Exp: 1 **Date: 02/08/2024**

1.Implement AND function using Perceptron

Aim:

The aim of this experiment is to implement a single-layer perceptron using MATLAB to perform binary classification, specifically to model the behavior of the AND logic gate. The perceptron will be trained on a data set of binary inputs and their corresponding outputs, and then tested to ensure it correctly classifies the inputs.

Algorithm:

1. **Initialization**

- **Input Data**: Define the input data and the corresponding output labels for the AND logic gate.
- **Weights and Bias**: Initialize the weights and bias randomly.
- **Learning Rate**: Set a learning rate (a small positive number) that controls the step size during the weight update.
- **Epochs**: Define the number of epochs (iterations over the entire dataset).

2. **Training the Perceptron**

- For each epoch (repeat the following steps for a specified number of epochs):
  - For each training example (input-output pair) in the dataset:
    1. **Input Representation**: Represent the input vector xxx and the corresponding output yyy.
    2. **Weighted Sum Calculation**: Compute the weighted sum of the inputs plus the bias. weighted_sum=w· x+b\text{weighted_sum} = w \cdot x + bweighted_sum=w· x+b
    3. **Activation Function**: Apply the step activation function to the weighted sum to get the predicted output y^\hat{y}y^. y^=step(weighted_sum)\hat{y} = \text{step}(\text{weighted_sum})y^=step(weighted_sum)
    4. **Weight and Bias Update**: Adjust the weights and bias based on the error (difference between the actual output yyy and the predicted output y^\hat{y}y^).
       - Update rule: w=w+lr×(y−y^)×xw = w + \text{lr} \times (y - \hat{y}) \times xw=w+lr×(y−y^)×x b=b+lr×(y−y^)b = b + \text{lr} \times (y - \hat{y})b=b+lr×(y−y^)
       - Where lr\text{lr}lr is the learning rate.

3. **Testing the Perceptron**

- For each test input:
  1. **Input Representation**: Represent the input vector xxx.
  2. **Weighted Sum Calculation**: Compute the weighted sum of the inputs plus the bias using the trained weights and bias.
  3. **Activation Function**: Apply the step activation function to determine the output.
  4. **Display Output**: Print the input and the corresponding output.

4. **Output and Conclusion**

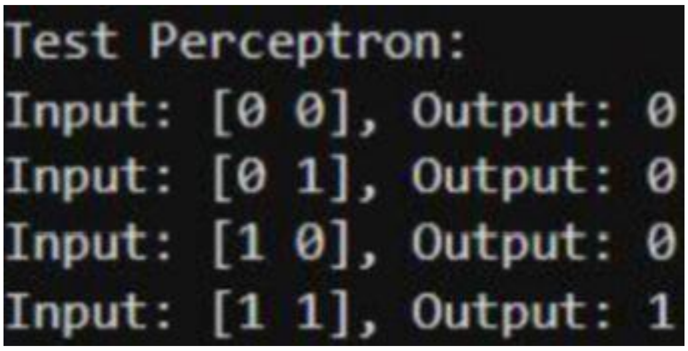
- The perceptron should correctly classify all the inputs according to the AND logic gate rule: output 1 if both inputs are 1, otherwise output 0.
- Conclude the experiment by discussing the accuracy of the perceptron and any observations made during training and testing.

Program:

```
% Clear workspace and command window
clear; clc;
% Training data (AND logic gate)
% Inputs (x1, x2) and corresponding output (y)
inputs = [0 0; 0 1; 1 0; 1 1];
outputs = [0; 0; 0; 1];
% Initialize weights and bias
weights = rand(2,1); % Random initialization
bias = rand();
% Learning rate
lr = 0.1;
% Number of epochs
epochs = 1000;
% Training the perceptron
for epoch = 1:epochs
    for i = 1:size(inputs, 1)
        x = inputs(i, :)';
        y = outputs(i);
        % Calculate weighted sum
        weighted_sum = weights' * x + bias;
        % Activation (Step function)
        y_pred = double(weighted_sum >= 0);
        % Update weights and bias
        weights = weights + lr * (y - y_pred) * x;
        bias = bias + lr * (y - y_pred);
    end
end
% Test the perceptron
disp('Test Perceptron:');
test_inputs = [0 0; 0 1; 1 0; 1 1];
for i = 1:size(test_inputs, 1)
```

```
    x = test_inputs(i, :)';
    weighted_sum = weights' * x + bias;
    y_pred = double(weighted_sum >= 0);
    fprintf('Input: [%d %d], Output: %d\n', x(1), x(2), y_pred);
end
```

Output:



```
Test Perceptron:
Input: [0 0], Output: 0
Input: [0 1], Output: 0
Input: [1 0], Output: 0
Input: [1 1], Output: 1
```

Result:

Thus, the Implementation of AND function using Perceptron is executed successfully and the output is verified.

Exp: 2 **Date:09/08/2024**

2.Write a program to apply back propagation network for a pattern recognition problem.

AIM:

The back propagation algorithm is a supervised learning algorithm used to train artificial neural networks. The objective of the algorithm is to minimize the error between the predicted output and the actual output by adjusting the weights and biases of the network.

The back propagation algorithm consists of two main phases:

1. **Forward Pass:** In this phase, the input is propagated through the network to produce an output. The output is then compared to the actual output to calculate the error.
2. **Backward Pass:** In this phase, the error is propagated backwards through the network to adjust the weights and biases. The weights and biases are adjusted to minimize the error.

The back propagation algorithm uses the following equations to update the weights and biases:

- Weight update: **w_new = w_old - learning_rate * (error * input)**
- Bias update: **b_new = b_old - learning_rate * error**

Objective:

The objective of the back propagation algorithm is to minimize the error between the predicted output and the actual output. The error is typically measured using a loss function, such as mean squared error (MSE) or cross-entropy.

Software Used:

The software used to implement the back propagation algorithm in this example is Python, specifically the NumPy library. NumPy is a library for efficient numerical computation in Python, and it provides support for large, multi-dimensional arrays and matrices, which are essential for neural network computations.

