INDEX

SR. NO.	PRACTICAL NAME	DATE	SIGNATURE
1.	Fuzzy Logic	**5/10/19**	
2.	Implementation of AND/OR/NOT Gate using Single Layer Perceptron	**12/10/19**	
3.	Implementation of XOR Gate Using Multi-Layer Perceptron/ Error Back Propagation	**19/10/19**	
4.	Implementation of XOR Gate Using Radial Basis Function Network	**26/10/19**	
5.	Understanding the concepts of Perceptron Learning Rule	**2/11/19**	
6.	Understanding the concepts of Hebbiann Learning Rule	**9/11/19**	
7.	Understanding the concepts of Correlation Learning Rule	**16/11/19**	
8.	Understanding the working of Kohonen's Self Organising Maps	**23/11/19**	
9.	Understanding the functioning of Fuzzification process	**30/11/19**	
10.	Implementation of different method of Defuzzification process	**30/11/19**	

PRACTICAL: 1

AIM: Fuzzy Logic.

Code:

```
import numpy as np
class FuzzySet:

    def __init__(self, iterable: any):
        self.f_set = set(iterable)
        self.f_list = list(iterable)
        self.f_len = len(iterable)
        for elem in self.f_set:
            if not isinstance(elem, tuple):
                raise TypeError("No tuples in the fuzzy set...")
            if not isinstance(elem[1], float):
                raise ValueError("Probabilities not assigned to elements...")

    def __or__(self, other):
        # fuzzy set union
        if len(self.f_set) != len(other.f_set):
            raise ValueError("Length of the sets is different !!!")
        f_set = [x for x in self.f_set]
        other = [x for x in other.f_set]
        return FuzzySet([f_set[i] if f_set[i][1] > other[i][1] else other[i] for i in range(len(self))])

    def __and__(self, other):
        # fuzzy set intersection
        if len(self.f_set) != len(other.f_set):
            raise ValueError("Length of the sets is different !!!")
        f_set = [x for x in self.f_set]
        other = [x for x in other.f_set]
        return FuzzySet([f_set[i] if f_set[i][1] < other[i][1] else other[i] for i in range(len(self))])

    def __invert__(self):
        f_set = [x for x in self.f_set]
        for indx, elem in enumerate(f_set):
            f_set[indx] = (elem[0], float(round(1 - elem[1], 2)))
        return FuzzySet(f_set)

    def __sub__(self, other):
        if len(self) != len(other):
```

```
        raise ValueError("Length of the sets is different !!!")
    return self & ~other

def __mul__(self, other):
    if len(self) != len(other):
        raise ValueError("Length of the sets is different !!!")
    return FuzzySet([(self[i][0], self[i][1] * other[i][1]) for i in range(len(self))])

def __mod__(self, other):
    # cartesian product
    print(f'The size of the relation will be : {len(self)}x{len(other)}')
    mx = self
    mi = other
    tmp = [[] for i in range(len(mx))]
    i = 0
    for x in mx:
        for y in mi:
            tmp[i].append(min(x[1], y[1]))
        i += 1
    return np.array(tmp)

@staticmethod
def max_min(array1: np.ndarray, array2: np.ndarray):
    tmp = np.zeros((array1.shape[0], array2.shape[1]))
    t = list()
    for i in range(len(array1)):
        for j in range(len(array2[0])):
            for k in range(len(array2)):
                t.append(round(min(array1[i][k], array2[k][j]), 2))
            tmp[i][j] = max(t)
            t.clear()
    return tmp

def __len__(self):
