UNIT 4

Self-Organization Maps (SOM):

CONTENTS

- Introduction
- Two Basic Feature Mapping Models
- SOM Algorithm
- Properties of Feature Maps
- Computer Simulations
- Learning Vector Quantization (LVQ)
- Adaptive Pattern Classification
- Comparison Table

Self-Organization Maps (SOM): Introduction

- **Purpose**
- **Dimensionality Reduction:** Simplifies complex datasets into a 2D representation that is easier to interpret.
- **Clustering:** Groups similar data points together based on feature similarity.
- **Visualization:** Provides interpretable maps for exploring and analyzing high-dimensional data.

Basic Feature Mapping Models

Rectangular Grid Topology

- Neurons are arranged in a **rectangular lattice**.
- **Advantages:**
- Simple structure
- Easy to implement and visualize
- **Limitations:**
- Neighborhood preservation is less natural
- Can lead to **distortion** in mapping relationships

Basic Feature Mapping Models

Hexagonal Grid Topology

- Neurons are arranged in a **hexagonal lattice**.
- **Advantages:**
- Better **neighborhood preservation**
- Reduces distortion in feature mapping
- Each neuron has **six equidistant neighbors**, improving smoothness of the map
- **Limitations:**
- Slightly more complex to implement

SOM Algorithm STEPS

Initialization

- Each neuron in the 2D grid is assigned a random weight vector.
- These weights have the same dimension as the input data.

Input Selection

- A sample data point is chosen randomly from the dataset.

Best Matching Unit (BMU) Identification

- Compute the distance (usually Euclidean) between the input vector and all neurons' weight vectors.
- The neuron with the smallest distance is the **BMU**.

SOM Algorithm STEPS

Neighborhood Function

- Define a neighborhood around the BMU.
- Neurons close to the BMU are considered neighbors and will also update their weights.

Weight Update Rule

- Update the BMU and its neighbors using:

- $w(t+1)=w(t)+\alpha(t)\cdot hBMU(t)\cdot(x-w(t))$
- where: $w(t)$ = weight vector at time t
- $\alpha(t)$ = learning rate (decreases over time)
- $hBMU(t)$ = neighborhood function (shrinks over time)
- x = input vector

Iteration

- Repeat steps 2–5 for many epochs.
- Over time, the map organizes itself so that similar inputs are mapped to nearby neurons.

Properties of feature map

Property	Description	Benefit
Approximation	Neurons approximate the input distribution	Compact representation of data
Topological Ordering	Preserves neighborhood relationships in the grid	Maintains similarity structure
Density Matching	Neuron density reflects input data density	Better representation of frequent patterns
Feature Selection	Highlights key distinguishing features	Useful for clustering & visualization

Computer Simulations

```
class SOM:

    # Function here computes the winning vector
    # by Euclidean distance
    def winner(self, weights, sample):

        D0 = 0
        D1 = 0

        for i in range(len(sample)):

            D0 = D0 + math.pow((sample[i] - weights[0][i]), 2)
            D1 = D1 + math.pow((sample[i] - weights[1][i]), 2)

        # Selecting the cluster with smallest distance as winning cluster

        if D0 < D1:
            return 0
        else:
            return 1

    # Function here updates the winning vector
    def update(self, weights, sample, J, alpha):
        # Here iterating over the weights of winning cluster and modifying them
        for i in range(len(weights[0])):
            weights[J][i] = weights[J][i] + alpha * (sample[i] - weights[J][i])

        return weights

# Driver code
```

Computer Simulations

```
def main():

    # Training Examples ( m, n )
    T = [[1, 1, 0, 0], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 1]]

    m, n = len(T), len(T[0])

    # weight initialization ( n, C )
    weights = [[0.2, 0.6, 0.5, 0.9], [0.8, 0.4, 0.7, 0.3]]

    # training
    ob = SOM()

    epochs = 3
    alpha = 0.5

    for i in range(epochs):
        for j in range(m):

            # training sample
            sample = T[j]

            # Compute winner vector
            J = ob.winner(weights, sample)

            # Update winning vector
            weights = ob.update(weights, sample, J, alpha)

    # classify test sample
    s = [0, 0, 0, 1]
    J = ob.winner(weights, s)

    print("Test Sample s belongs to Cluster : ", J)
    print("Trained weights : ", weights)


if __name__ == "__main__":
    main()
```

Computer Simulations

- Cluster : 0
- Trained weights : [[0.6000000000000001, 0.8, 0.5, 0.9], [0.3333984375, 0.0666015625, 0.7, 0.3]]

Learning Vector Quantization

Learning Vector Quantization (LVQ) is a *supervised classification algorithm* that uses representative prototypes (codebook vectors) to classify input data.

It works by adjusting these prototypes during training .

They best represent different classes, making it a bridge between neural networks and nearest-neighbor methods.

Learning Vector Quantization

How LVQ Works

- **Initialization:** Select one or more training samples per class as starting prototypes.
- **Distance calculation:** For each input, compute the distance (usually Euclidean) to all prototypes.
- **Winner selection:** Identify the closest prototype (the “winner”).
- **Prototype update:**
  - If the winner’s class matches the input label → move prototype closer.
  - If mismatched → move prototype away.
- **Iteration:** Repeat until prototypes stabilize and represent their classes well.

Learning Vector Quantization

Method	Learning Type	Representation	Strengths	Weaknesses
LVQ	Supervised	Prototypes	Simple, interpretable, efficient	Sensitive to initialization
Vector Quantization (VQ)	Unsupervised	Codebooks	Good for compression	No class labels
Self-Organizing Maps (SOM)	Unsupervised	Grid of neurons	Good visualization	No direct classification
k-Nearest Neighbors (k-NN)	Supervised	Training samples	High accuracy with enough data	Computationally heavy

Learning Vector Quantization

- **Key Advantages**
- **Interpretable:** Easy to understand since prototypes represent classes.
- **Efficient:** Faster than k-NN because only prototypes are stored.
- **Flexible:** Can adapt to complex decision boundaries with enough prototypes.

Limitations

- **Initialization sensitivity:** Poor starting prototypes can lead to bad classification.
- **Not optimal for high-dimensional data:** Struggles with very complex datasets.
- **Requires careful tuning:** Learning rate and number of prototypes must be chosen wisely.

Adaptive Pattern Classification

Adaptive Pattern Classification:

- Refers to machine learning techniques that dynamically adjust their internal parameters to improve pattern recognition over time.
- These systems learn from data and adapt to changes, making them powerful tools for tasks like speech recognition, image classification, and predictive analytics.

Key Concepts in Adaptive Pattern Classification

- **Adaptivity:** The system updates its model based on new data or feedback.
- **Pattern Recognition:** Identifies regularities or structures in input data.
- **Supervised or Unsupervised Learning:** Can be guided by labeled data or discover patterns on its own.
- **Real-time Learning:** Some adaptive systems can learn continuously as new data arrives.

Examples of Adaptive Pattern Classification Algorithms

Algorithm	Adaptivity Type	Use Case
Learning Vector Quantization (LVQ)	Prototype adjustment	Image and signal classification
Perceptron	Weight updates	Binary classification
Adaptive Resonance Theory (ART)	Stable learning with plasticity	Clustering and recognition
Self-Organizing Maps (SOM)	Topology-preserving mapping	Data visualization
Reinforcement Learning	Policy updates via rewards	Game playing, robotics

Adaptive Pattern Classification

Benefits of Adaptive Classification

- **Handles non-stationary data:** Useful when data distributions change over time.
- **Improves accuracy:** Learns from mistakes and refines predictions.
- **Scalable:** Can be applied to large and complex datasets.

Thank You..