Laboratory Record
Of : SC

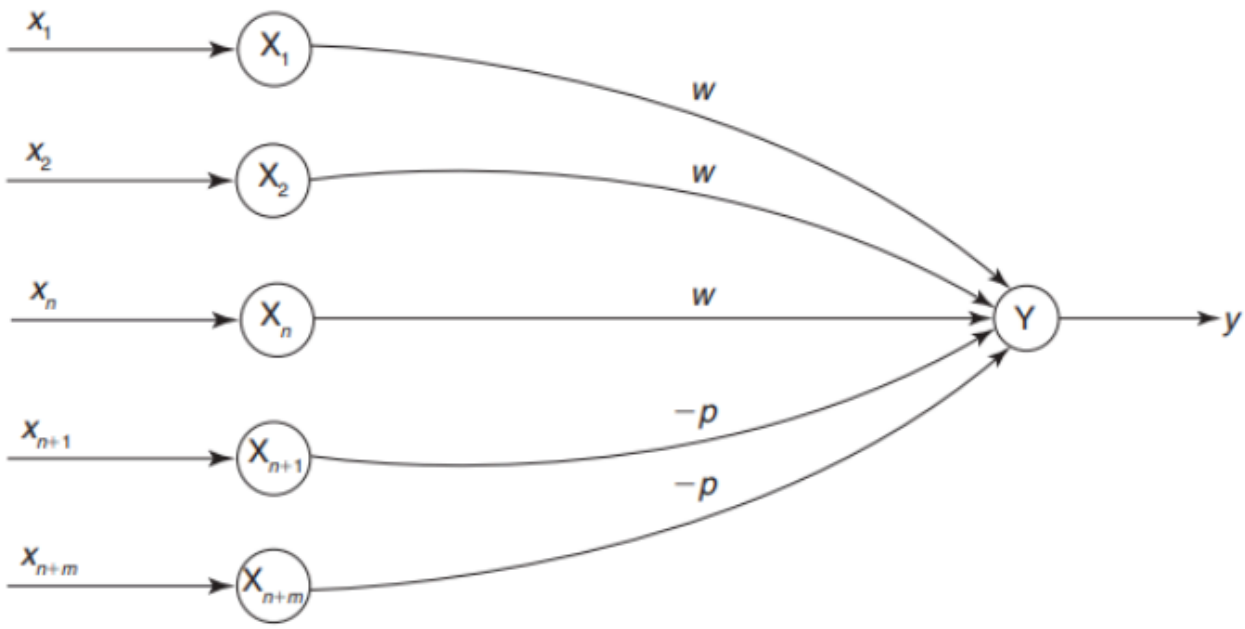
Roll No.: 160121749031
Experiment No.:
Sheet No.:1
Date.:

EXPERIMENT-1

AIM: To implement a Simple Neural Network (McCulloh-Pitts model) for realising AND Operation and OR operation.

DESCRIPTION:

The McCulloch-Pitts model, introduced by Warren McCulloch and Walter Pitts in 1943, represents one of the earliest and simplest forms of artificial neural networks. This mathematical model mimics the basic functioning of a biological neuron. It operates with binary inputs and outputs, where each input can be either 0 or 1, symbolizing the absence or presence of a signal. Each input to the neuron is associated with a binary weight (0 or 1), indicating the strength of the connection. The neuron also has a threshold value, and it fires (outputs a 1) if the weighted sum of its inputs exceeds this threshold; otherwise, it does not fire (outputs a 0). The activation function used is a simple step function that compares the weighted sum of inputs to the threshold to determine the output. This model, though basic, laid the foundational principles for modern neural networks and artificial intelligence, illustrating how networks of simple binary units could perform complex computations.



CODE:

```
import numpy as np
class McCullohPittsNeuron:
    def __init__(self, weight, threshold):
        self.weight = np.array(weight)
        self.threshold = threshold
    def activate(self, inputs):
        weighted_sum = np.dot(inputs, self.weight)
        return 1 if weighted_sum >= self.threshold else 0
andneuron = McCullohPittsNeuron([1, 1], 2)
print("AND OPERATION:")
print(andneuron.activate([0, 0]))
print(andneuron.activate([0, 1]))
print(andneuron.activate([1, 0]))
print(andneuron.activate([1, 1]))
orneuron = McCullohPittsNeuron([1, 1], 1)
print("\nOR OPERATION:")
print(orneuron.activate([0, 0]))
print(orneuron.activate([0, 1]))
print(orneuron.activate([1, 0]))
print(orneuron.activate([1, 1]))
```

OUTPUT:

```
AND OPERATION:
0
0
0
1

OR OPERATION:
0
1
1
1
```

EXPERIMENT-2

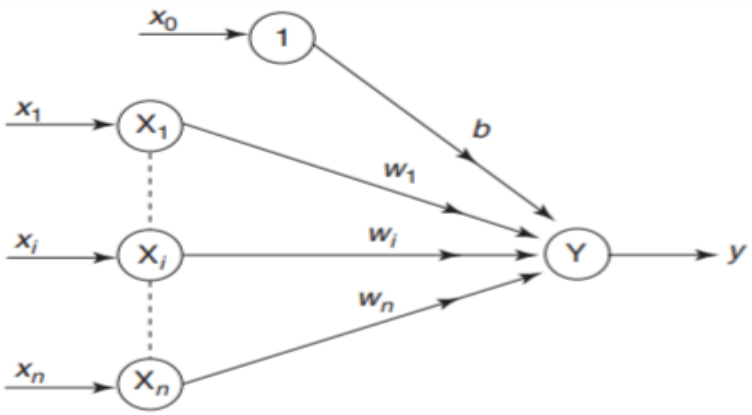
AIM: To implement of Perceptron network for realizing NAND operation.

DESCRIPTION:

The perceptron network, introduced by Frank Rosenblatt in 1958, is a fundamental building block of neural networks and one of the earliest models for supervised learning.

In the original perceptron network, the output obtained from the associator unit is a binary vector, and hence that output can be taken as input signal to the response unit, and classification can be performed. Here only the weights between the associator unit and the output unit can be adjusted, and the weights between the sensory and associator units are fixed. As a result, the discussion of the network is limited to a single portion. Thus, the associator unit behaves like the input unit

It is designed for binary classification tasks and consists of an input layer and an output layer without any hidden layers, making it a simple single-layer neural network. Each input feature is associated with a weight that determines its importance, and a bias term is added to the weighted sum to improve the model's fit. The perceptron uses a step activation function, outputting 1 if the weighted sum plus the bias exceeds a threshold, and 0 otherwise. The training process involves initializing the weights and bias, computing the output, and adjusting the weights and bias based on the error between the predicted and actual outputs. This process is repeated iteratively until the error converges to an acceptable level. While the perceptron is useful for simple binary classification tasks where data is linearly separable, it has limitations in handling non-linearly separable problems and does not generalize well to multi-class classification tasks. Despite these limitations, the perceptron network is crucial for understanding the principles of neural networks and has paved the way for more advanced models in machine learning.



Perceptron network activation function

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \le y_{in} \le \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

The weight updation in case of perceptron learning is as shown.

If $y \neq t$, then

$$w(\text{new}) = w(\text{old}) + \alpha t x \quad (\alpha - \text{learning rate})$$

else, we have

$$w(\text{new}) = w(\text{old})$$

Laboratory Record

Of : SC

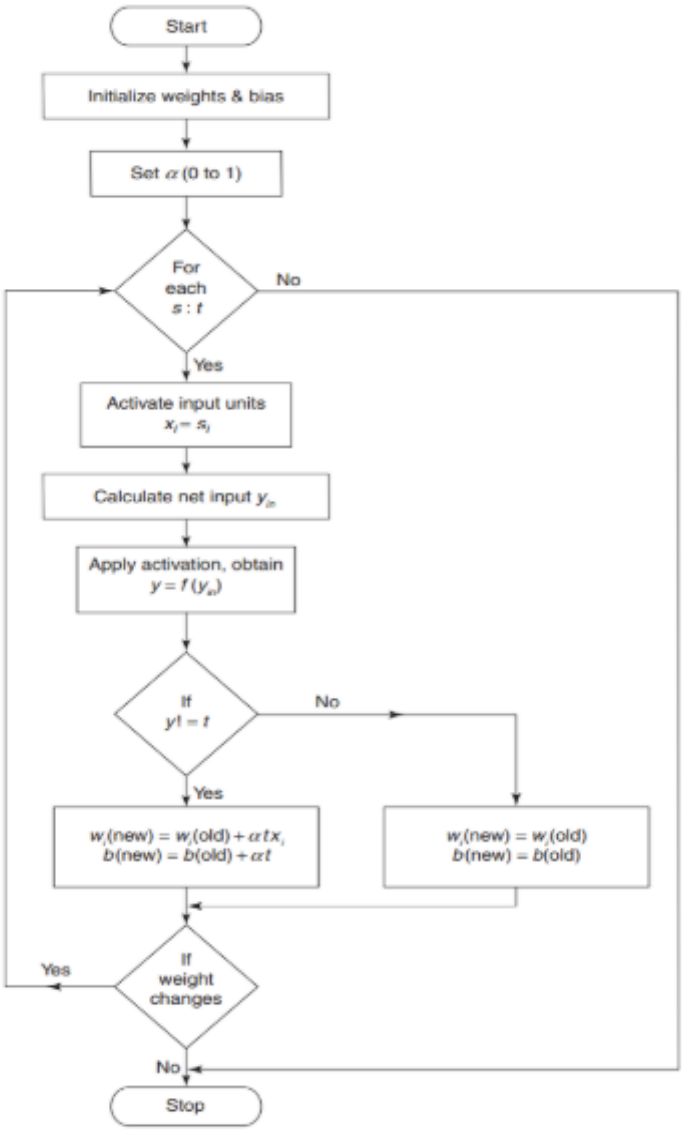
Roll No.: 160121749031

Experiment No.:

Sheet No.:5

Date.:

FLOWCHART:



ALGORITHM:

Step 0: Initialize the weights and the bias (for easy calculation they can be set to zero). Also initialize the learning rate $\alpha(0< \alpha \leq 1)$. For simplicity α is set to 1.