    self.f_len = sum([1 for i in self.f_set])
    return self.f_len

def __str__(self):
    return f'{[x for x in self.f_set]}'

def __getitem__(self, item):
    return self.f_list[item]
```

```
    def __iter__(self):
        for i in range(len(self)):
            yield self[i]


a = FuzzySet({('x1', 0.5), ('x2', 0.7), ('x3', 0.0)})
b = FuzzySet({('x1', 0.8), ('x2', 0.2), ('x3', 1.0)})
c = FuzzySet({('x', 0.3), ('y', 0.3), ('z', 0.5)})
x = FuzzySet({('a', 0.5), ('b', 0.3), ('c', 0.7)})
y = FuzzySet({('a', 0.6), ('b', 0.4)})

print(f'a -> {a}')
print(f'b -> {b}')
print('-----------------------------------------------------------------------')
print(f'Fuzzy union : \n{a | b}')
print('-----------------------------------------------------------------------')
print(f'Fuzzy intersection : \n{a & b}')
print('-----------------------------------------------------------------------')
print(f"Fuzzy inversion of a : \n {~a}")
print('-----------------------------------------------------------------------')
print(f'Fuzzy inversion of b : \n{~b}')
print('-----------------------------------------------------------------------')
print(f'Fuzzy Subtraction : \n{a - b}')
print('-----------------------------------------------------------------------')

r = np.array([[0.6, 0.6, 0.8, 0.9], [0.1, 0.2, 0.9, 0.8], [0.9, 0.3, 0.4, 0.8], [0.9, 0.8, 0.1, 0.2]])
s = np.array([[0.1, 0.2, 0.7, 0.9], [1.0, 1.0, 0.4, 0.6], [0.0, 0.0, 0.5, 0.9], [0.9, 1.0, 0.8, 0.2]])
print(f"Max Min of \n{r} \nand \n{s}\n\nis as follows : \n")
print(FuzzySet.max_min(r, s))
print('-----------------------------------------------------------------------')
```

Output:

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 19:29:22) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\student\Downloads\Soft-Computing-ANN-Lab-master\fuzzy.py =
a -> [('x1', 0.5), ('x3', 0.0), ('x2', 0.7)]
b -> [('x2', 0.2), ('x1', 0.8), ('x3', 1.0)]
------------------------------------------------------------------------
Fuzzy union :
[('x1', 0.5), ('x1', 0.8), ('x3', 1.0)]
------------------------------------------------------------------------
Fuzzy intersection :
[('x2', 0.2), ('x3', 0.0), ('x2', 0.7)]
------------------------------------------------------------------------
Fuzzy inversion of a :
 [('x1', 0.5), ('x2', 0.3), ('x3', 1.0)]
------------------------------------------------------------------------
Fuzzy inversion of b :
[('x1', 0.2), ('x3', 0.0), ('x2', 0.8)]
------------------------------------------------------------------------
Fuzzy Subtraction :
[('x1', 0.2), ('x3', 0.0), ('x2', 0.7)]
------------------------------------------------------------------------
Max Min of
[[0.6 0.6 0.8 0.9]
 [0.1 0.2 0.9 0.8]
 [0.9 0.3 0.4 0.8]
 [0.9 0.8 0.1 0.2]]
and
[[0.1 0.2 0.7 0.9]
 [1.  1.  0.4 0.6]
 [0.  0.  0.5 0.9]
 [0.9 1.  0.8 0.2]]

is as follows :

[[0.9 0.9 0.8 0.8]
 [0.8 0.8 0.8 0.9]
 [0.8 0.8 0.8 0.9]
 [0.8 0.8 0.7 0.9]]
------------------------------------------------------------------------
>>>
```

PRACTICAL: 2

AIM: To understand the working of neural networks using AND, OR, NOT Gates implemented through a single neuron of the neural network.

Theory:

Activation functions:-

In computational networks, the activation function of a node defines the output of that node given input or set of inputs.

