Figure-2: Dataset feature characteristics Plot

Kohonen self organizing Neurals Network Algorithm:

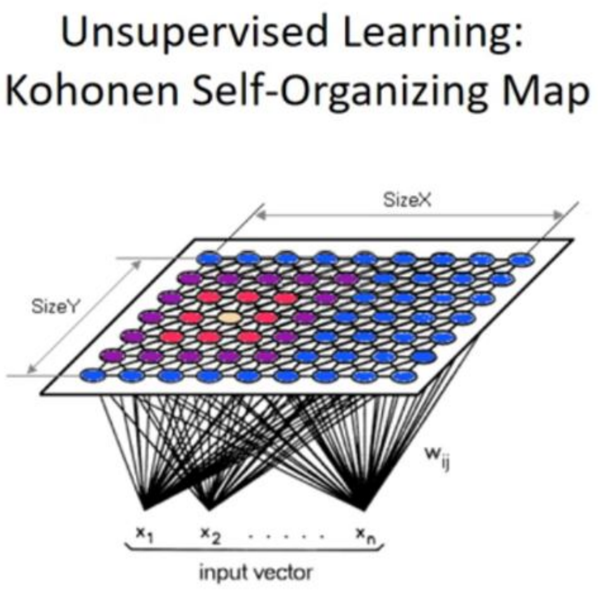

Figure-3: Kohonen Algorithm Map

Split Dataset

Separates the features and target variable, and then splits them into training and testing sets. The training set (Xtrain and Ytrain) is used to train a machine learning model, while the testing set (Xtest and Ytest) is used to evaluate the model's performance.Finally single lair perceptron algorithm will be applied .Here test size = 0.1

```
X_train, X_test,
y_train,y_test = train_test_split(X,y, test_size=0.1)
```

Kohonen Neural Network Algorithm

Weight vectors are central to the Kohonen Network, representing the parameters that evolve during training to capture the network's understanding of the input space. Each neuron in the network has an associated weight vector, and the values in these vectors determine the influence of the neuron in representing specific input patterns.The adaptation of weight vectors allows the network to organize information hierarchically, with different neurons specializing in different aspects of the input data.

Weight Update

Code Snippet:

```
def fit(self, X):
for i in range(self.epochs):
for idx, x in enumerate(X):
d = [np.linalg.norm(e) for e in x - self.weight]
winner = np.argmin(d)
self.weight[winner] += self.learning-rate *(x - self.weight[winner])
# print(self.weight)
```

Code Snippet:

```
class KSON:
def __init__(self, num_inputs, num_outputs,
learning_rate=0.01, epochs=5):

self.num_inputs = num_inputs
self.num_outputs = num_outputs
self.learning_rate = learning_rate
self.epochs = epochs
# init weights
self.weight = np.random.rand(self.num_outputs, self.num_inputs)
print(self.weight)
def fit(self, X):
for i in range(self.epochs):
# print(f'Epoch {i+1}')
for idx, x in enumerate(X):
d = [np.linalg.norm(e) for e in x - self.weight]
winner = np.argmin(d)
self.weight[winner] += self.learning_rate * (x - self.weight[winner])
# print(self.weight)
def predict(self, X):
pred = []
for idx, x in enumerate(X):
d = [np.linalg.norm(e) for e in x - self.weight]
winner = np.argmin(d)
pred.append(winner)
return np.array(pred)
```

Model Fit

Finally model was fitted and no of epoch is 5 and accuracy was found of this model on this dataset

Code-Snippet:

```
X = df.iloc[:, 0:8]
```

```
y = df.iloc[:, 8]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.
X_train = np.array(X_train)
X_test = np.array(X_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
kson = KSON(8, 2,1)
kson.fit(X_train)
```

Evaluate Model and Accuracy

Kohonen Algorithm accuracy is- Code-Snippet:

```
kson = KSON(8, 2,1)
kson.fit(X_train)
pred = kson.predict(X_test)
print(pred)
print(y_test)
print(accuracy_score(y_test, pred))
```

Accuracy: 0.7802597402597403

Evaluation(Confusion) Matrics

confusion matrix visualize and summarize the performance of this trained model performance on trainning and testing and code seems correct and should display a heatmap with annotations for True Positives, True Negatives, False Positives, and False Negatives.
Confusion Matrix:

```
import seaborn as sn
import pandas as pd
from sklearn.metrics import confusion_matrix
import seaborn as sns
cm = confusion_matrix(y_test, pred)
sns.heatmap(cm, annot=True)
plt.show()
plt.figure(figsize = (5,5))
```