Step 1: Perform Steps 2–6 until the final stopping condition is false.

Step 2: Perform Steps 3–5 for each training pair indicated by $s:t$.

Step 3: The input layer containing input units is applied with identity activation functions:

$$x_i = s_i$$

Step 4: Calculate the output of the network. To do so, first obtain the net input:

$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

where "n" is the number of input neurons in the input layer. Then apply activations over the net input calculated to obtain the output:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

Step 5: *Weight and bias adjustment:* Compare the value of the actual (calculated) output and desired (target) output.

If $y \neq t$, then

$$w_i(\text{new}) = w_i(\text{old}) + \alpha t x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha t$$

else we have

$$w_i(\text{new}) = w_i(\text{old})$$
$$b(\text{new}) = b(\text{old})$$

Step 6: Train the network until there is no weight change. This is the stopping condition for the network. If this condition is not met, then start again from Step 2.

CODE:

```
import numpy as np
class Perceptron:
    def __init__(self, num_inputs, learning_rate=0.1, epochs=100):
        #explanation
        #How many input features your model will have (num_inputs).
        #How many input features your model will have (num_inputs).
        #How many times your model should learn from the data (epochs). (Default is 100)
        self.weights = np.random.rand(num_inputs + 1)  # +1 for the bias weight

        self.learning_rate = learning_rate
        self.epochs = epochs
```

```
    def predict(self, inputs):
        # Add bias input (always 1) to the inputs
        inputs_with_bias = np.concatenate((inputs, [1]))
        activation = np.dot(self.weights, inputs_with_bias)
        #np.dot() is a function from the NumPy library that calculates the dot product of two
arrays.
        #self.weights contains the weights of the model.
        #inputs_with_bias contains the input data along with a bias (always 1).
        return 1 if activation >= 0 else 0
    def train(self, training_inputs, labels):
        for _ in range(self.epochs):
            for inputs, label in zip(training_inputs, labels):
                prediction = self.predict(inputs) #to get the model's prediction for the current input.
                error = label - prediction #calculates the error
                # Update weights
                self.weights += self.learning_rate * error * np.concatenate((inputs, [1]))
                #for np.concatenate((inputs, [1])) it adds a bias (always 1) to the input data.
                #for self.learning_rate * error it adjusts the weights to reduce the error.
# Training data for NAND operation
training_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]) #containing input data for the
training.
labels = np.array([1, 1, 1, 0])  # NAND truth table and containing the correct outputs
#NAND Truth Table
# If both inputs are 0, the output is 1.
# If at least one input is 1, the output is 0.
#why NAND?
#By combining multiple NAND gates, more complex neural network architectures can be
constructed.
# NAND-based neural networks can learn and model non-linear relationships between input
and output data.
# Create and train the Perceptron for NAND operation
perceptron = Perceptron(num_inputs=2) #num_input= 2 mean two input neurons.
perceptron.train(training_inputs, labels)
```

```
# Test the trained Perceptron
print("NAND Operation:")
for inputs in training_inputs:
    output = perceptron.predict(inputs)
    print(f"NAND({inputs[0]}, {inputs[1]}) = {output}")
```

OUTPUT:

```
NAND Operation:
NAND(0, 0) = 1
NAND(0, 1) = 1
NAND(1, 0) = 1
NAND(1, 1) = 0
```

EXPERIMENT-3

AIM: To Implement of ANDNOT using ADALINE network.

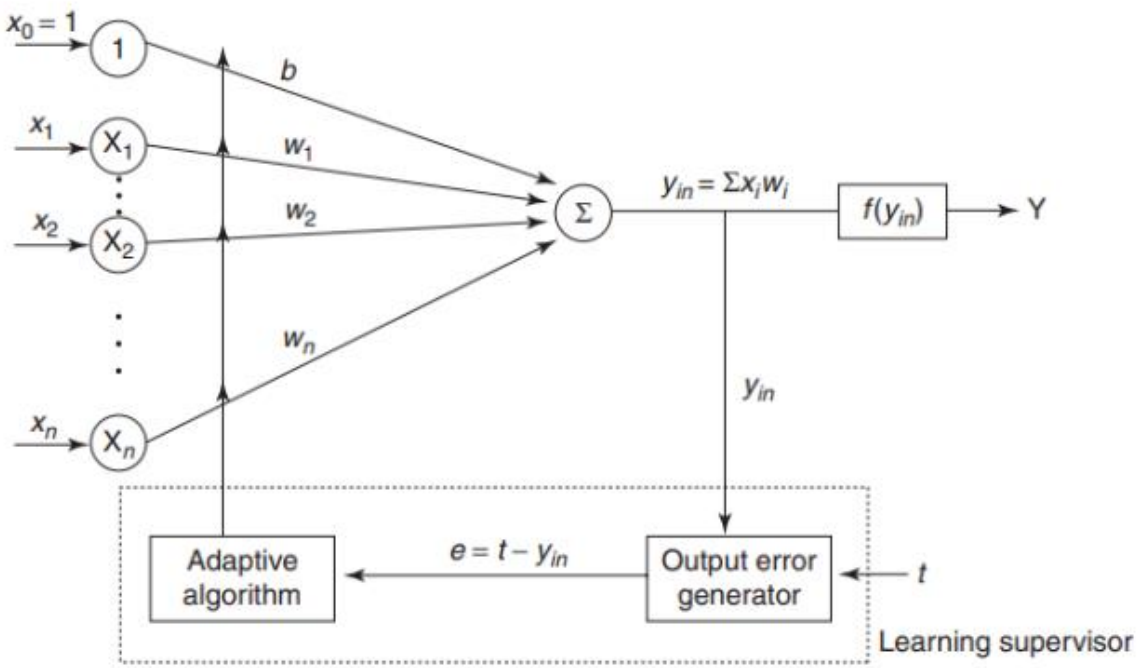
DESCRIPTION:

The Adaptive Linear Neuron (ADALINE) network, developed by Bernard Widrow and Ted Hoff in 1960, is a variation of the perceptron model tailored for linear regression tasks. The units with linear activation function are called linear units. A network with a single linear

unit is called an Adaline (adaptive linear neuron). That is, in an Adaline, the input–output relationship is linear. Adaline uses bipolar activation for its input signals and its target output.

The weights between the input and the output are adjustable. The bias in Adaline acts like an adjustable weight, whose connection is from a unit with activations being always 1. Adaline is a net which has only one output unit. The Adaline network may be trained using delta rule. The delta rule may also be called as least mean squares (LMS) rule or WidrowHoff rule. This learning rule is found to minimize the mean-squared error between the activation and the target value.

Structurally similar to the perceptron, ADALINE comprises an input layer and an output layer without hidden layers, making it a single-layer neural network. Inputs are represented as features \(x_1, x_2, ..., x_n \), each associated with a weight \(w_1, w_2, ..., w_n \) dictating its influence on the output. Unlike the perceptron, ADALINE employs a linear activation function, yielding the output as the weighted sum of inputs. It utilizes a form of gradient descent for weight adjustment, iteratively minimizing the error between predicted and target outputs. ADALINE finds applications in linear regression tasks like signal processing and prediction, but it's constrained by its linear separability assumption and sensitivity to hyperparameters such as the learning rate. Nonetheless, ADALINE's ability to handle continuous output and smoother transitions distinguishes it from the perceptron, making it a valuable tool for linear regression tasks within neural network frameworks



ALGORITHM:

Step 0: Weights and bias are set to some random values but not zero. Set the learning rate parameter α.

Step 1: Perform Steps 2–6 when stopping condition is false.

Step 2: Perform Steps 3–5 for each bipolar training pair *s*:*t*.

Step 3: Set activations for input units $i = 1$ to n.

$$x_i = s_i$$

Step 4: Calculate the net input to the output unit.

