# Rajshahi University of Engineering & Technology

**Heaven’s Light is Our Guide**

**Course Code**: CSE 4204
**Course Title**: Sessional based on CSE 4203
**Experiment no.**: 04


**Submitted by:**

**Name**: Ayan Sarkar
**Roll**: 1903162
**Section**: C
**Department**: CSE

**Submitted to:**

Md. Mazharul Islam
Lecturer,
Department of CSE,
RUET

**Problem:** Self-Organizing Map (SOM) Implementation using the Kohonen Algorithm.

**Experiment Description:** This experiment demonstrates the implementation of a self-organizing map (SOM) using the Kohonen algorithm. The primary goal is to map high-dimensional data into a lower-dimensional (in this case, 2D) grid while preserving the topological properties of the input space. The procedure involves:

- **Weight Initialization:** Random weights are assigned to each neuron in a 3×3 grid.
- **Input Data Presentation:** A set of 2D data points is provided as the input.
- **Best Matching Unit (BMU) Identification:** For every input vector, the neuron with the minimum Euclidean distance (the BMU) is identified.
- **Weight Update:** The weights of the BMU and its neighboring neurons are updated, moving them closer to the input vector. This update is controlled by a learning rate and influenced by the neighborhood radius.
- **Parameter Decay:** Both the learning rate and the neighborhood radius decrease exponentially over the epochs, allowing for rapid initial adjustments that gradually fine-tune the network.
- **Visualization:** The evolving positions of neurons are visualized using scatter plots at regular intervals to monitor training progress.

This structured approach enables the SOM to learn the underlying topology of the dataset, making it an effective unsupervised learning tool for clustering and visualization tasks.

**Dataset Characteristic:**

**Input Data:** 8 synthetic 2D points with the following coordinates:

[1, 8], [5, 2], [9, 7], [4, 5], [1, 1], [2, 2], [3, 2], [3, 3]

These points are manually specified and serve as a simple yet effective demonstration of how the SOM organizes data based on spatial relationships. The limited number of data points allows clear observation of how neurons adjust their positions relative to the clusters in the dataset.

**Code:**

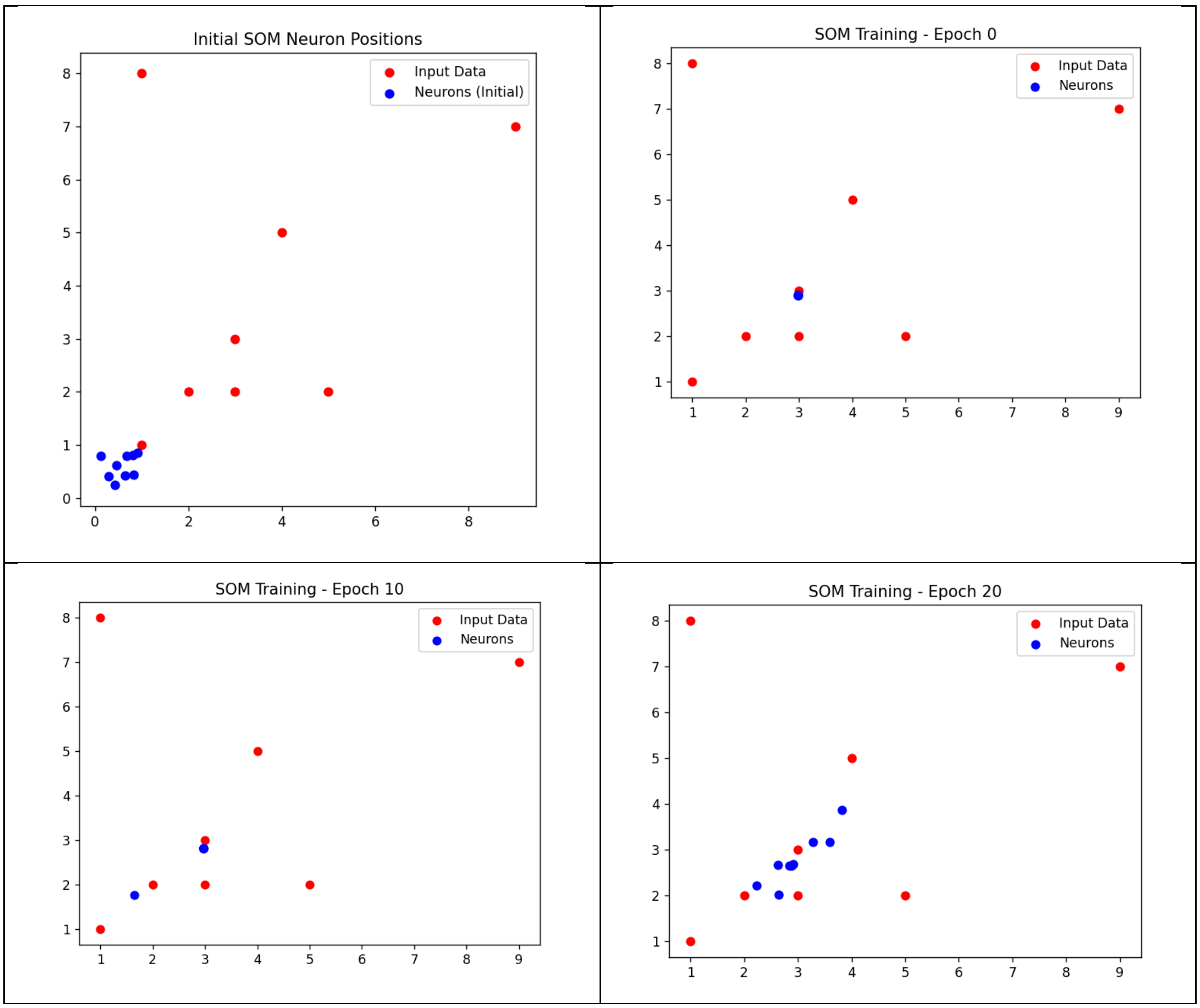
```python
import numpy as np
import matplotlib.pyplot as plt

# Step 1: Initialize weights
def initialize_weights(grid_size, input_dim):
    return np.random.rand(grid_size, grid_size, input_dim)  # Random weights
```