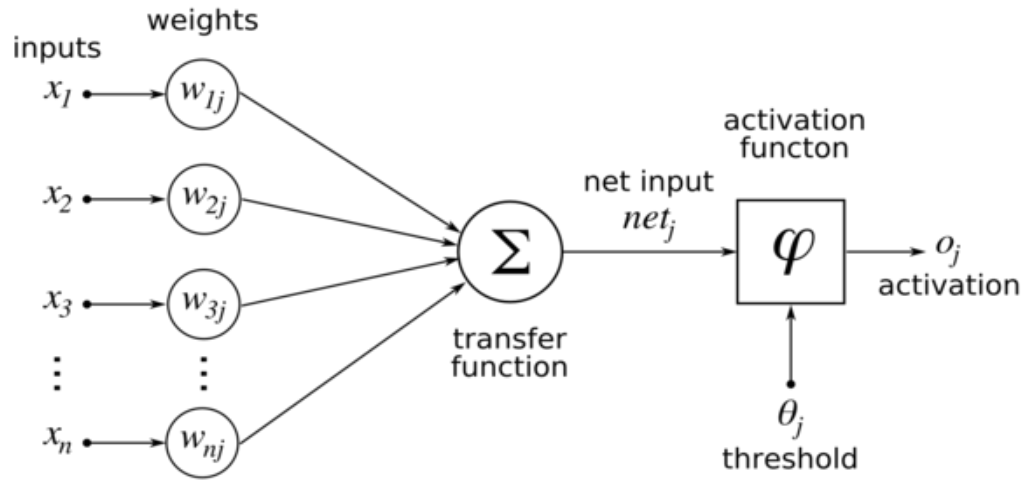


Fig 1. General structure of an artificial neural network with a single perceptron.

Types of activation functions:-

1. Hard-limit Activation Function

$$\varphi^{\text{hlim}}(v) = \begin{cases} 1 & \text{for } v \geq 0 \\ 0 & \text{for } v < 0 \end{cases}$$

2. Soft-limit (Sigmoidal) Activation Function

$$\varphi_a^{\text{sig}}(v) = \frac{1}{1 + \exp(-av)}.$$

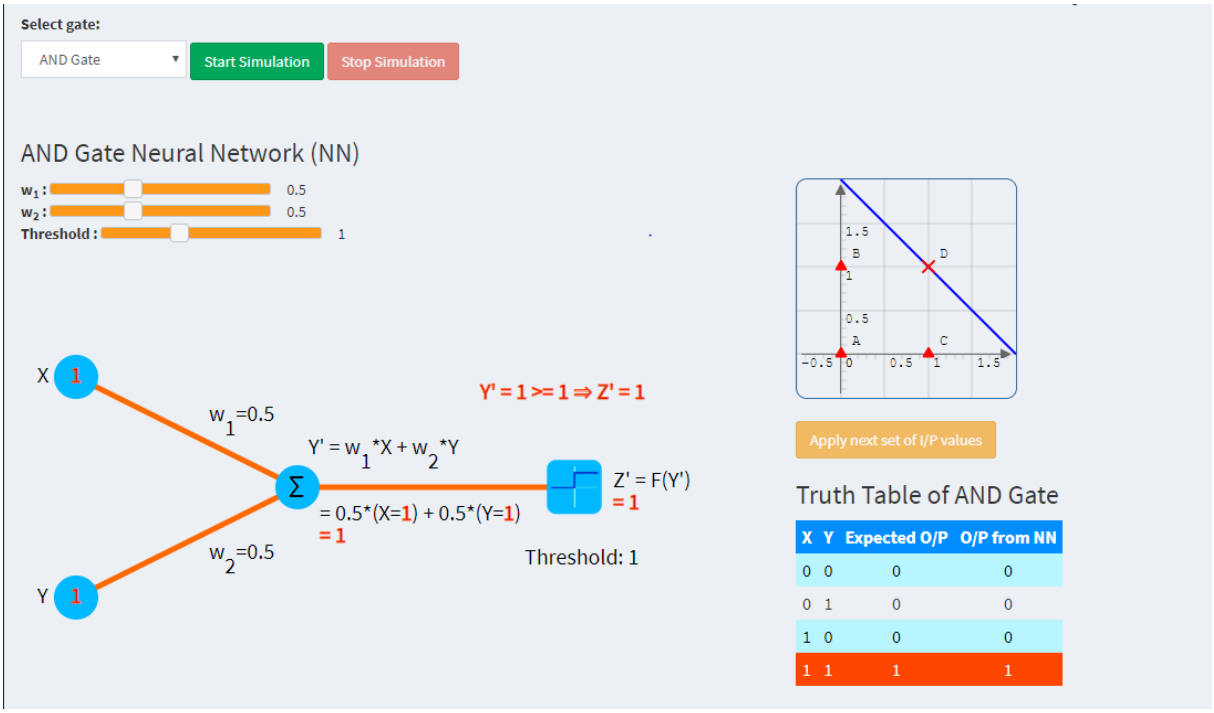
3. Piecewise Linear Activation Function

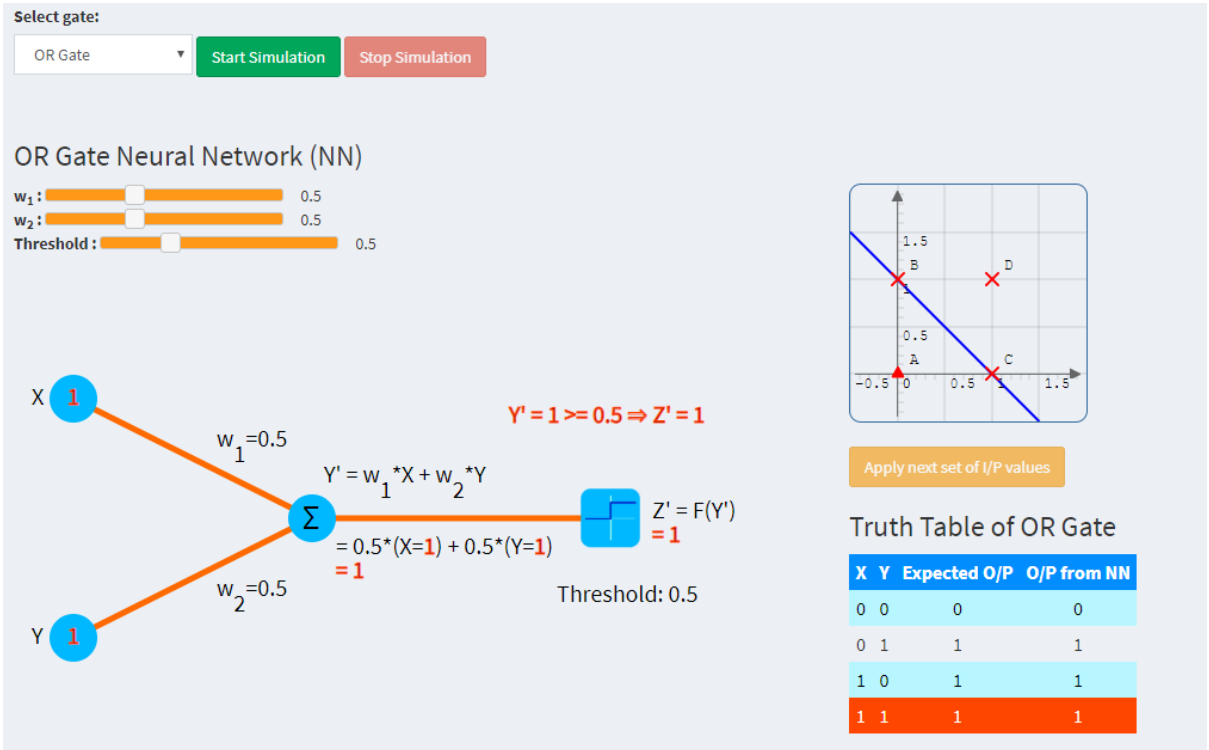
$$\varphi^{\text{pwl}}(v) = \begin{cases} 1 & \text{for } v \geq \frac{1}{2} \\ v + \frac{1}{2} & \text{for } -\frac{1}{2} < v < \frac{1}{2} \\ 0 & \text{for } v \leq -\frac{1}{2} \end{cases}$$

4. Signum Activation Function

$$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$$

OUTPUT:


Select gate:
AND Gate
Start Simulation
Stop Simulation
AND Gate Neural Network (NN)
w1 : 0.5
w2 : 0.5
Threshold : 1
X 1
Y 1
w1=0.5