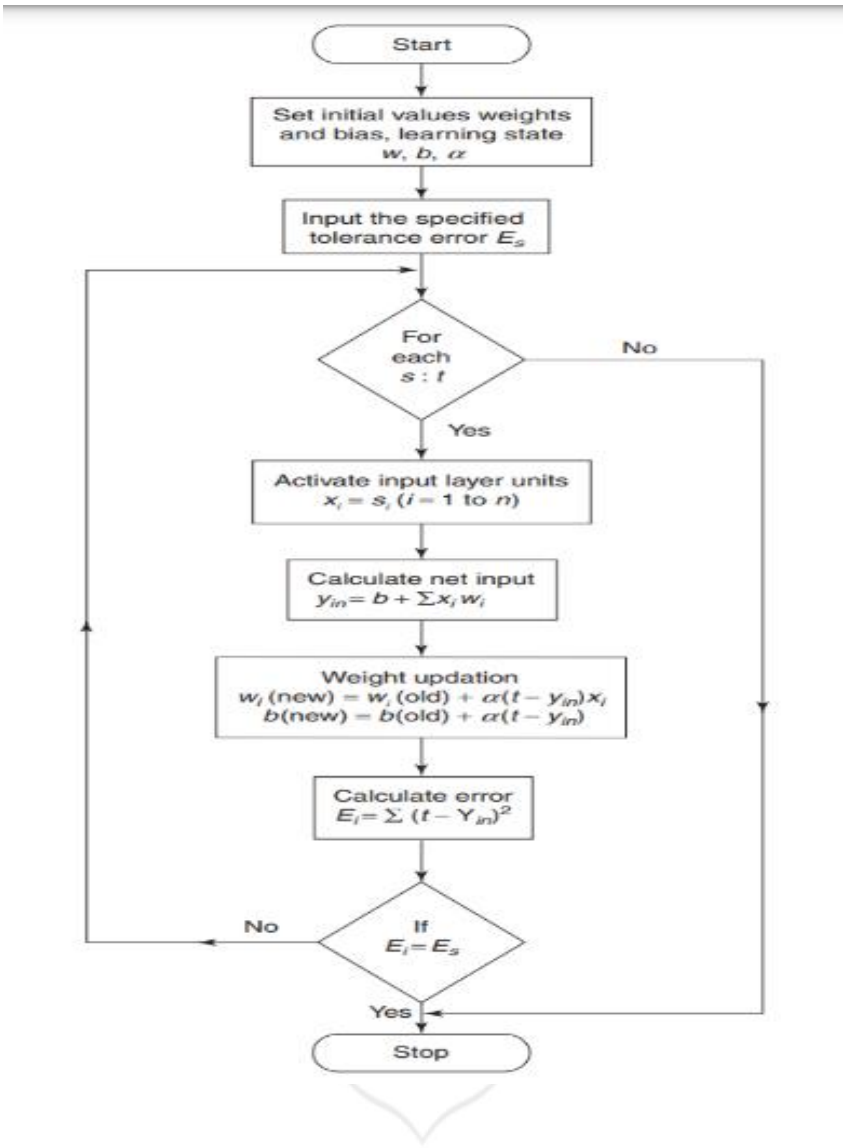
$$y_{in} = b + \sum_{i=1}^{n} x_i w_i$$

Step 5: Update the weights and bias for $i = 1$ to n:

$$w_i(\text{new}) = w_i(\text{old}) + \alpha (t - y_{in}) x_i$$
$$b(\text{new}) = b(\text{old}) + \alpha (t - y_{in})$$

Step 6: If the highest weight change that occurred during training is smaller than a specified tolerance then stop the training process, else continue. This is the test for stopping condition of a network.

FLOWCHART:



CODE:

```
import numpy as np
class Adaline:
    def __init__(self, input_size):
        # Initialize weights randomly
        self.weights = np.random.rand(input_size + 1)  # +1 for bias
    def predict(self, inputs):
        # Calculate weighted sum of inputs
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]
        # Apply step function (0 if summation < 0, 1 otherwise)
```

```
        return 1 if summation >= 0 else 0
    def train(self, inputs, target, learning_rate=0.1, epochs=100):
        for _ in range(epochs):
            for i in range(len(inputs)):
                # Make a prediction
                prediction = self.predict(inputs[i])
                # Calculate the error
                error = target[i] - prediction


                # Update weights
                self.weights[1:] += learning_rate * error * inputs[i]
                self.weights[0] += learning_rate * error
# Define training data for ANDNOT operation
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([0, 0, 1, 0])
# Create and train ADALINE network
adaline = Adaline(input_size=2)
adaline.train(inputs, targets)
# Test the trained network
print("Testing the ADALINE network for ANDNOT operation:")
for i in range(len(inputs)):
    prediction = adaline.predict(inputs[i])
    print(f"Input: {inputs[i]}, Output: {prediction}")
```

OUTPUT:

```
Testing the ADALINE network for ANDNOT operation:
Input: [0 0], Output: 0
Input: [0 1], Output: 0
Input: [1 0], Output: 1
Input: [1 1], Output: 0
```

Laboratory Record **Roll No.: 160121749031**

Of : SC **Experiment No.:**

Sheet No.:13

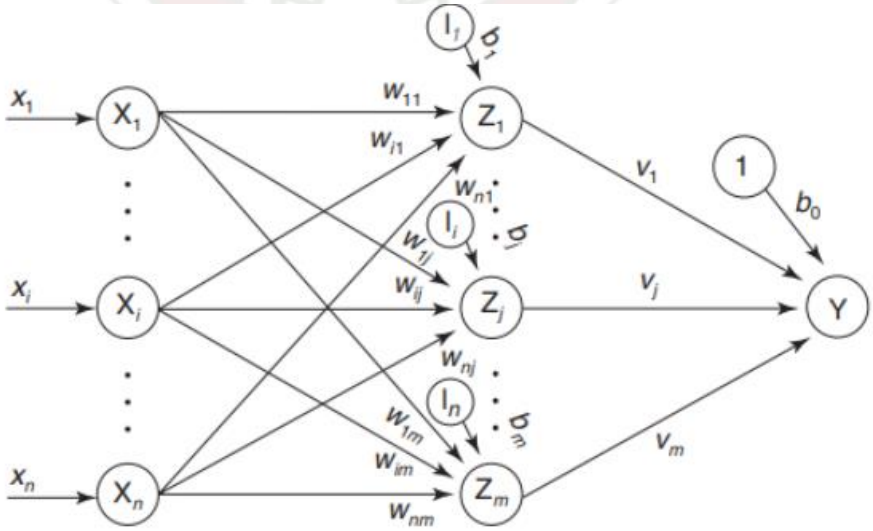
Date.:

EXPERIMENT-4

AIM: : To implement of XOR problem using MADALINE network

DESCRIPTION:

The Multiple ADALINE (MADALINE) network, pioneered by Bernard Widrow and Marcian Hoff in 1960, extends the capabilities of the Adaptive Linear Neuron (ADALINE) model by incorporating multiple layers of ADALINE units. Structurally, MADALINE is a multi-layer neural network where each layer comprises several ADALINE units. These units utilize linear activation functions, similar to ADALINE, computing the weighted sum of inputs without a threshold. During training, MADALINE employs supervised learning, adjusting weights based on the error between predicted and target outputs, often through gradient descent. MADALINE networks find applications in various pattern recognition tasks, including image classification, speech recognition, and character recognition. They offer advantages over single-layer ADALINE networks by capturing more complex relationships in data and handling a broader range of classification tasks. However, MADALINE networks are still constrained by their linear separability assumption and require careful parameter tuning for optimal performance. Overall, MADALINE networks represent an important advancement in neural network models, providing increased flexibility and capability for tackling complex pattern recognition challenges.



The activation for the Adaline (hidden) and Madaline (output) units is given by

$$f(x)=\begin{cases}1 & \text{if } x\geq 0\\ -1 & \text{if } x<0\end{cases}$$

Laboratory Record

Of : SC

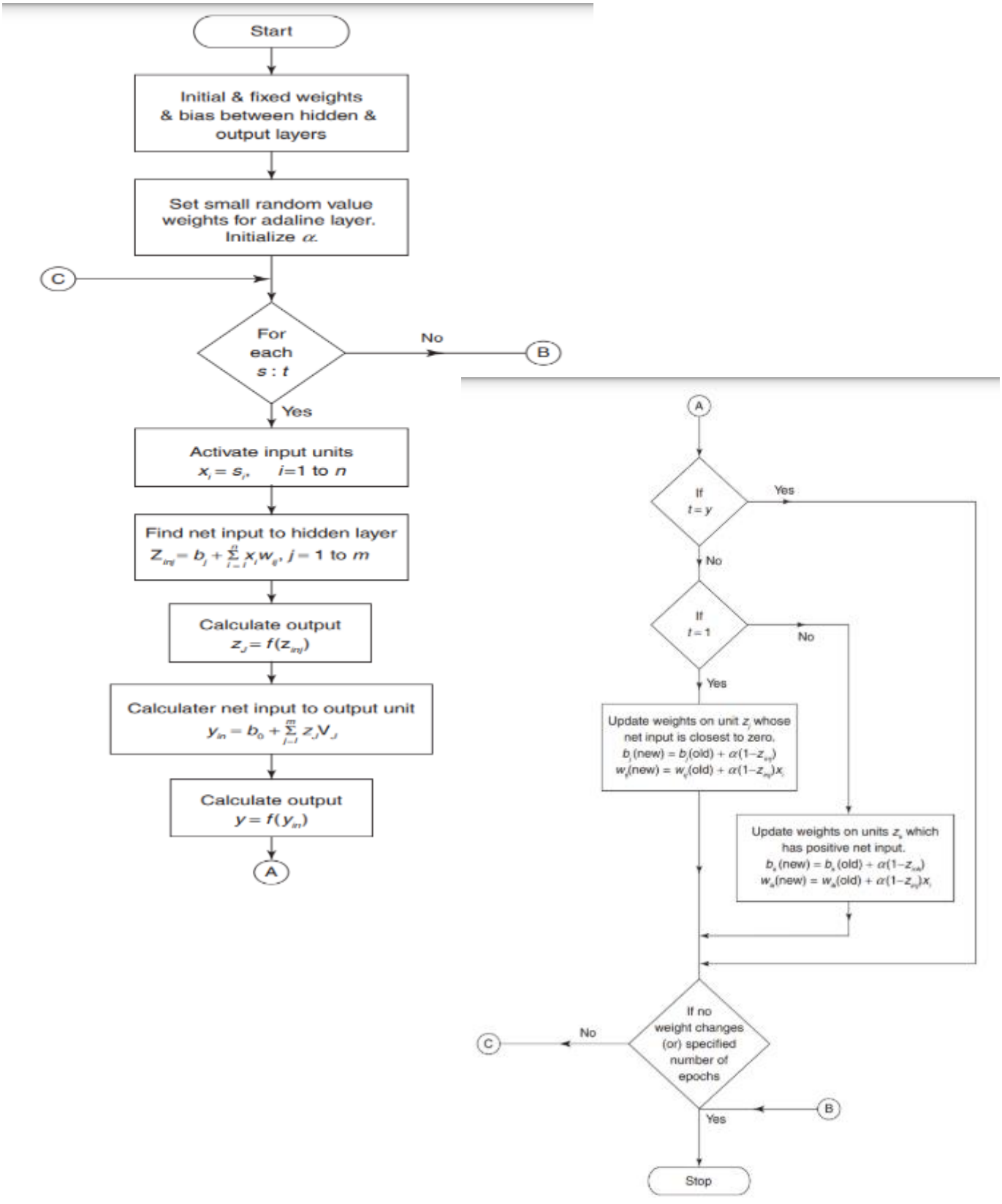
Roll No.: 160121749031

Experiment No.:

Sheet No.:14

Date.:

FLOWCHART:



ALGORITHM:

Step 0: Initialize the weights. The weights entering the output unit are set as above. Set initial small random values for Adaline weights. Also set initial learning rate α.