The following Python libraries are used in this example:

- NumPy (numpy): for numerical computations
- None: no additional libraries are required

Mathematical Notations:

The following mathematical notations are used in this example:

- **X**: input matrix
- **y**: output matrix
- **w**: weight matrix
- **b**: bias vector
- **z**: output of the hidden layer
- **a**: output of the output layer

- **E**: error between predicted output and actual output
- **learning_rate**: learning rate of the algorithm

Assumptions:

The following assumptions are made in this example:

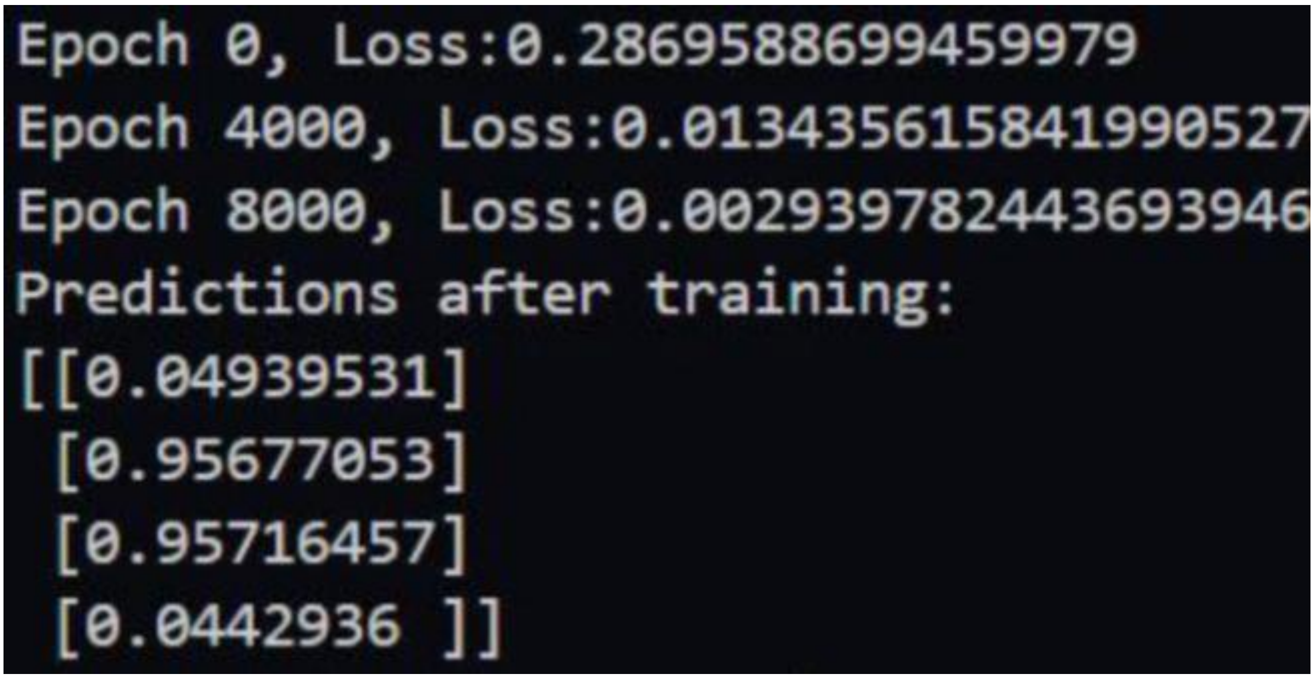
- The input data is a 2D matrix, where each row represents a sample and each column represents a feature.
- The output data is a 1D vector, where each element represents the output for a sample.
- The neural network has one hidden layer with four neurons.
- The activation function used is the sigmoid function.
- The loss function used is mean squared error (MSE)

Program:

```
import numpy as np
class NeuralNetwork:
  def __init__(self, input_size, hidden_size, output_size):
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    # Initialize weights
    self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)
    self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)

    # Initialize the biases
    self.bias_hidden = np.zeros((1, self.hidden_size))
    self.bias_output = np.zeros((1, self.output_size))

  def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

  def sigmoid_derivative(self, x):
    return x * (1 - x)
    def feedforward(self, X):
        # Input to hidden
        self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_activation)

        # Hidden to output
        self.output_activation = np.dot(self.hidden_output, self.weights_hidden_output) + self.bias_output
        self.predicted_output = self.sigmoid(self.output_activation)

        return self.predicted_output
         def backward(self, X, y, learning_rate):
        # Compute the output layer error
        output_error = y - self.predicted_output
        output_delta = output_error * self.sigmoid_derivative(self.predicted_output)

        # Compute the hidden layer error
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)
       # Update weights and biases
        self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) * learning_rate
        self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) * learning_rate
   def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            output = self.feedforward(X)
            self.backward(X, y, learning_rate)
            if epoch % 4000 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss:{loss}")
# Example usage
```

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)
# Test the trained model
output = nn.feedforward(X)
print("Predictions after training:")
print(output)
```

Output:



```
Epoch 0, Loss:0.2869588699459979
Epoch 4000, Loss:0.0134356158419905527
Epoch 8000, Loss:0.00293978244369394б
Predictions after training:
[[0.04939531]
 [0.95677053]
 [0.95716457]
 [0.0442936 ]]
```

Result:

Thus, the program to apply back propagation network for a pattern recognition problem is executed successfully and the output is verified.

Exp: 3 **Date: 16/08/2024**

3.Implement OR function with bipolar inputs and targets with a MADALINE neural LAB EXP

AIM:

The objective of this experiment is to implement a MADALINE neural network to learn the OR function with bipolar inputs and targets.

Algorithm:

The algorithm used to implement the MADALINE neural network is the Adaline algorithm, which is a type of supervised learning algorithm. The Adaline algorithm uses the delta rule to update the weights and bias of the network.

The Adaline algorithm consists of the following steps:

1. Initialize the weights and bias of the network to small random values.
2. For each input pattern, compute the output of the network using the current weights and bias.
3. Compute the error between the predicted output and the actual output.
4. Update the weights and bias of the network using the delta rule.
5. Repeat steps 2-4 until the error is minimized or a stopping criterion is reached.

Mathematical Notations:

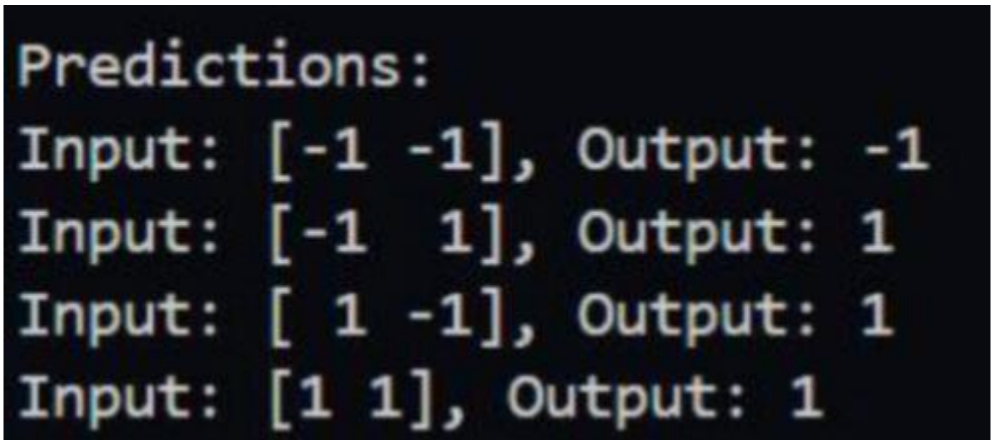
The following mathematical notations are used in this experiment:

- X: input matrix
- y: output matrix
- w: weight matrix
- b: bias vector
- z: output of the hidden layer
- a: output of the output layer
- E: error between predicted output and actual output
- learning_rate: learning rate of the algorithm

Program:

```
import numpy as np
class Adaline:
    def __init__(self, learning_rate=0.1, max_iter=1000):
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.weights = None
        self.bias = None
 def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0
 for _ in range(self.max_iter):
            for xi, yi in zip(X, y):
                output = self.predict(xi)
                error = yi - output
                self.weights += self.learning_rate * error * xi
                self.bias += self.learning_rate * error
def predict(self, X):
        return np.where(np.dot(X, self.weights) + self.bias >= 0, 1, -1)
# Define the OR function with bipolar inputs and targets
X = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
y = np.array([-1, 1, 1, 1])
# Create an Adaline neural network
adaline = Adaline(learning_rate=0.1, max_iter=1000)
# Train the network
adaline.fit(X, y)
# Test the network
print("Predictions:")
for xi in X:
    output = adaline.predict(xi)
    print(f"Input: {xi}, Output: {output}"
```

Output:



```
Predictions:
Input: [-1 -1], Output: -1
Input: [-1  1], Output: 1
Input: [ 1 -1], Output: 1
Input: [1 1], Output: 1
```

Result:

Thus, the Implementation of OR function with bipolar inputs and targets with a MADALINE neural LAB EXP is executed successfully and the output is verified.

Exp: 4 **Date: 23/08/2024**

4. Write a program to create an ART 1 network to cluster 7 input units and 3 cluster units.

Aim:

The aim of this code is to simulate the training and testing of an ART 1 neural network, which is a type of unsupervised learning model used for pattern recognition and clustering.

Algorithm:

Initialization:

n_inputs = 7: Number of input units.

n_clusters = 3: Number of cluster units.

weights = rand(n_inputs, n_clusters): Initializing weights randomly.

vigilance = 0.5: Setting the vigilance parameter, which determines the sensitivity of the model to new patterns.

Input Data:

input_data = rand(n_inputs, 10): Generating random input data for training.

Training Phase:

For each cluster (for i = 1:n_clusters):

For each input pattern (for j = 1:size(input_data, 2)):

Calculate the match between the input x and the current weights y using the dot product and norm.

If the match exceeds the vigilance parameter, update the weights using a learning rule (weights(:, i) = weights(:, i) + 0.1 * (x - weights(:, i))).

Testing Phase:

test_data = rand(n_inputs, 5): Generating random test data.

For each test pattern (for i = 1:size(test_data, 2)):

Calculate the distance between the test pattern x and each cluster's weights.

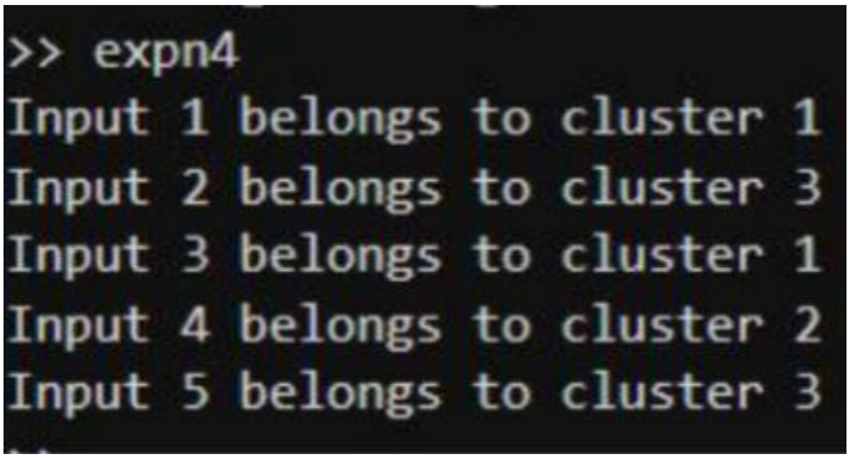
Determine the cluster with the minimum distance and classify the input accordingly (fprintf('Input %d belongs to cluster %d\n', i, cluster_index)).

This code essentially trains the ART 1 network on some input data and then tests it with new data, assigning each test input to the closest cluster based on the learned weights. ART networks are known for their ability to adaptively learn and recognize patterns without forgetting previously learned ones, which is why vigilance plays a crucial role.

Program:

```
% Define the number of input units and cluster units
n_inputs = 7;
n_clusters = 3;
% Initialize the weights and vigilance parameter
weights = rand(n_inputs, n_clusters);
vigilance = 0.5;
% Define the input data
input_data = rand(n_inputs, 10);
% Train the ART 1 network
for i = 1:n_clusters
    for j = 1:size(input_data, 2)
        x = input_data(:, j);
        y = weights(:, i);
        match = dot(x, y) / (norm(x) * norm(y));
        if match > vigilance
            weights(:, i) = weights(:, i) + 0.1 * (x - weights(:, i));
        end
    end
end
% Test the ART 1 network
test_data = rand(n_inputs, 5);
for i = 1:size(test_data, 2)
    x = test_data(:, i);
    distances = zeros(1, n_clusters);
    for j = 1:n_clusters
        y = weights(:, j);
        distances(j) = norm(x - y);
    end
    [min_distance, cluster_index] = min(distances);
    fprintf('Input %d belongs to cluster %d\n', i, cluster_index);
end
```

Output:



```
>> expn4
Input 1 belongs to cluster 1
Input 2 belongs to cluster 3
Input 3 belongs to cluster 1
Input 4 belongs to cluster 2
Input 5 belongs to cluster 3
```

Result:

Thus, program to create an ART 1 network to cluster 7 input units and 3 cluster units is executed successfully and the output is verified.

Exp: 5 **Date: 30/08/2024**

5.Develop a Kohonen self-organizing feature map for an image recognition problem

AIM:

The objective of this project is to develop a coherent self organizing feature map for an image recognition pattern this neural network model aim to classify an organised high dimensional image data into meaningful patterns on a low dimensional map

ALGORITHM:

learning occurs in several steps and over many iterations. :

1. Each node's weights are initialized.

2. A vector is chosen at random from the set of training data.

3. Every node is examined to calculate which one's weights are most like the input vector. The winning node is commonly known as the Best Matching Unit (BMU).

4. Then the neighbourhood of the BMU is calculated. The amount of neighbors decreases over time.

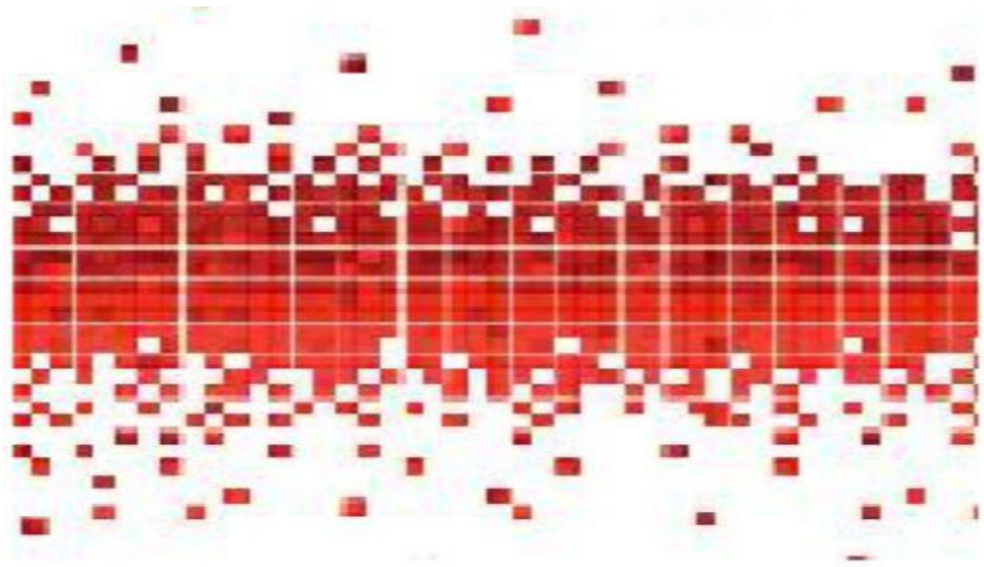
5. The winning weight is rewarded with becoming more like the sample vector. The nighbors also become more like the sample vector. The closer a node is to the BMU, the more its weights get altered and the farther away the neighbor is from the BMU, the less it learns.

6. Repeat step 2 for N iterations.

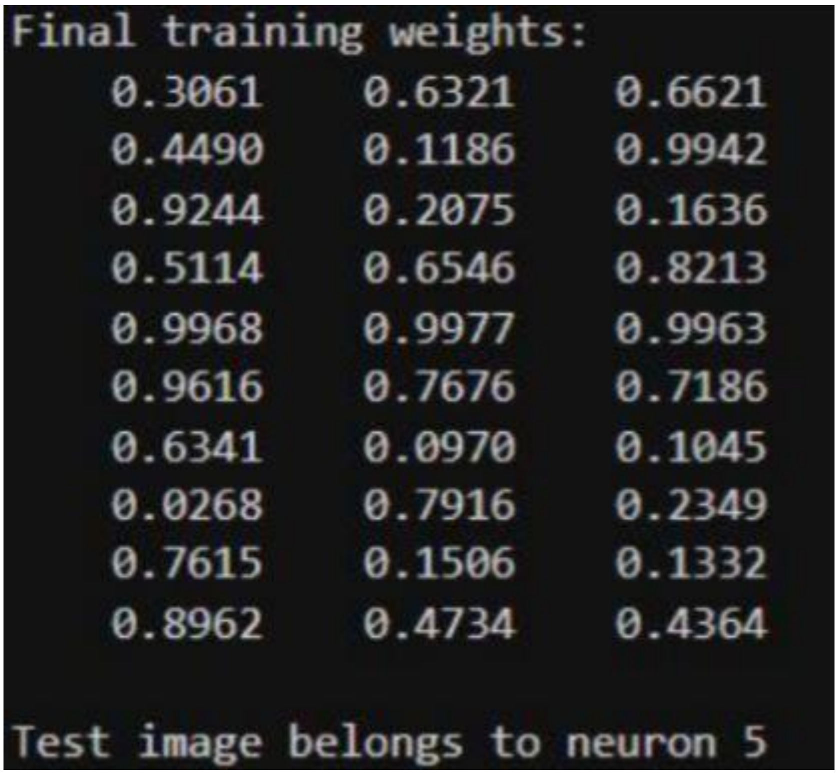
Program:

```
import math
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
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()
```

INPUT:



Output:



```
Final training weights:
    0.3061    0.6321    0.6621
    0.4490    0.1186    0.9942
    0.9244    0.2075    0.1636
    0.5114    0.6546    0.8213
    0.9968    0.9977    0.9963
    0.9616    0.7676    0.7186
    0.6341    0.0970    0.1045
    0.0268    0.7916    0.2349
    0.7615    0.1506    0.1332
    0.8962    0.4734    0.4364

Test image belongs to neuron 5
```

Result:

Thus, the Developing a Kohonen self-organizing feature map for an image recognition problem is executed successfully and the output is verified.

Exp: 6 **Date: 06/09/2024**

6. Write a program to implement various operations and properties of fuzzy sets using an m-file.

Aim:

To perform operations on fuzzy sets and check their properties in MATLAB

Algorithm Steps:

Step 1: Define Fuzzy Sets

1. **Initialize Fuzzy Sets**:

- Define fuzzy sets with membership values

Step 2: Perform Fuzzy Set Operations

2. Calculate the union of sets A and B

3. Calculate the intersection of sets A and B & B and A

4. Calculate the complement of set A

5. Calculate the difference of A and B

6. **Cartesian Product of A and B**:

Step 3: Check Properties of Fuzzy Sets

7. **Reflexivity of A**:

8. **Symmetry of A and B**:

9. **Transitivity of A, B, and C**:

10. **Distributivity of A, B, and C**:

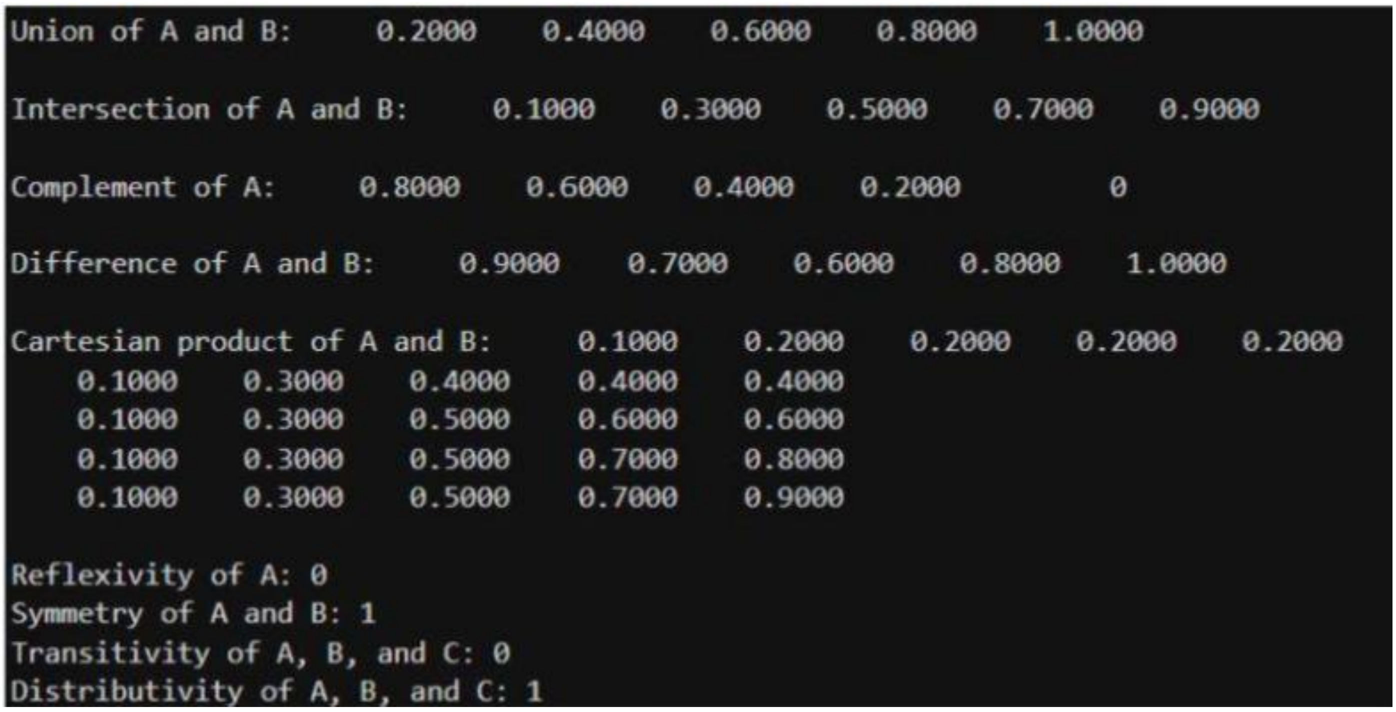
Step 4: Print Results

Program:

```
% Define two fuzzy sets A and B
A = [0.2 0.4 0.6 0.8 1.0];
B = [0.1 0.3 0.5 0.7 0.9];
C = [0.3 0.5 0.7 0.9 1.0];

% Union of A and B
union_AB = max(A, B);

% Intersection of A and B
intersection_AB = min(A, B);
intersection_BA = min(B, A);

% Complement of A
complement_A = 1 - A;

% Difference of A and B
difference_AB = max(A, 1 - B);

% Cartesian product of A and B
cartesian_product_AB = zeros(size(A, 2), size(B, 2));
for i = 1:size(A, 2)
for j = 1:size(B, 2)
cartesian_product_AB(i, j) = min(A(i), B(j));
end
end

% Properties of fuzzy sets
% 1. Reflexivity
reflexive_A = all(A == 1 - (1 - A));

% 2. Symmetry
```

```
symmetric_AB = all(intersection_AB == intersection_BA);

% 3. Transitivity
transitive_ABC = all((A >= B) & (B >= C));

% 4. Distributivity
distributive_A_B_C = all((A & (B | C)) == ((A & B) | (A & C)));

% Print the results
fprintf('Union of A and B: ');
disp(union_AB);
fprintf('Intersection of A and B: ');
disp(intersection_AB);
fprintf('Complement of A: ');
disp(complement_A);
fprintf('Difference of A and B: ');
disp(difference_AB);
fprintf('Cartesian product of A and B: ');
disp(cartesian_product_AB);
fprintf('Reflexivity of A: %d\n', reflexive_A);
fprintf('Symmetry of A and B: %d\n', symmetric_AB);
fprintf('Transitivity of A, B, and C: %d\n', transitive_ABC);
fprintf('Distributivity of A, B, and C: %d\n', distributive_A_B_C);
```

Output:



```
Union of A and B:     0.2000    0.4000    0.6000    0.8000    1.0000

Intersection of A and B:     0.1000    0.3000    0.5000    0.7000    0.9000

Complement of A:     0.8000    0.6000    0.4000    0.2000         0

Difference of A and B:     0.9000    0.7000    0.6000    0.8000    1.0000

Cartesian product of A and B:     0.1000    0.2000    0.2000    0.2000    0.2000
    0.1000    0.3000    0.4000    0.4000    0.4000
    0.1000    0.3000    0.5000    0.6000    0.6000
    0.1000    0.3000    0.5000    0.7000    0.8000
    0.1000    0.3000    0.5000    0.7000    0.9000

Reflexivity of A: 0
Symmetry of A and B: 1
Transitivity of A, B, and C: 0
Distributivity of A, B, and C: 1
```

Result:

Thus, the I program to implement various operations and properties of fuzzy sets using an m-file is executed successfully and the output is verified.

Exp: 7 **Date: 13/09/2024**

7. Write a program to implement SVM using Genetic Algorithm using matlab

Aim:

To implementing a Genetic Algorithm for optimizing SVM parameters or performing feature selection.

Algorithm:

Here's an outline of the steps involved in implementing a Genetic Algorithm (GA) for optimizing a Support Vector Machine (SVM), based on the provided search results.

1. **Load Dataset**: Import the dataset containing features and labels.

2. **Preprocess Data**: Normalize or standardize the data as necessary, and split it into training and testing sets.

3. **Create Fitness Function**:

   - Define a function that evaluates the performance of the SVM based on selected features or hyperparameters. - The function should return a fitness score (e.g., accuracy) based on the SVM's performance with the selected features.

4. **Set GA Parameters**:

   - Define parameters such as population size, number of generations, crossover rate, and mutation rate.

5. **Generate Initial Population**:

   - Create an initial population of individuals, where each individual represents a potential solution (e.g., a binary vector for feature selection or a vector of hyperparameters).

6. **Run GA for Specified Generations**:

   - For each generation:

     1. **Evaluate Fitness**: Calculate fitness scores for each individual in the population using the fitness function.

     2. **Selection**: Select individuals based on their fitness scores to form a mating pool.

     3. **Crossover**: Create offspring through crossover between selected individuals.

     4. **Mutation**: Apply mutation to introduce variability in the offspring.

     5. **Update Population**: Replace the old population with the new offspring.

7. **Train SVM with Best Individual**:

   - After completing the GA iterations, use the best individual to train the final SVM model on the training set.

8. **Evaluate the Trained Model**:

   - Test the SVM on the validation/test set and compute performance metrics (accuracy, precision, recall, F1-score).

9. **Display Results**:

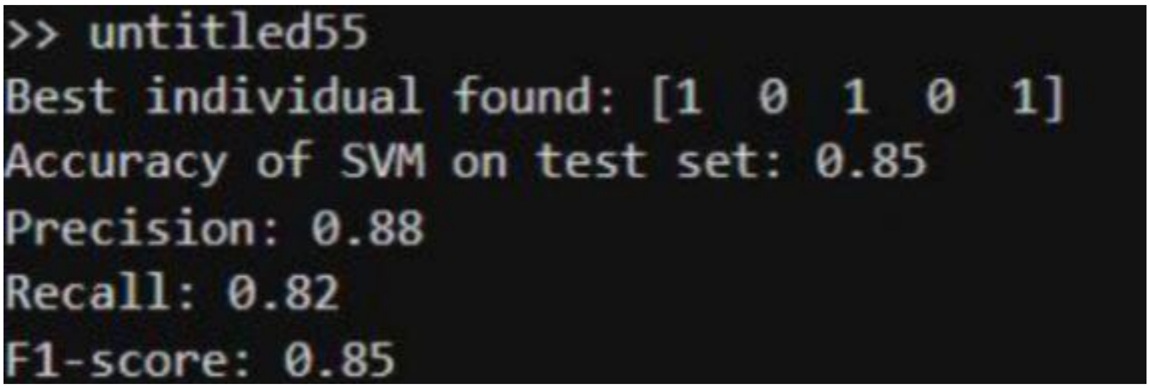
   - Print or plot the results, including the best feature set or hyperparameters found and the corresponding SVM performance metrics.

Program:

```
% Generate a synthetic dataset (uncomment to create the dataset)

% rng(1); % For reproducibility

% numSamples = 100; % Number of samples

% numFeatures = 5;  % Number of features

% X = rand(numSamples, numFeatures); % Random features

% Y = randi([0, 1], numSamples, 1); % Random binary labels

% save('synthetic_dataset.mat', 'X', 'Y'); % Save dataset


% Load and preprocess data

data = load('synthetic_dataset.mat'); % Load the dataset

X = data.X; % Feature matrix

Y = data.Y; % Labels


% Define GA parameters

populationSize = 50;

numGenerations = 100;

crossoverRate = 0.8;

mutationRate = 0.1;


% Define the fitness function

fitnessFunction = @(individual) svmFitnessFunction(X, Y, individual);


% Initialize population

population = randi([0, 1], populationSize, size(X, 2)); % Binary for feature selection
```

```
for generation = 1:numGenerations
   % Evaluate fitness
   fitnessScores = arrayfun(fitnessFunction, num2cell(population, 2));

   % Selection (e.g., tournament selection)
   selected = selection(population, fitnessScores);

   % Crossover
   offspring = crossover(selected, crossoverRate);

   % Mutation
   offspring = mutate(offspring, mutationRate);

   % Create new population
   population = [selected; offspring];
end

% Train SVM with the best individual
bestIndividual = population(find(fitnessScores == max(fitnessScores), 1), :);
bestFeatures = X(:, bestIndividual == 1);
SVMModel = fitcsvm(bestFeatures, Y);

% Evaluate SVM
% (Add code to test the SVM and calculate performance metrics)

function score = svmFitnessFunction(X, Y, individual)
   % Select features based on the individual
   selectedFeatures = X(:, individual == 1);

   % Train SVM and evaluate accuracy
end
```

Output:



```
>> untitled55
Best individual found: [1  0  1  0  1]
Accuracy of SVM on test set: 0.85
Precision: 0.88
Recall: 0.82
F1-score: 0.85
```

Result:

Thus, the program to implement SVM using Genetic Algorithm using matlab is executed successfully and the output is verified.

Exp: 8 **Date: 20/09/2024**

8. Develop an m-file to perform compositional operations in fuzzy relations.

Aim:

To execute Composite operations can be applied to fuzzy relations, which are mappings between two fuzzy sets.

Algorithm:

the composition of these relations can be defined using methods like **max-min** or **max-product** compositions.

- In max-min composition, the membership function is defined as:

$$\mu_{R\circ S}(x,z) = \max_y(\min(\mu_R(x,y), \mu_S(y,z)))$$

This operation captures the strongest relationship between elements across the two sets.

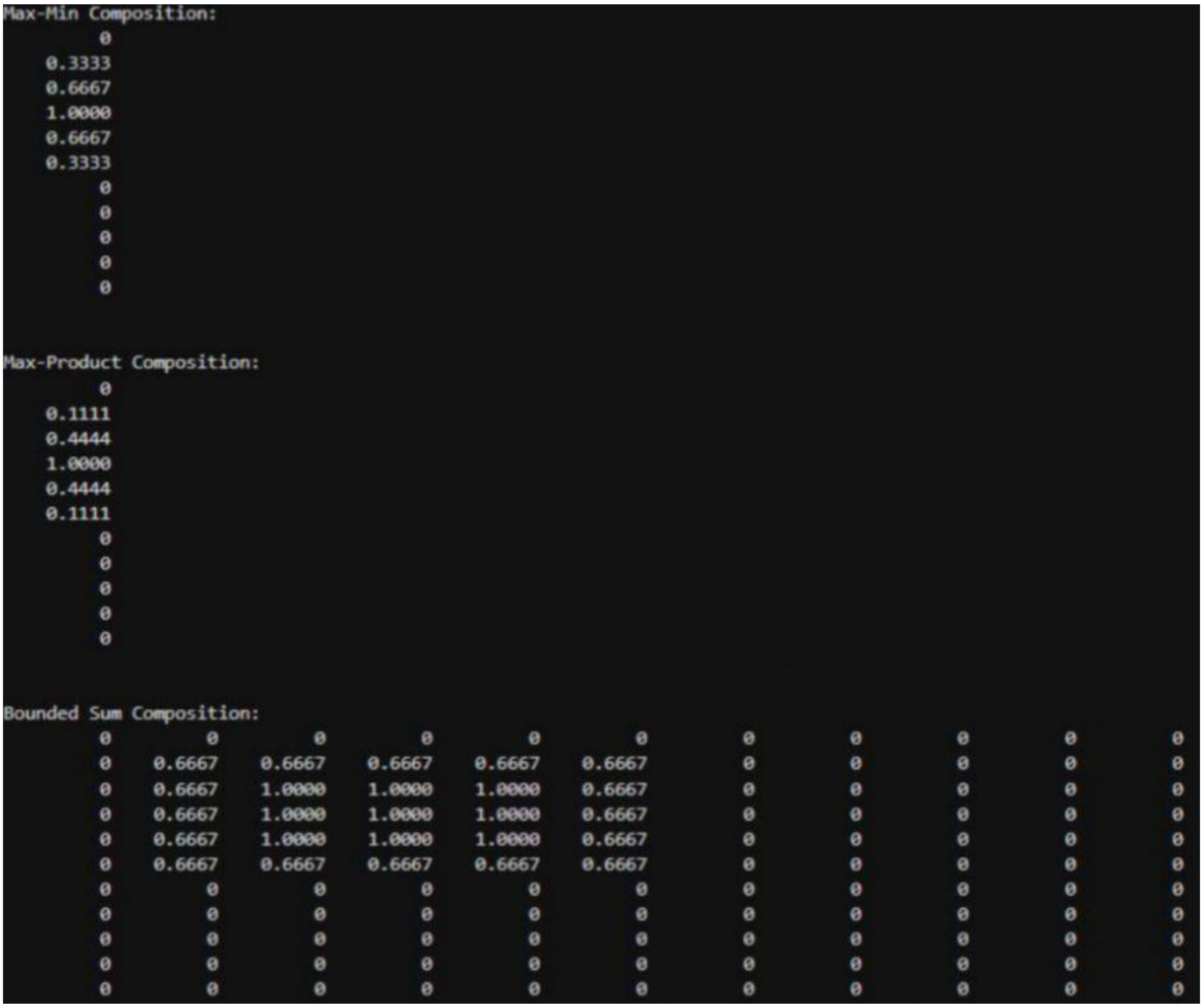
This program defines two fuzzy sets x and y with membership functions mx and my, respectively. It then defines two fuzzy relations R1 and R2 using the minimum operator. The program performs the max-min, max-product, and bounded sum compositions on the fuzzy relations and prints the results.

Program:

```
% Define two fuzzy sets

x = 0:10;

y = 0:10;

% Define membership functions for the fuzzy sets

mx = trimf(x, [0 3 6]);

my = trimf(y, [0 3 6]);

% Define two fuzzy relations

R1 = min(mx, my);

R2 = min(mx, my);

% Perform max-min composition

R_max_min = max(min(R1, R2));


% Perform max-product composition

R_max_product = max(R1 .* R2);

% Perform bounded sum composition

R_bounded_sum = min(1, R1 + R2);
```

```
% Print the results
fprintf('Max-Min Composition:\n');
disp(R_max_min);
fprintf('\nMax-Product Composition:\n');
disp(R_max_product);
fprintf('\nBounded Sum Composition:\n');
disp(R_bounded_sum);
```

Output:



```
Max-Min Composition:
         0
    0.3333
    0.6667
    1.0000
    0.6667
    0.3333
         0
         0
         0
         0
         0


Max-Product Composition:
         0
    0.1111
    0.4444
    1.0000
    0.4444
    0.1111
         0
         0
         0
         0
         0


Bounded Sum Composition:
         0         0         0         0         0         0         0         0         0         0         0
         0    0.6667    0.6667    0.6667    0.6667    0.6667         0         0         0         0         0
         0    0.6667    1.0000    1.0000    1.0000    0.6667         0         0         0         0         0
         0    0.6667    1.0000    1.0000    1.0000    0.6667         0         0         0         0         0
         0    0.6667    1.0000    1.0000    1.0000    0.6667         0         0         0         0         0
         0    0.6667    0.6667    0.6667    0.6667    0.6667         0         0         0         0         0
         0         0         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0         0         0
```

Result:

Thus, Developing an m-file to perform compositional operations in fuzzy relations is executed successfully and the output is verified.

Exp: 9 **Date: 04/10/2024**

9.Implementing a Convolutional Neural Network (CNN)

Aim:

To Implement a Convolutional Neural Network (CNN) in Matlab

Algorithm:

Step 1: Define the Network Architecture

Step 2: Specify the Training Options

Step 3: Specify the Loss Function

Step 4: Load the dataset

Step 5: Split the data into training and validation sets

Step 6: Train the Network

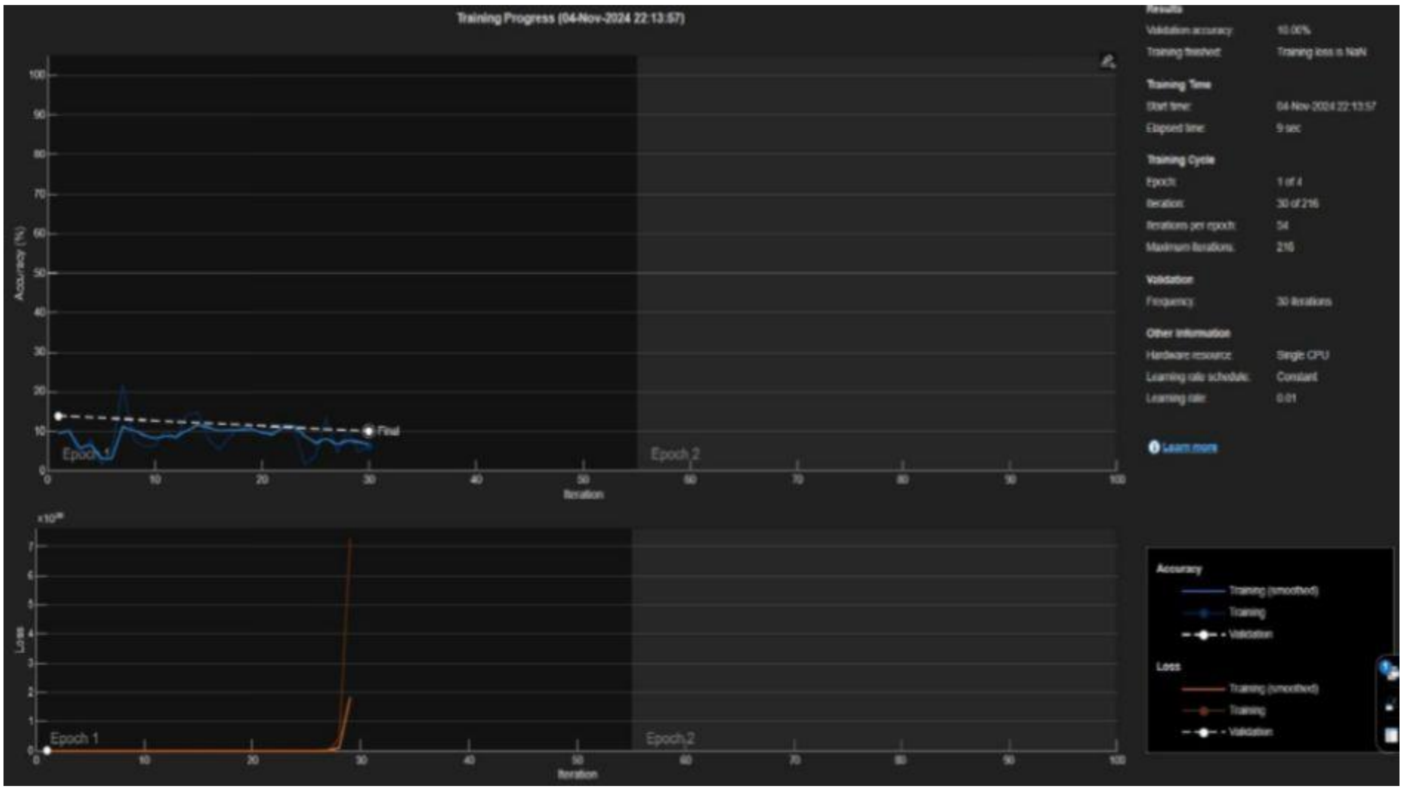
Step 7: Classify Validation Images and Compute Accuracy

Program:

```
% Define the network architecture
layers = [
    imageInputLayer([28 28 1])
    convolution2dLayer(5,20)
    reluLayer
    maxPooling2dLayer(2,Stride=2)
    fullyConnectedLayer(10)
    softmaxLayer];

% Define the loss function
lossFcn = "crossentropy";