w2=0.5
Y' = w1*X + w2*Y
= 0.5*(X=1) + 0.5*(Y=1)
= 1
Y' = 1 >= 1 ⇒ Z' = 1
Z' = F(Y')
= 1
Threshold: 1
Apply next set of I/P values
Truth Table of AND Gate
X Y Expected O/P O/P from NN
0 0 0 0
0 1 0 0
1 0 0 0
1 1 1 1


Select gate:
OR Gate
Start Simulation
Stop Simulation
OR Gate Neural Network (NN)
w1 : 0.5
w2 : 0.5
Threshold : 0.5
X 1
Y 1
w1=0.5
w2=0.5
Y' = w1*X + w2*Y
= 0.5*(X=1) + 0.5*(Y=1)
= 1
Y' = 1 >= 0.5 ⇒ Z' = 1
Z' = F(Y')
= 1
Threshold: 0.5
Apply next set of I/P values
Truth Table of OR Gate
X Y Expected O/P O/P from NN
0 0 0 0
0 1 1 1
1 0 1 1
1 1 1 1

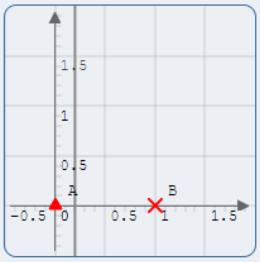
Select gate:

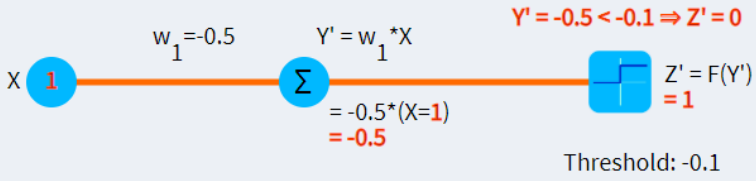
NOT Gate ▾ Start Simulation Stop Simulation

NOT Gate Neural Network (NN)

w_1 : -0.5

Threshold : -0.1





Threshold: -0.1

Apply next set of I/P values

Truth Table of NOT Gate

X	Expected O/P	O/P from NN
0	1	1
1	0	0

PRACTICAL: 3

AIM: Implementation of XOR Gate Using Multi-Layer Perceptron/ Error Back Propagation.