Step 1: When stopping condition is false, perform Steps 2–3.

Step 2: For each bipolar training pair $s:t$, perform Steps 3–7.

Step 3: Activate input layer units. For $i = 1$ to n,

$$x_i = s_i$$

Step 4: Calculate net input to each hidden Adaline unit:

$$z_{inj} = b_j + \sum_{i=1}^{n} x_i w_{ij}, \quad j = 1 \text{ to } m$$

Step 5: Calculate output of each hidden unit:

$$z_j = f(z_{inj})$$

Step 6: Find the output of the net:

$$y_{in} = b_0 + \sum_{j=1}^{m} z_j v_j$$

$$y = f(y_{in})$$

Step 7: Calculate the error and update the weights.

1. If $t = y$, no weight updation is required.
2. If $t \neq y$ and $t = +1$, update weights on z_j, where net input is closest to 0 (zero):

$$b_j(\text{new}) = b_j(\text{old}) + \alpha(1 - z_{inj})$$

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha(1 - z_{inj})x_i$$

3. If $t \neq y$ and $t = -1$, update weights on units z_k whose net input is positive:

$$w_{ik}(\text{new}) = w_{ik}(\text{old}) + \alpha(-1 - z_{ink})x_i$$

$$b_k(\text{new}) = b_k(\text{old}) + \alpha(-1 - z_{ink})$$

Step 8: Test for the stopping condition. (If there is no weight change or weight reaches a satisfactory level, or if a specified maximum number of iterations of weight updation have been performed then stop, or else continue).

CODE:

```
import numpy as np
class MADALINE:
    def __init__(self, num_inputs, num_hidden, learning_rate=0.1, epochs=100):
        self.num_inputs = num_inputs
        self.num_hidden = num_hidden
        self.learning_rate = learning_rate
        self.epochs = epochs
```

```
        # Initialize weights randomly for input to hidden layer
        self.weights_hidden = np.random.rand(num_inputs, num_hidden)
        # Initialize weights randomly for hidden to output layer
        self.weights_output = np.random.rand(num_hidden)
        # Initialize thresholds randomly for hidden layer
        self.thresholds_hidden = np.random.rand(num_hidden)
        # Initialize threshold randomly for output layer
        self.threshold_output = np.random.rand()
    def activate(self, x):
        return 1 if x >= 0 else 0
    def predict(self, inputs):
        # Calculate the output of the hidden layer
        hidden_output = np.array([self.activate(np.dot(inputs, self.weights_hidden[:, i]) -
self.thresholds_hidden[i]) for i in range(self.num_hidden)])
        # Calculate the output of the output layer
        output = self.activate(np.dot(hidden_output, self.weights_output) -
self.threshold_output)
        return output
    def train(self, training_inputs, labels):
        for _ in range(self.epochs):
            for inputs, label in zip(training_inputs, labels):
                # Forward propagation
                hidden_output = np.array([self.activate(np.dot(inputs, self.weights_hidden[:, i]) -
self.thresholds_hidden[i]) for i in range(self.num_hidden)])
                output = self.activate(np.dot(hidden_output, self.weights_output) -
self.threshold_output)
                # Backpropagation
                output_error = label - output
                hidden_error = np.dot(output_error, self.weights_output)
                # Update weights and thresholds
                self.weights_output += self.learning_rate * output_error * hidden_output
                self.threshold_output -= self.learning_rate * output_error
                for i in range(self.num_hidden):
                    self.weights_hidden[:, i] += self.learning_rate * hidden_error[i] * inputs
                    self.thresholds_hidden[i] -= self.learning_rate * hidden_error[i]
```

```
# Training data for XOR operation
training_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
labels = np.array([0, 1, 1, 0])  # XOR truth table
# Create and train the MADALINE network for XOR operation
madaline = MADALINE(num_inputs=2, num_hidden=2)
madaline.train(training_inputs, labels)
# Test the trained MADALINE network
print("XOR Operation:")
for inputs in training_inputs:
    output = madaline.predict(inputs)
    print(f"XOR({inputs[0]}, {inputs[1]}) = {output}")
```

OUTPUT:

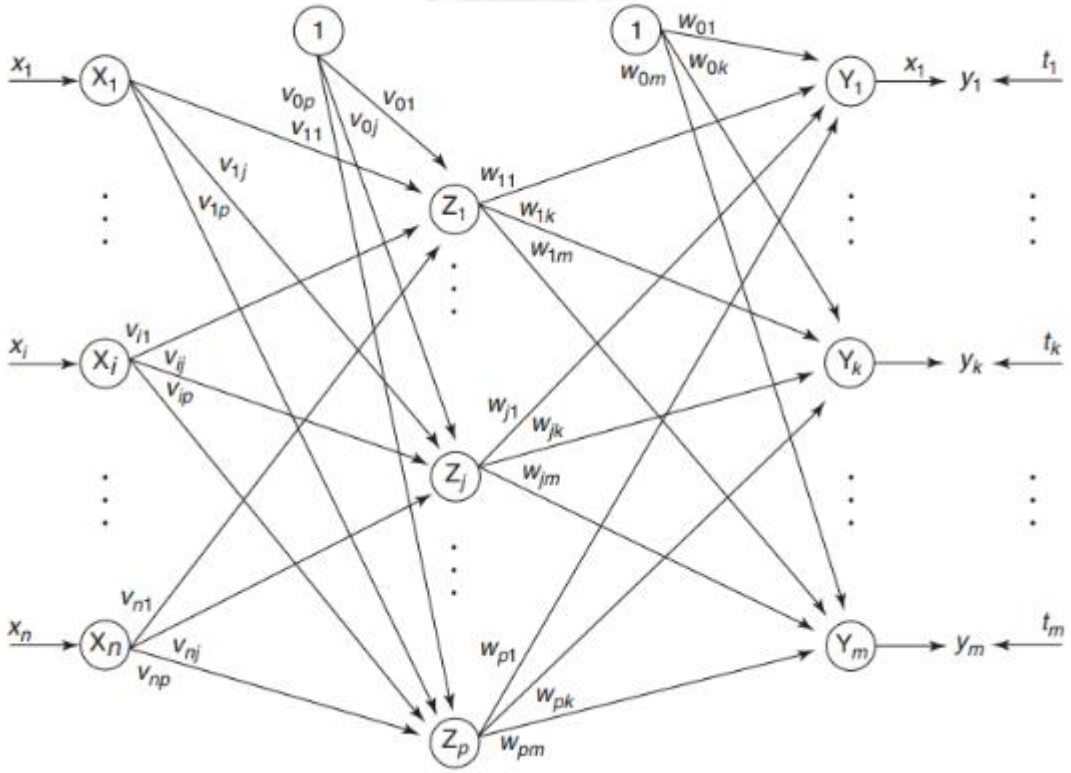
```
XOR Operation:
XOR(0, 0) = 1
XOR(0, 1) = 1
XOR(1, 0) = 1
XOR(1, 1) = 0
```

EXPERIMENT-5

AIM: : : To Design and Develop the Back Propagation Algorithm

DESCRIPTION:

The Backpropagation algorithm is a cornerstone in training artificial neural networks, facilitating the refinement of network weights to minimize prediction errors. During the forward pass, input data traverses through the network, with each layer performing computations to generate an output. Following this, the error between predicted and actual outputs is computed using a chosen loss function. The Backpropagation process then begins, propagating this error backward through the network to compute gradients of the loss function with respect to each weight. This gradient information is then used to update the weights in the opposite direction of the gradient, scaled by a learning rate, thereby minimizing the loss. This iterative process continues through multiple epochs until convergence is achieved. Backpropagation has revolutionized various domains, including image recognition, natural language processing, and financial modeling, by enabling neural networks to learn intricate patterns and make accurate predictions. Despite its effectiveness, careful tuning of hyperparameters and consideration of potential challenges like vanishing gradients are essential for successful implementation.



ALGORITHM:

Step 0: Initialize weights and learning rate (take some small random values).

Step 1: Perform Steps 2–9 when stopping condition is false.

Step 2: Perform Steps 3–8 for each training pair.

Feed-forward phase (Phase I):

Step 3: Each input unit receives input signal x_i and sends it to the hidden unit ($i = 1$ to n).