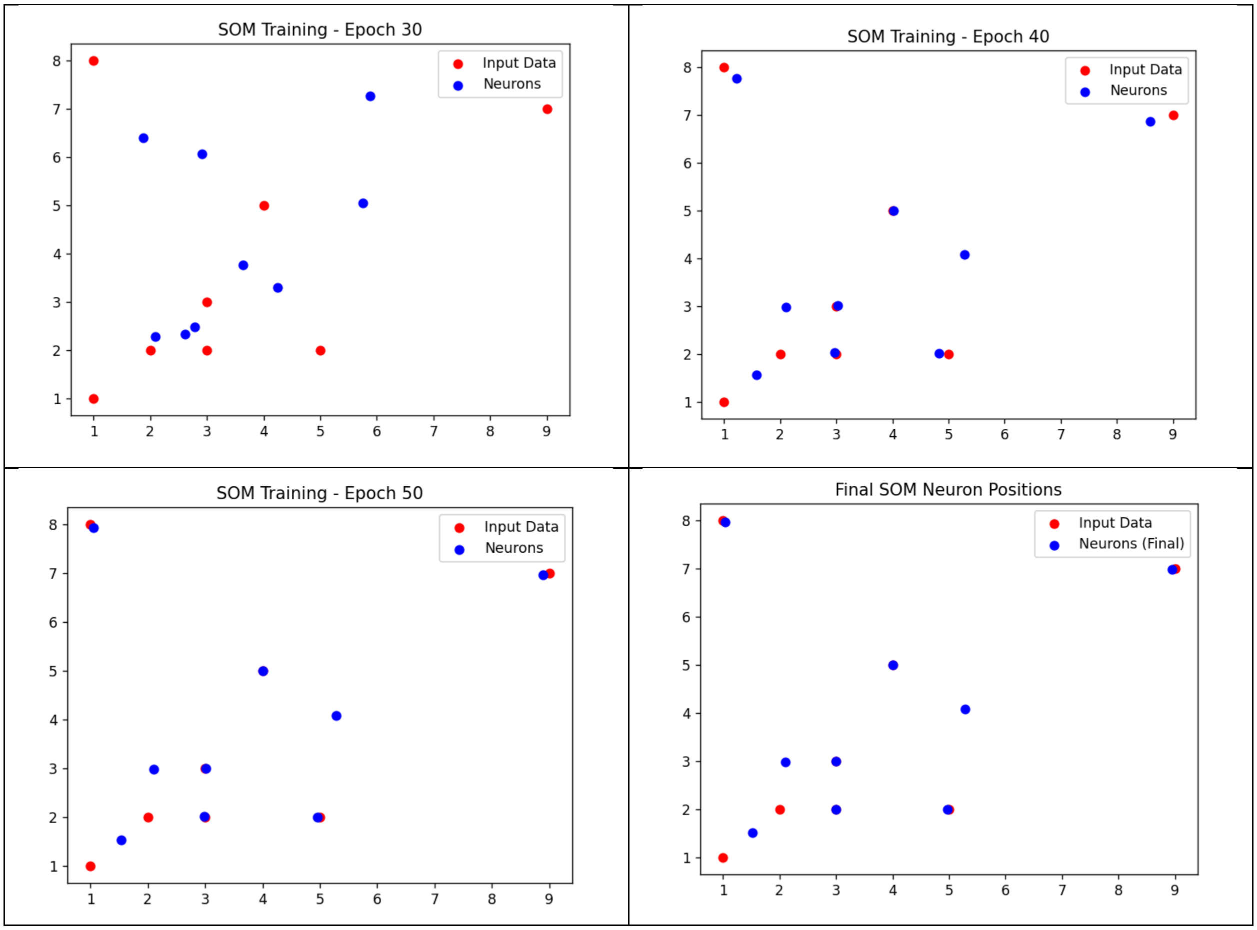
```python
# Step 2: Input pattern (Example: 2D points)
input_data = np.array([[1, 8], [5, 2], [9, 7], [4, 5], [1, 1], [2, 2], [3, 2], [3, 3]])

# SOM Parameters
grid_size = 3   # 3x3 grid of neurons
input_dim = 2   # Each input has 2 features
epochs = 70     # Number of iterations
learning_rate = 0.9
neighborhood_radius = 3
# Initialize weight matrix
weights = initialize_weights(grid_size, input_dim)

# Step 3: Find Best Matching Unit (BMU)
def find_bmu(weights, input_vector):
    distances = np.linalg.norm(weights - input_vector, axis=2)  # Euclidean distance
    bmu_index = np.unravel_index(np.argmin(distances), distances.shape)  # Find BMU
    return bmu_index
# Step 4: Update BMU and its neighbors
def update_weights(weights, bmu, input_vector, learning_rate, neighborhood_radius):
    grid_x, grid_y, _ = weights.shape
    for x in range(grid_x):
        for y in range(grid_y):
            distance_to_bmu = np.linalg.norm(np.array([x, y]) - np.array(bmu))
            if distance_to_bmu <= neighborhood_radius:
                # influence = np.exp(-distance_to_bmu**2 / (2 * (neighborhood_radius**2)))  # Gaussian function
                # weights[x, y] += influence * learning_rate * (input_vector - weights[x, y])

                weights[x, y] += learning_rate * (input_vector - weights[x, y])
    return weights
# Step 5: Update learning rate over time
def decay_learning_rate(initial_lr, epoch, t1= 500):
    return initial_lr * np.exp(-epoch / t1)

# Step 6: Update learning rate over time
def decay_neighbourhood(initial_lr, epoch, t2 = 500):
    return initial_lr * np.exp(-epoch / t2)
# Visualization
plt.figure(figsize=(6, 6))

# Initial state
plt.scatter(input_data[:, 0], input_data[:, 1], color='red', label='Input Data')
plt.scatter(weights[:, :, 0], weights[:, :, 1], color='blue', label='Neurons (Initial)')
plt.legend()
plt.title("Initial SOM Neuron Positions")
plt.show()
# Training Loop
for epoch in range(epochs):
    for input_vector in input_data:
        bmu = find_bmu(weights, input_vector)  # Step 3: Find BMU
        weights = update_weights(weights, bmu, input_vector, learning_rate, neighborhood_radius)  # Step 4: Update BMU
    learning_rate = decay_learning_rate(learning_rate, epoch)  # Step 5: Update learning rate
    neighborhood_radius = decay_neighbourhood(neighborhood_radius, epoch)  # Step 6: Update neighbourhood

