# Practical 7A

**# A. Aim: Write a program for Linear separation in python.**

import numpy as np

# # Perceptron class for linear separation

class Perceptron:

def init (self, num\_features): self.weights = np.zeros(num\_features) self.bias = 0.0

def predict(self, inputs):

weighted\_sum = np.dot(self.weights, inputs) + self.bias return 1 if weighted\_sum >= 0 else 0

def train(self, training\_data, labels, learning\_rate, epochs): for epoch in range(epochs):

for i in range(len(training\_data)): inputs = training\_data[i]

label = labels[i]

prediction = self.predict(inputs) if prediction != label:

update = (label - prediction) \* learning\_rate self.weights += update \* inputs

self.bias += update

# # Example usage for linear separation

if name == " main ":

# # Define training data and labels for a simple linearly separable problem

training\_data = np.array([[1, 2], [2, 3], [3, 1], [4, 4], [5, 5], [6, 4]])

labels = np.array([0, 0, 0, 1, 1, 1])

**# Create a Perceptron instance** num\_features = training\_data.shape[1] perceptron = Perceptron(num\_features)

# # Train the Perceptron with the training data

learning\_rate = 0.1

epochs = 100

perceptron.train(training\_data, labels, learning\_rate, epochs) **# Test the trained Perceptron with new data points** test\_data = np.array([[2, 2], [4, 3], [5, 6]])

for data\_point in test\_data:

prediction = perceptron.predict(data\_point) print(f"Prediction for {data\_point}: Class {prediction}")

**Output:**

Prediction for [2 2]: Class 0

Prediction for [4 3]: Class 1

Prediction for [5 6]: Class 1

# Practical 7B

**# B. Aim: Write a program for Hopfield network model for associative memory**

import numpy as np class HopfieldNetwork:

def init (self, num\_neurons): self.num\_neurons = num\_neurons

self.weights = np.zeros((num\_neurons, num\_neurons)) def train(self, patterns):

num\_patterns = len(patterns) for pattern in patterns:

pattern = np.array(pattern)

self.weights += np.outer(pattern, pattern) np.fill\_diagonal(self.weights, 0) self.weights /= num\_patterns

def energy(self, state):

return -0.5 \* np.dot(state, np.dot(self.weights, state)) def update\_rule(self, state):

h = np.dot(self.weights, state) return np.where(h >= 0, 1, -1)

if name == " main ":

patterns = [[1, 1, -1, -1], [-1, -1, 1, 1]]

num\_neurons = len(patterns[0])

hopfield\_net = HopfieldNetwork(num\_neurons) hopfield\_net.train(patterns)

test\_patterns = [[1, 1, 1, -1], [-1, 1, 1, 1]] for pattern in test\_patterns:

initial\_state = np.array(pattern) iterations = 5

print(f"Initial state: {initial\_state}") for i in range(iterations):

new\_state = hopfield\_net.update\_rule(initial\_state) print(f"Iteration {i + 1}: {new\_state}")

if np.array\_equal(new\_state, initial\_state): print("Converged to a stable state.") break

initial\_state = new\_state

**Output:**

Initial state: [ 1 1 1 -1]

Iteration 1: [ 1 1 -1 -1]

Iteration 2: [ 1 1 -1 -1] **Converged to a stable state.** Initial state: [-1 1 1 1]

Iteration 1: [-1 -1 1 1]

Iteration 2: [-1 -1 1 1]

**Converged to a stable state.**