Step 4: Each hidden unit $z_j (j = 1$ to $p)$ sums its weighted input signals to calculate net input:

$$z_{inj} = v_{0j} + \sum_{i=1}^{n} x_i v_{ij}$$

Calculate output of the hidden unit by applying its activation functions over z_{inj} (binary or bipolar sigmoidal activation function):

$$z_j = f(z_{inj})$$

and send the output signal from the hidden unit to the input of output layer units.

Step 5: For each output unit y_k ($k = 1$ to m), calculate the net input:

$$y_{ink} = w_{0k} + \sum_{j=1}^{p} z_j w_{jk}$$

and apply the activation function to compute output signal

$$y_k = f(y_{ink})$$

Back-propagation of error (Phase II):

Step 6: Each output unit $y_k (k = 1$ to $m)$ receives a target pattern corresponding to the input training pattern and computes the error correction term:

$$\delta_k = (t_k - y_k) f'(y_{ink})$$

The derivative $f'(y_{ink})$ can be calculated as in Section 2.3.3. On the basis of the calculated error correction term, update the change in weights and bias:

$$\Delta w_{jk} = \alpha \delta_k z_j; \quad \Delta w_{0k} = \alpha \delta_k$$

Also, send δ_k to the hidden layer backwards.

Step 7: Each hidden unit ($z_j, j = 1$ to p) sums its delta inputs from the output units:

$$\delta_{inj} = \sum_{k=1}^{m} \delta_k w_{jk}$$

The term δ_{inj} gets multiplied with the derivative of $f(z_{inj})$ to calculate the error term:

$$\delta_j = \delta_{inj} f'(z_{inj})$$

The derivative $f'(z_{inj})$ can be calculated as discussed in Section 2.3.3 depending on whether binary or bipolar sigmoidal function is used. On the basis of the calculated δ_j, update the change in weights and bias:

$$\Delta v_{ij} = \alpha \delta_j x_i; \quad \Delta v_{0j} = \alpha \delta_j$$

Weight and bias updation (Phase III):

Step 8: Each output unit ($y_k, k = 1$ to m) updates the bias and weights:

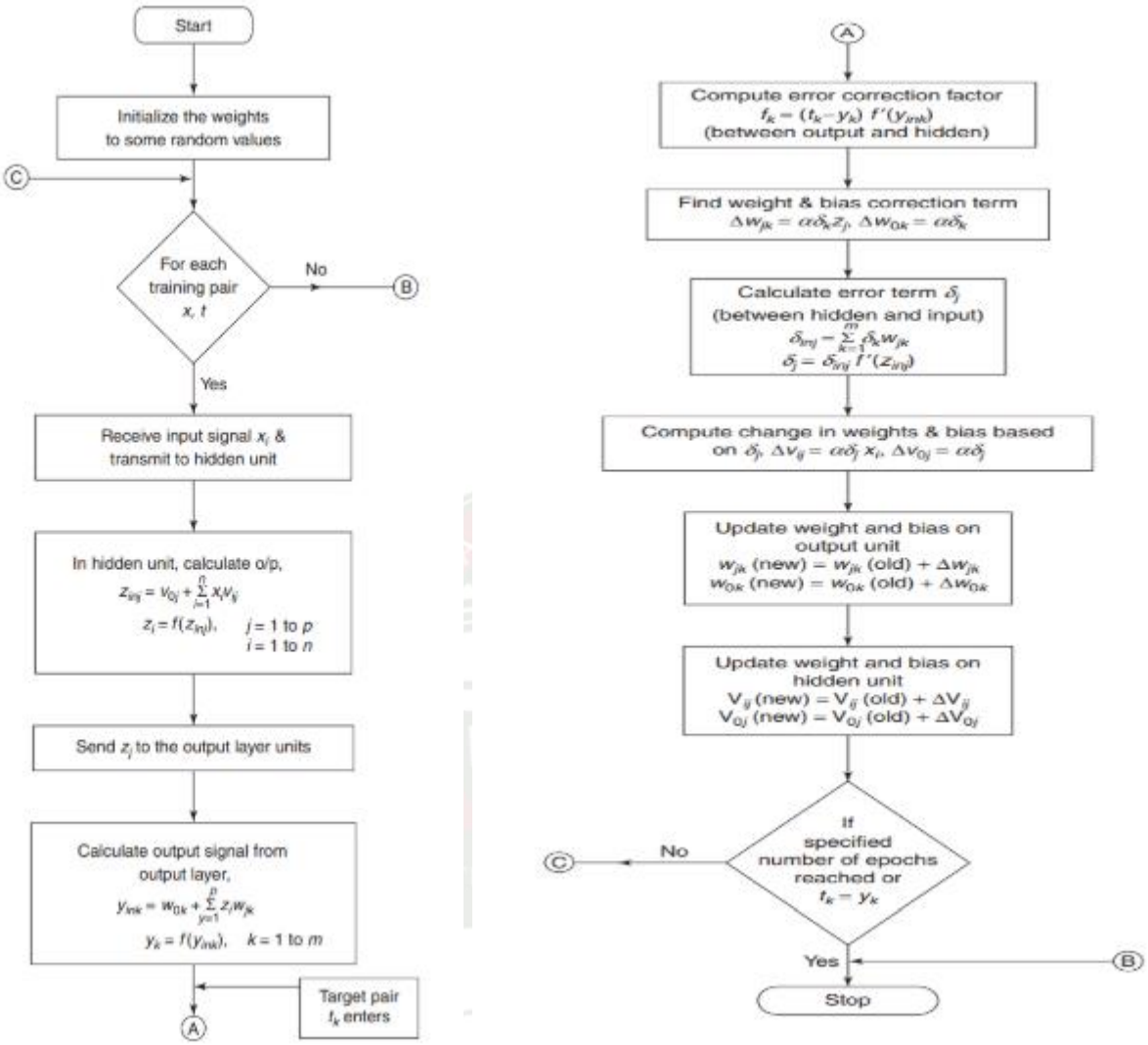
$$w_{jk}(\text{new}) = w_{jk}(\text{old}) + \Delta w_{jk}$$
$$w_{0k}(\text{new}) = w_{0k}(\text{old}) + \Delta w_{0k}$$

Each hidden unit ($z_j, j = 1$ to p) updates its bias and weights:

$$v_{ij}(\text{new}) = v_{ij}(\text{old}) + \Delta v_{ij}$$
$$v_{0j}(\text{new}) = v_{0j}(\text{old}) + \Delta v_{0j}$$

Step 9: Check for the stopping condition. The stopping condition may be certain number of epochs reached or when the actual output equals the target output.

FLOWCHART:



CODE:

```
import numpy as np
# Define the sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# Define the neural network class
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
```

```
        # Initialize weights randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
    def train(self, input_data, targets, learning_rate, epochs):
        for epoch in range(epochs):
            # Forward pass
            hidden_output = sigmoid(np.dot(input_data, self.weights_input_hidden))
            output = sigmoid(np.dot(hidden_output, self.weights_hidden_output))
            # Backpropagation
            output_error = targets - output
            output_delta = output_error * sigmoid_derivative(output)
            hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
            hidden_delta = hidden_error * sigmoid_derivative(hidden_output)
            # Update weights
            self.weights_hidden_output += learning_rate * np.dot(hidden_output.T, output_delta)
            self.weights_input_hidden += learning_rate * np.dot(input_data.T, hidden_delta)
    def predict(self, input_data):
        hidden_output = sigmoid(np.dot(input_data, self.weights_input_hidden))
        output = sigmoid(np.dot(hidden_output, self.weights_hidden_output))
        return output
# Example usage
input_data = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])
# Initialize neural network
input_size = 2
hidden_size = 3
output_size = 1
learning_rate = 0.1
epochs = 1000
nn = NeuralNetwork(input_size, hidden_size, output_size)
# Train neural network
nn.train(input_data, targets, learning_rate, epochs)
# Test the trained network
test_input = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```
predictions = nn.predict(test_input)
print("Predictions:")
print(predictions)
```

OUTPUT:

```
Predictions:
[[0.47733281]
 [0.49127459]
 [0.54109164]
 [0.52728026]]
```

EXPERIMENT-6

AIM: To Implement of Bidirectional Associative Memory (BAM) network

DESCRIPTION:

The Bidirectional Associative Memory (BAM) network, conceptualized by Bart Kosko in 1988, is a powerful recurrent neural network renowned for its bidirectional associative memory retrieval capabilities. Consisting of two layers – an input and an output layer – the network is fully interconnected, enabling it to establish associations between input and output patterns during training. Unlike traditional feedforward networks, BAM networks excel in bidirectional retrieval, effortlessly recalling both associated input and output patterns. Leveraging Hebbian learning during training, BAM networks adjust connection weights based on correlations between input and output patterns. This dynamic enables them to settle into stable states, providing robust associative memory retrieval. Common applications of BAM networks include pattern recognition tasks, associative memory retrieval, and autoassociation challenges. However, challenges like scalability and generalization to noisy or incomplete data persist, demanding careful consideration during deployment. Despite these limitations, the BAM network remains an invaluable tool for tasks requiring bidirectional associative memory and pattern recognition.

Activation Functions for BAM

The step activation function with a nonzero threshold is used as the activation function for discrete BAM networks. The activation function is based on whether the input target vector pairs used are binary or bipolar. The activation function for the Y layer.