```

```python
    # Plot training progress every 10 epochs
    if epoch % 10 == 0 or epoch == epochs - 1:
        print("Learning rate: ", learning_rate)
        print("Neighbour radius: ", neighborhood_radius)
        print()

        plt.scatter(input_data[:, 0], input_data[:, 1], color='red', label='Input Data')
        plt.scatter(weights[:, :, 0], weights[:, :, 1], color='blue', label='Neurons')
        plt.title(f"SOM Training - Epoch {epoch}")
        plt.legend()
        plt.show()
```

```python
# Final state
plt.scatter(input_data[:, 0], input_data[:, 1], color='red', label='Input Data')
plt.scatter(weights[:, :, 0], weights[:, :, 1], color='blue', label='Neurons (Final)')
plt.legend()
plt.title("Final SOM Neuron Positions")
plt.show()
```

## Visualization:






**Result:**

During the training process, the SOM undergoes 70 epochs of iterative updates. Key observations include:

- **Dynamic Adaptation:** Initially, neuron weights are random. However, as training progresses, they gradually shift closer to the clusters of input data.
- **Effective Convergence:** The exponential decay of both the learning rate and neighborhood radius ensures that the network makes large adjustments in the early stages and fine-tunes in later stages, leading to stable convergence.

These results confirm that the SOM algorithm can successfully capture and represent the inherent structure within the dataset, facilitating effective clustering and data visualization.

**Conclusion:** The experiment confirms the Kohonen algorithm's effectiveness in organizing a set of 2D input vectors into a topological map. The final visualization shows that neurons cluster around and between the key data points. Notably, some neurons remain positioned between (1,1) and (2,2), indicating the network has finely adapted to local variations in that region. Overall, the SOM successfully captures the underlying structure of the input data and offers a useful tool for unsupervised learning tasks such as clustering and dimensionality reduction.