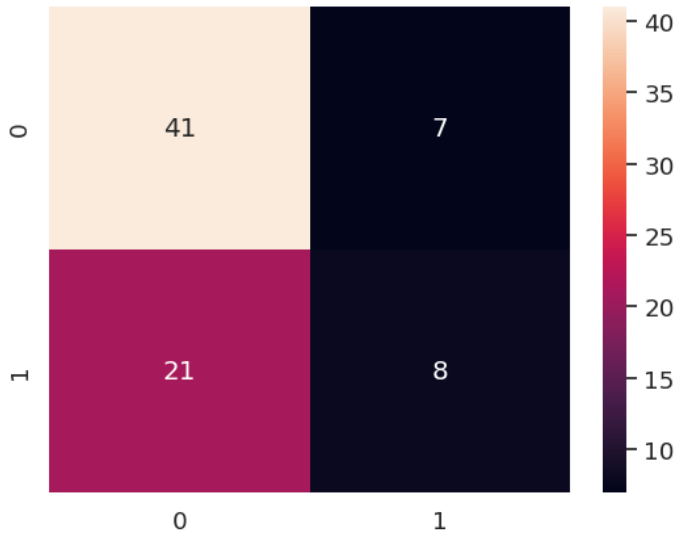

Figure-4:Confusion Matrix

Limitations:

1.Missing Value: It can't handle missing value well in data.

2.Computational Expensive:This is computationally expensive which is a major drawback. Because if the dimensions of the data increases, the computation increases a lot with it.

3.Global Information Loss: The algorithm may suffer from global information loss during the mapping process. Neurons in the output layer may represent clusters of data points, but detailed information about individual data points may be lost. .

4.Initialization Dependency: The performance of the algorithm can be influenced by the initial random weights of neurons. Different initializations may lead to different outcomes, and finding a suitable initialization can be challenging..

Improvements

Here are some improvements and considerations for enhancing the performance and capabilities of a Kohonen Neural Network:

1. Dataset is not enough good. Better dataset can give better result.
2.Adaptive learning Rates
3.By Batch training

Conclusion

The Kohonen Self-Organizing Map (SOM) algorithm is a powerful tool for unsupervised learning, particularly in visualizing and organizing high-dimensional data. Despite its limitations, advancements such as adaptive learning rates and topology preservation techniques contribute to its effectiveness. Careful parameter tuning and hybrid approaches further enhance its utility in diverse applications.

References

[1] Neural Computing: An Introduction by R-Beale and T-Jackson

[2] Kohonen Self Organizing Feature map and its use of clustering Markus TörmäInstitute of Photogrammetry and Remote SensingHelsinki University of TechnologyEspoo, Finlandmarkus@mato.hut.fu)

Hopfield Network algorithm:

The Hopfield Network is a type of artificial neural network that can be used for pattern recognition, optimization, and associative memory tasks. The basic idea behind the Hopfield Network is that it consists of a set of neurons that are fully connected to each other, meaning that each neuron is connected to every other neuron in the network.These neurons can either be in an "on" or "off" state, and they interact with each other through weighted connections.

The learning process in a Hopfield Network involves adjusting the synaptic weights based on the patterns presented to the network. The Hebbian learning rule, which strengthens connections between neurons that are simultaneously active,is employed. This process enables the network to store and retrieve patterns efficiently.

Hopfield Networks typically use binary neurons, where each neuron can be in one of two states (usually +1 or -1). This binary representation simplifies the mathematics involved in weight updates and network dynamics.

Hopfield Networks find applications in various domains, including optimization problems, image recognition, pattern fix and associative memory tasks. Their ability to retrieve patterns from partial or degraded inputs makes them suitable for real-world scenarios with imperfect data.

Steps of Hopfield Neural Networks algorithm:

Step-1: Assign Connection Weights:

$$w_{ij} = \sum_{s=0}^{M-1} (x_i)^s (x_j)^s$$

if i $\neq$ j
then 0 if i= 0 and 0$\leq$ i,j $\leq$ M-1
Step-2: Initialise with unknown pattern μ(0) $= x_i$ 0$\leq$ i $\leq$ N-1

Step-3: Iterate Until Convergence:

$$\mu(i+1) = f_h \sum_{i=0}^{N-1} (w_{ij})(\mu_j)$$

The function fh is the hard-limiting non-linearity, the step function. Repeat the iteration until the outputs from the nodes remain unchanged.