1. with binary input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > 0 \\ y_j & \text{if } y_{inj} = 0 \\ 0 & \text{if } y_{inj} < 0 \end{cases}$$

2. with bipolar input vectors is

$$y_j = \begin{cases} 1 & \text{if } y_{inj} > \theta_j \\ y_j & \text{if } y_{inj} = \theta_j \\ -1 & \text{if } y_{inj} < \theta_j \end{cases}$$

The activation function for the X layer

1. with binary input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > 0 \\ x_i & \text{if } x_{ini} = 0 \\ 0 & \text{if } x_{ini} < 0 \end{cases}$$

2. with bipolar input vectors is

$$x_i = \begin{cases} 1 & \text{if } x_{ini} > \theta_i \\ x_i & \text{if } x_{ini} = \theta_i \\ -1 & \text{if } x_{ini} < \theta_i \end{cases}$$

ALGORITHM:

Step 0: Initialize the weights to store p vectors. Also initialize all the activations to zero.

Step 1: Perform Steps 2–6 for each testing input.

Step 2: Set the activations of X layer to current input pattern, i.e., presenting the input pattern x to X layer and similarly presenting the input pattern y to Y layer. Even though, it is bidirectional memory, at one time step, signals can be sent from only one layer. So, either of the input patterns may be the zero vector.

Step 3: Perform Steps 4–6 when the activations are not converged.

Step 4: Update the activations of units in Y layer. Calculate the net input,

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Applying the activations (as in Section 4.5.3.2), we obtain

$$y_j = f(y_{inj})$$

Send this signal to the X layer.

Step 5: Update the activations of units in X layer. Calculate the net input,

$$x_{ini} = \sum_{j=1}^{m} y_j w_{ij}$$

Apply the activations over the net input,

$$x_i = f(x_{ini})$$

Send this signal to the Y layer.

Step 6: Test for convergence of the net. The convergence occurs if the activation vectors x and y reach equilibrium. If this occurs then stop, otherwise, continue.

CODE:

```
import numpy as np
class BAM:
    def __init__(self, num_input_neurons, num_output_neurons):
        self.num_input_neurons = num_input_neurons
        self.num_output_neurons = num_output_neurons
        self.weights = np.zeros((num_input_neurons, num_output_neurons))
    def train(self, input_patterns, output_patterns):
        for input_pattern, output_pattern in zip(input_patterns, output_patterns):
            input_pattern = np.array(input_pattern).reshape(-1, 1)
```

```
        output_pattern = np.array(output_pattern).reshape(-1, 1)
        self.weights += np.dot(input_pattern, output_pattern.T)
    def associate(self, input_pattern):
        input_pattern = np.array(input_pattern).reshape(-1, 1)
        output_pattern = np.dot(self.weights.T, input_pattern)
        output_pattern[output_pattern >= 0] = 1
        output_pattern[output_pattern < 0] = -1
        return output_pattern.T.tolist()
# Example usage
if __name__ == "__main__":
    # Training data for BAM network
    input_patterns = [[1, -1, 1], [1, 1, -1], [-1, -1, 1]]
    output_patterns = [[1, 1], [1, -1], [-1, 1]]
    # Create and train the BAM network
    bam = BAM(num_input_neurons=len(input_patterns[0]),
num_output_neurons=len(output_patterns[0]))
    bam.train(input_patterns, output_patterns)
    # Test the BAM network
    print("Associations:")
    for input_pattern in input_patterns:
        output_pattern = bam.associate(input_pattern)
        print(f"Input: {input_pattern} => Output: {output_pattern}")
```

OUTPUT:

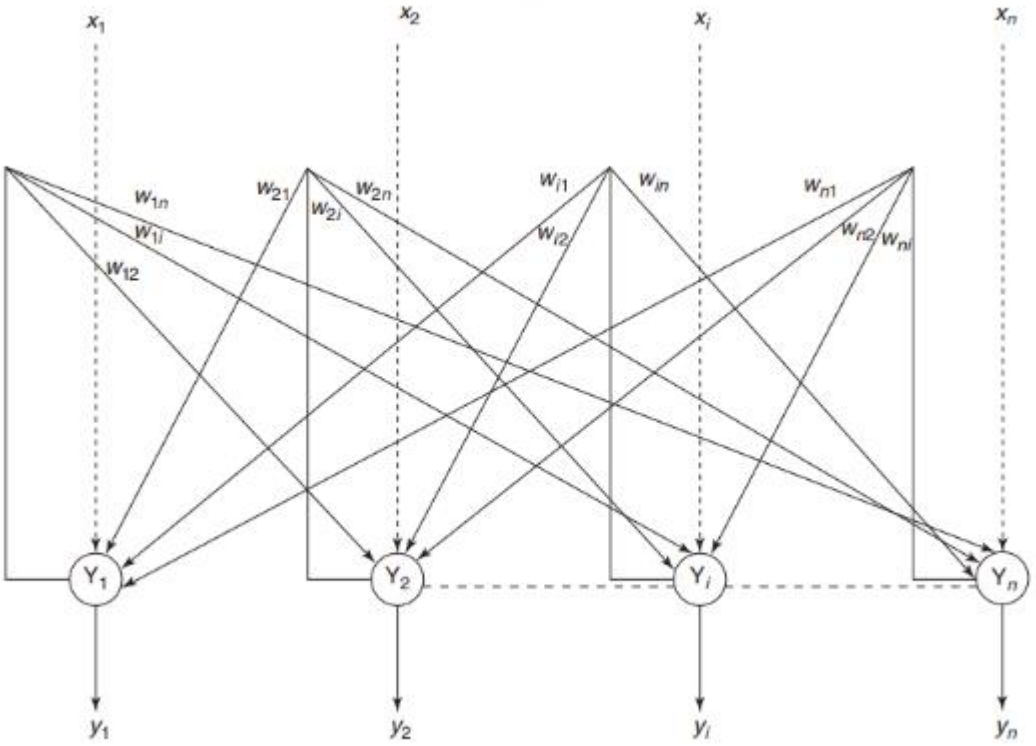
```
Associations:
Input: [1, -1, 1] => Output: [[1.0, 1.0]]
Input: [1, 1, -1] => Output: [[1.0, -1.0]]
Input: [-1, -1, 1] => Output: [[-1.0, 1.0]]
```

EXPERIMENT-7

AIM: To Implement Hopfield Network.

DESCRIPTION:

The Hopfield Network, pioneered by John Hopfield in 1982, is a recurrent neural network renowned for its associative memory and optimization capabilities. Comprising a single layer of neurons interconnected in a fully connected manner, these networks store patterns as stable states and retrieve them through iterative updates. Central to their operation is an energy function derived from Hebbian learning, measuring the compatibility of the current state with stored patterns. With their content-addressable memory system, Hopfield networks excel at recalling stored patterns from partial or noisy inputs, finding applications in diverse domains like pattern recognition, associative memory systems, and optimization tasks. Despite their effectiveness, Hopfield networks are limited by their capacity to store a finite number of patterns and may converge to spurious states unrelated to any stored pattern. Nonetheless, their associative memory capabilities make them invaluable tools for tasks requiring pattern recognition and optimization.



ALGORITHM:

Step 0: Initialize the weights to store patterns, i.e., weights obtained from training algorithm using Hebb rule.

Step 1: When the activations of the net are not converged, then perform Steps 2–8.

Step 2: Perform Steps 3–7 for each input vector X.

Step 3: Make the initial activations of the net equal to the external input vector X:

$$y_i = x_i \ (i = 1 \text{ to } n)$$

Step 4: Perform Steps 5–7 for each unit Y_i. (Here, the units are updated in random order.)

Step 5: Calculate the net input of the network:

$$y_{ini} = x_i + \sum_j y_j w_{ji}$$

Step 6: Apply the activations over the net input to calculate the output:

$$y_i = \begin{cases} 1 & \text{if } y_{ini} > \theta_i \\ y_i & \text{if } y_{ini} = \theta_i \\ -1 & \text{if } y_{ini} < \theta_i \end{cases}$$

where θ_i is the threshold and is normally taken as zero.

Step 7: Now feed back (transmit) the obtained output y_i to all other units. Thus, the activation vectors are updated.

Step 8: Finally, test the network for convergence.

CODE:

```
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).reshape(-1, 1)
            self.weights += np.dot(pattern, pattern.T)
        np.fill_diagonal(self.weights, 0)
        self.weights /= num_patterns
    def energy(self, pattern):
```

```
        pattern = np.array(pattern).reshape(-1, 1)
        return -0.5 * np.dot(pattern.T, np.dot(self.weights, pattern))
    def update(self, pattern, steps=1):
        pattern = np.array(pattern).reshape(-1, 1)
        for _ in range(steps):
            activation = np.dot(self.weights, pattern)
            pattern = np.where(activation >= 0, 1, -1)
        return pattern.T.tolist()
# Example usage
if __name__ == "__main__":
    # Training data for Hopfield network
    patterns = [[1, -1, 1, -1], [-1, 1, -1, 1], [1, 1, -1, -1]]
    # Create and train the Hopfield network
    hopfield_net = HopfieldNetwork(num_neurons=len(patterns[0]))
    hopfield_net.train(patterns)
    # Test the Hopfield network
    print("Energy of patterns:")
    for pattern in patterns:
        energy = hopfield_net.energy(pattern)
        print(f"Pattern: {pattern} => Energy: {energy}")
    print("\nAssociations:")
    for pattern in patterns:
        associated_pattern = hopfield_net.update(pattern)
        print(f"Pattern: {pattern} => Associated Pattern: {associated_pattern}")
```

OUTPUT:

```
Energy of patterns:
Pattern: [1, -1, 1, -1] => Energy: [[-3.33333333]]
Pattern: [-1, 1, -1, 1] => Energy: [[-3.33333333]]
Pattern: [1, 1, -1, -1] => Energy: [[-0.66666667]]

Associations:
Pattern: [1, -1, 1, -1] => Associated Pattern: [[1, -1, 1, -1]]
Pattern: [-1, 1, -1, 1] => Associated Pattern: [[-1, 1, -1, 1]]
Pattern: [1, 1, -1, -1] => Associated Pattern: [[1, 1, -1, -1]]
```

EXPERIMENT-8

AIM: To Implement Membership Functions in Fuzzy Sets

DESCRIPTION:

Fuzzy sets, conceived by Lotfi A. Zadeh in 1965, revolutionized how we handle uncertainty and vagueness in data. Unlike traditional sets that strictly categorize elements as either belonging or not, fuzzy sets introduce the concept of membership degrees, allowing elements to belong to a set to varying degrees. Central to fuzzy sets is the notion of a membership function, which assigns a degree of membership to each element, reflecting its level of belongingness. This flexibility enables fuzzy sets to capture nuances in data and accommodate imprecision inherent in real-world scenarios. Fuzzy sets support operations such as union, intersection, complement, and difference, extending classical set operations to handle fuzzy membership degrees. They find diverse applications in fields like artificial intelligence, control systems, decision-making, and pattern recognition, providing a natural framework for modeling uncertainty and facilitating human-like reasoning processes. Despite the need for careful design and tuning of membership functions, fuzzy sets offer a powerful and intuitive approach to dealing with uncertainty, contributing significantly to advancements in various domains.