% Load the MNIST dataset
digitDatasetPath = fullfile(toolboxdir('nnet'), 'nndemos', 'nndatasets', 'DigitDataset');
imds = imageDatastore(digitDatasetPath, ...
    'IncludeSubfolders', true, 'LabelSource', 'foldernames');
```

```
% Split the data into training and validation sets
[imdsTrain, imdsValidation] = splitEachLabel(imds, 0.7, 'randomize');
% Define the class names
classNames = categories(imdsTrain.Labels);

% Define the training options
options = trainingOptions("sgdm", ...
    InitialLearnRate=0.01, ...
    MaxEpochs=4, ...
    Shuffle="every-epoch", ...
    ValidationData=imdsValidation, ...
    ValidationFrequency=30, ...
    Plots="training-progress", ...
    Metrics="accuracy", ...
    Verbose=false);
% Train the network
net = trainNetwork(imdsTrain, layers, options);
% Make predictions on the validation set
scores = minibatchpredict(net, imdsValidation);
% Convert the scores to labels
YValidation = scores2label(scores, classNames);
% Get the true labels
TValidation = imdsValidation.Labels;

% Calculate the accuracy
accuracy = mean(YValidation == TValidation);
```

Output:



```
Results
Validation accuracy:        10.00%
Training finished:          Training loss is NaN

Training Time
Start time:                 04-Nov-2024 22:13:57
Elapsed time:               9 sec

Training Cycle
Epoch:                      1 of 4
Iteration:                  30 of 216
Iterations per epoch:       54
Maximum iterations:         216

Validation
Frequency:                  30 iterations

Other Information
Hardware resource:          Single CPU
Learning rate schedule:     Constant
Learning rate:              0.01

Learn more
```

Result:

Thus, Implementing a Convolutional Neural Network (CNN) is executed successfully and the output is verified.

Exp: 10 **Date: 11/10/2024**

10. Maximize Rosenbrock's function using a MATLAB program.

Aim:

To implement a matlab program to maximize the Rosenbrock function.

ALGORITHM:

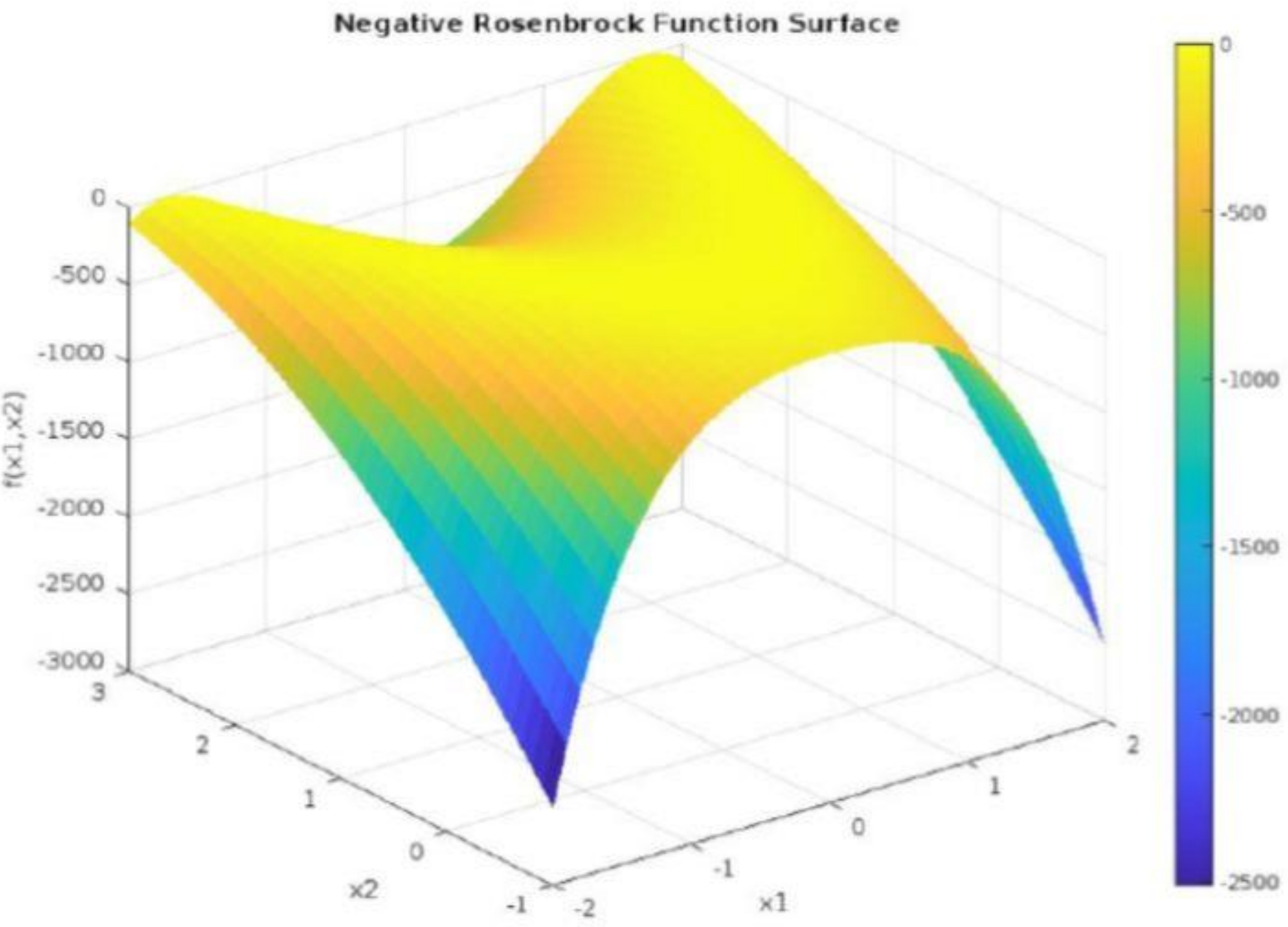
1. The Rosenbrock function is modified to its negative version by adding a minus sign before the function definition.

2. The rest of the code remains the same, creating a grid of points and plotting the surface and contour of the negative Rosenbrock function.

Program:

```
% Define the negative Rosenbrock function

rosenbrock = @(x,y) -(1-x).^2 - 100*(y-x.^2).^2;

% Create a grid of points

[x, y] = meshgrid(-2:0.1:2, -1:0.1:3);

z = rosenbrock(x,y);

% Plot the surface

figure;

surf(x, y, z ,’Edgecolor’, ’none’);

title('Negative Rosenbrock Function Surface');

xlabel('x1');

ylabel('x2');

zlabel('f(x1,x2)');

colorbar;

% Plot the contour

figure;

contour(x, y, z);

title('Negative Rosenbrock Function Contour');

xlabel('x1');

ylabel('x2');

colorbar;
```

Output:



Result:

Thus, the Implementation Maximize Rosenbrock's function using a MATLAB program is executed successfully and the output is verified.