Hopfield Neural Networks algorithm:

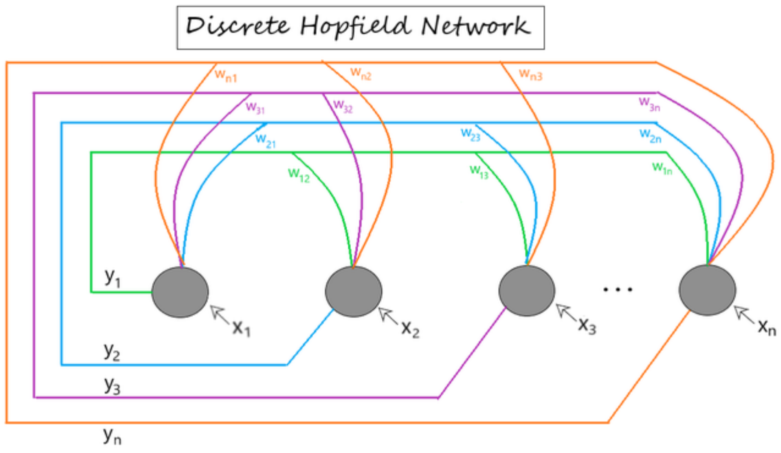

Figure-5:Hopfield Network Map

Training Data

The dataset used in this code consists of randomly generated binary patterns representing neural network inputs.The training set (training patterns) is composed of M patterns, each with N binary values, where M and N arepredefined parameters. The Hopfield network is trained on these patterns to learn the associated connection weights. Subsequently, a test pattern (test pattern) is generated to assess the network's recall capability. The network's iterative recall process is visualized, showing the original test pattern, the expected output pattern, and the actual output pattern obtained from the Hopfield network.

Training Data:
[-1, -1, 1, -1, 1, 1,-1]
[1, -1, 1, -1, 1, -1, 1]
[-1, 1, -1, 1, -1 , 1,-1]
[1, 1 , 1, -1, -1 , 1, -1]
[1, 1, -1, -1, 1, -1, 1]

Input Pattern: [-1, 1, 1, -1, -1, 1,-1]

Hopfield Network Code Snippet

Code Snippet:

```
class HopfieldNetwork:
def __init__(self, num_inputs):
self.num_inputs = num_inputs
self.X = None
self.weight = np.zeros((self.num_inputs, self.num_inputs))
# print(self.weight)
def fit(self, X):
self.X = X
# print(self.X)
for pattern in X:
self.weight += np.dot(pattern.reshape(-1, 1), pattern.reshape(1, -1))
np.fill_diagonal(self.weight, 0)
# print(self.weight)
def predict(self, inp):
inp_copy = copy.deepcopy(inp)
for i in range(5):
print(f'Iteration {i+1}')
fixed = np.dot(inp, self.weight)
fixed[fixed >= 0] = 1
fixed[fixed < 0] = -1
if np.array_equal(inp, fixed):
print("Found")
print(f'Input: {inp_copy}')
print(f'Fixed: {fixed}')
break
inp = fixed
print(inp)
hp = HopfieldNetwork(7)
X = np.array([
    [-1, -1, 1, -1, 1,1,-1],
    [1, -1, 1, -1, 1,-1,1],
    [-1, 1, -1, 1, -1 , 1,-1],
    [1, 1 ,1, -1, -1,1,-1],
    [1, 1,-1, -1, 1,-1,1]
], dtype='float32')
hp.fit(X)
hp.predict([-1, 1, 1, -1, -1, 1,-1])
```

Iteration & Pattern Fix

```
Output: Iteration 1
[-1. 1. 1. 1. -1. 1. -1.]
Iteration 2
[-1. 1. -1. 1. -1. 1. -1.]
Iteration 3
Found
Input: [-1, 1, 1, -1, -1, 1, -1]
Fixed: [-1. 1. -1. 1. -1. 1. -1.]
```

Advantages:

1.Pattern Completion: They can complete or reconstruct distorted patterns, making them robust in handling incomplete or noisy data.
2.No Supervised Training: Training is unsupervised, meaning there is no need for a target output during the training phase.
3.Parallel Processing: The network operates in a parallel processing manner, allowing for efficient implementation on parallel computing architectures.

Limitations:

1.Capacity Limitation: Hopfield networks have limited storage capacity, and the number of patterns they can reliably store is bounded by a fraction of the total number of nodes.
2. Fixed Weights: The weights are fixed once the network is trained, limiting the ability to adapt to changing environments or learn new patterns dynamically.
3.Not Suitable for Complex Tasks: Hopfield networks are not well-suited for complex tasks such as image recognition in high-dimensional spaces, where more sophisticated neural network architectures like convolutional neural networks (CNNs) are often more effective.

Conclusion:

In conclusion, Hopfield networks offer valuable features such as associative memory, pattern completion, and a simple learning rule. Their ability to handle incomplete or noisy data, coupled with unsupervised training, makes them suitable for various applications. However, it's essential to consider their limitations, including capacity constraints, sensitivity to symmetrical patterns, and the fixed nature of weights.While Hopfield networks have proven effective for certain tasks, particularly in small to moderately sized problems, they may not be the optimal choice for more complex tasks, such as high-dimensional image recognition. As technology advances, neural network.

References

[1] Neural Computing: An Introduction by R-Beale and T-Jackson