CODE:

```
import numpy as np

class TriangularMembershipFunction:
    def __init__(self, lower_bound, peak, upper_bound):
        self.lower_bound = lower_bound
        self.peak = peak
        self.upper_bound = upper_bound

    def membership(self, x):
        if x < self.lower_bound or x > self.upper_bound:
            return 0
        elif self.lower_bound <= x < self.peak:
            return (x - self.lower_bound) / (self.peak - self.lower_bound)
        elif x == self.peak:
            return 1
        else:
            return (self.upper_bound - x) / (self.upper_bound - self.peak)
```

```
# Example usage
if __name__ == "__main__":
    # Define triangular membership functions for linguistic variables
    poor = TriangularMembershipFunction(0, 0, 5)
    average = TriangularMembershipFunction(0, 5, 10)
    good = TriangularMembershipFunction(5, 10, 10)

    # Test membership functions
    print("Membership of 3 in 'Poor':", poor.membership(3))
    print("Membership of 7 in 'Average':", average.membership(7))
    print("Membership of 9 in 'Good':", good.membership(9))
```

OUTPUT:

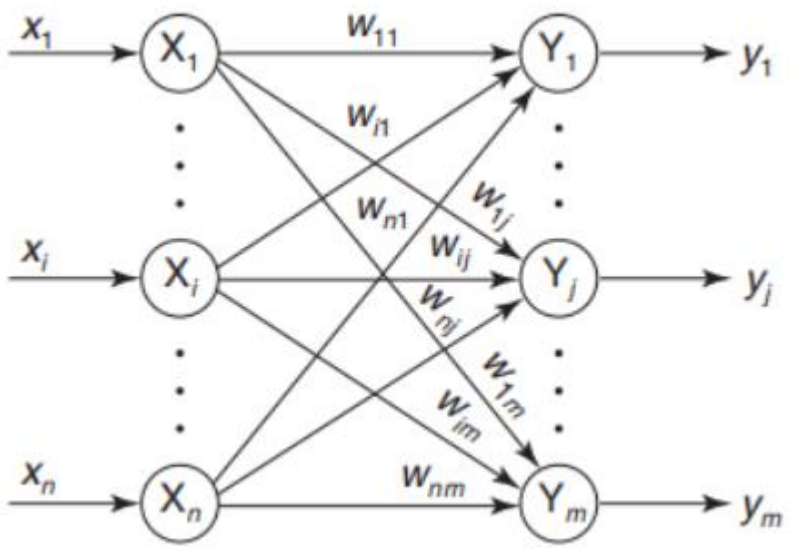
```
Membership of 3 in 'Poor': 0.4
Membership of 7 in 'Average': 0.6
Membership of 9 in 'Good': 0.8
```

EXPERIMENT-9

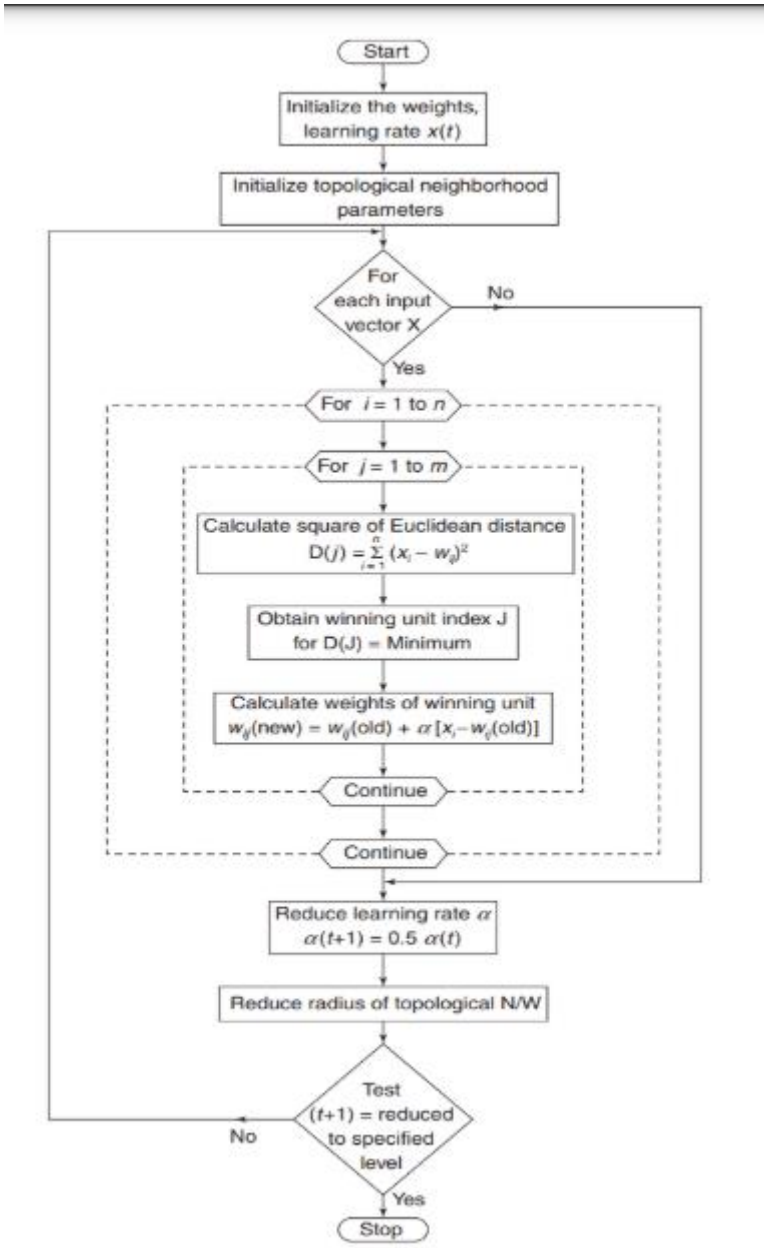
AIM: To implement the Kohonen Self-Organizing Feature Maps (KSOFM) network for Clustering

DESCRIPTION:

Feature mapping is a process which converts the patterns of arbitrary dimensionality into a response of one- or twodimensional arrays of neurons, i.e., it converts a wide pattern space into a typical feature space. The network performing such a mapping is called feature map. Apart from its capability to reduce the higher dimensionality, it has to preserve the neighborhood relations of the input patterns, i.e., it has to obtain a topology preserving map. For obtaining such feature maps, it is required to find a self-organizing neural array, which consists of neurons arranged in a one-dimensional array or a two-dimensional array. To depict this, consider a typical network structure where each component of the input vector x is connected to each of the nodes.



FLOWCHART:



ALGORITHM:

Step 0:
- Initialize the weights w_{ij}: Random values may be assumed. They can be chosen as the same range of values as the components of the input vector. If information related to distribution of clusters is known, the initial weights can be taken to reflect that prior knowledge.
- Set topological neighborhood parameters: As clustering progresses, the radius of the neighborhood decreases.
- Initialize the learning rate α: It should be a slowly decreasing function of time.

Step 1: Perform Steps 2–8 when stopping condition is false.

Step 2: Perform Steps 3–5 for each input vector x.

Step 3: Compute the square of the Euclidean distance, i.e., for each $j = 1$ to m,

$$D(j) = \sum_{i=1}^{n}\sum_{j=1}^{m}(x_i - w_{ij})^2$$

Step 4: Find the winning unit index J, so that $D(J)$ is minimum. (In Steps 3 and 4, dot product method can also be used to find the winner, which is basically the calculation of net input, and the winner will be the one with the largest dot product.)

Step 5: For all units j within a specific neighborhood of J and for all i, calculate the new weights:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})]$$

or

$$w_{ij}(\text{new}) = (1-\alpha)w_{ij}(\text{old}) + \alpha x_i$$

Step 6: Update the learning rate α using the formula $\alpha(t+1) = 0.5\alpha(t)$.

Step 7: Reduce radius of topological neighborhood at specified time intervals.

Step 8: Test for stopping condition of the network.

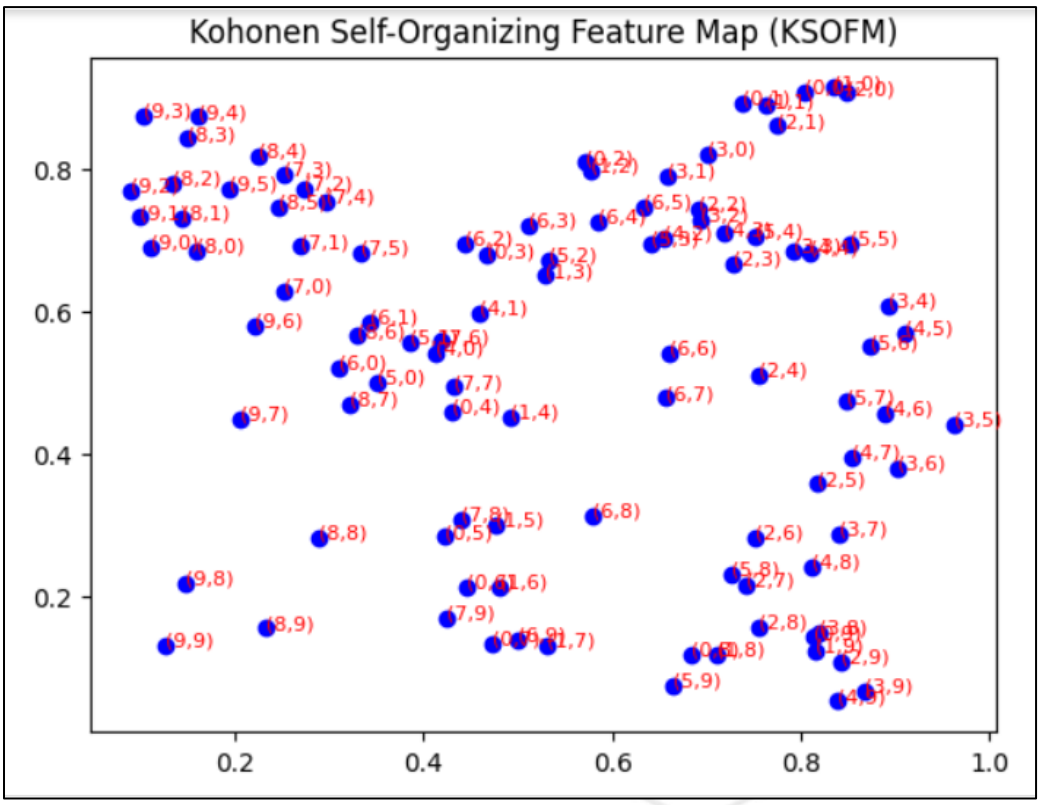
CODE:

```
import numpy as np
import matplotlib.pyplot as plt
class KSOFM:
  def __init__(self, input_dim, map_dim, initial_learning_rate, initial_radius,
num_iterations):
    self.input_dim = input_dim
    self.map_dim = map_dim
    self.initial_learning_rate = initial_learning_rate
    self.initial_radius = initial_radius
    self.num_iterations = num_iterations
    self.weights = np.random.rand(map_dim[0], map_dim[1], input_dim)
  def train(self, input_data):
    for iteration in range(self.num_iterations):
      for input_vector in input_data:
```

```
        distances = np.sum((self.weights - input_vector) ** 2, axis=-1)
        winning_unit = np.unravel_index(np.argmin(distances, axis=None),distances.shape)
        learning_rate = self.initial_learning_rate * np.exp(-iteration / self.num_iterations)
        neighborhood_radius = self.initial_radius * np.exp(-iteration / self.num_iterations)
        self.update_weights(input_vector, learning_rate, neighborhood_radius,winning_unit)
  def update_weights(self, input_vector, learning_rate, neighborhood_radius, winning_unit):
    for x in range(self.weights.shape[0]):
      for y in range(self.weights.shape[1]):
        distance_to_winner = np.linalg.norm(np.array([x, y]) - np.array(winning_unit))
        if distance_to_winner <= neighborhood_radius:
          influence = np.exp(-(distance_to_winner ** 2) / (2 * (neighborhood_radius ** 2)))
          self.weights[x, y, :] += learning_rate * influence * (input_vector - self.weights[x, y,:])
  def get_weights(self):
    return self.weights
  def plot_map(self):
    fig, ax = plt.subplots()
    for i in range(self.map_dim[0]):
      for j in range(self.map_dim[1]):
        ax.scatter(self.weights[i, j, 0], self.weights[i, j, 1], color='blue')
        ax.text(self.weights[i, j, 0], self.weights[i, j, 1], f'({i},{j})', color='red', fontsize=8)
    ax.set_title('Kohonen Self-Organizing Feature Map (KSOFM)')
    plt.show()
# Example usage
if __name__ == "__main__":
  # Generate some sample data
  input_data = np.random.rand(100, 3) # 100 input vectors of dimension 3
  # Define parameters
  input_dim = input_data.shape[1]
  map_dim = (10, 10) # Dimensions of the SOM grid
  initial_learning_rate = 0.1
  initial_radius = 5
  num_iterations = 1000
  # Train the KSOFM
```

```
ksofm = KSOFM(input_dim, map_dim, initial_learning_rate, initial_radius, num_iterations)
ksofm.train(input_data)
# Plot the final weights
ksofm.plot_map()
```

OUTPUT:



EXPERIMENT-10

AIM: Implement the Genetic Algorithm for the function f(x) = x2

DESCRIPTION:

Genetic algorithms (GAs) are optimization techniques inspired by the principles of natural selection and genetics. They start with a population of potential solutions encoded as strings of binary digits. These solutions undergo evaluation against a fitness function, which quantifies their quality. Through a process akin to natural selection, individuals with higher fitness are more likely to be selected for reproduction, where pairs of individuals exchange genetic material through crossover and introduce random changes through mutation. This continual process of selection, crossover, and mutation drives the evolution of the population toward better solutions. GAs excel in exploring large and complex search spaces, making them valuable tools in solving optimization problems across diverse fields like engineering, finance, and biology.

CODE:

```
import random
# Define the function to be optimized
def f(x):
  return x ** 2
# Generate an initial population
def generate_population(size, x_min, x_max):
  return [random.uniform(x_min, x_max) for _ in range(size)]
# Evaluate the fitness of each individual in the population
def evaluate_population(population):
  return [f(x) for x in population]
# Select parents using roulette wheel selection
def select_parents(population, fitnesses):
  total_fitness = sum(fitnesses)
  selection_probs = [fitness / total_fitness for fitness in fitnesses]
  parents = random.choices(population, weights=selection_probs, k=len(population))
  return parents
# Perform crossover to produce offspring
def crossover(parents, crossover_rate=0.7):
  offspring = []
  for i in range(0, len(parents), 2):
```

```
    parent1, parent2 = parents[i], parents[i+1]
    if random.random() < crossover_rate:
      # Single-point crossover
      crossover_point = random.randint(1, len(bin(int(parent1))) - 2)
      mask = (1 << crossover_point) - 1
      offspring1 = (int(parent1) & mask) | (int(parent2) & ~mask)
      offspring2 = (int(parent2) & mask) | (int(parent1) & ~mask)
      offspring.append(offspring1)
      offspring.append(offspring2)
    else:
      offspring.append(parent1)
      offspring.append(parent2)
  return offspring
# Apply mutation to the offspring
def mutate(offspring, mutation_rate=0.01, x_min=-10, x_max=10):
  for i in range(len(offspring)):
    if random.random() < mutation_rate:
      offspring[i] = random.uniform(x_min, x_max)
  return offspring
# Genetic Algorithm
def genetic_algorithm(pop_size, x_min, x_max, generations):
  # Step 1: Initialize the population
  population = generate_population(pop_size, x_min, x_max)
  for generation in range(generations):
    # Step 2: Evaluate the fitness of the population
    fitnesses = evaluate_population(population)
    # Step 3: Select parents
    parents = select_parents(population, fitnesses)
    # Step 4: Perform crossover
    offspring = crossover(parents)
    # Step 5: Apply mutation
    population = mutate(offspring)
    # Evaluate the new population
    fitnesses = evaluate_population(population)
```

```
    best_fitness = max(fitnesses)
    best_individual = population[fitnesses.index(best_fitness)]
    print(f"Generation {generation}: Best Fitness = {best_fitness}, Best Individual
={best_individual}")
# Parameters
population_size = 10
x_min = -10
x_max = 10
generations = 20
# Run the Genetic Algorithm
genetic_algorithm(population_size, x_min, x_max, generations)
```

OUTPUT:

```
Generation 0: Best Fitness = 100, Best Individual =-10
Generation 1: Best Fitness = 225, Best Individual =15
Generation 2: Best Fitness = 225, Best Individual =15
Generation 3: Best Fitness = 225, Best Individual =15
Generation 4: Best Fitness = 225, Best Individual =15
Generation 5: Best Fitness = 225, Best Individual =15
Generation 6: Best Fitness = 225, Best Individual =15
Generation 7: Best Fitness = 225, Best Individual =15
Generation 8: Best Fitness = 225, Best Individual =15
Generation 9: Best Fitness = 225, Best Individual =15
Generation 10: Best Fitness = 225, Best Individual =15
Generation 11: Best Fitness = 225, Best Individual =15
Generation 12: Best Fitness = 225, Best Individual =15
Generation 13: Best Fitness = 225, Best Individual =15
Generation 14: Best Fitness = 225, Best Individual =15
Generation 15: Best Fitness = 225, Best Individual =15
Generation 16: Best Fitness = 225, Best Individual =15
Generation 17: Best Fitness = 225, Best Individual =15
Generation 18: Best Fitness = 225, Best Individual =15
Generation 19: Best Fitness = 225, Best Individual =15
```