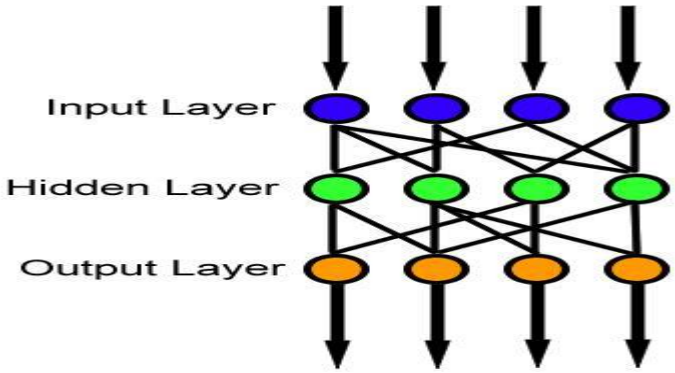
Theory:

A Multi-Layer Perceptron (MLP) is a feed forward artificial neural network model that maps sets of input data onto a set of appropriate outputs. An MLP consists of multiple layers of nodes in a directed graph, with each layer fully connected to the next one. Except for the input nodes, each node is a neuron (or processing element) with a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training the network. MLP is a modification of the standard linear perceptron and can distinguish data that are not linearly separable.

The multilayer perceptron consists of three or more layers (an input and an output layer with one or more hidden layers) of nonlinearly-activating nodes and is thus considered a deep neural network. Since an MLP is a Fully Connected Network, each node in one layer connects with a certain weight wij t{\display style w_{ij}}to every node in the following layer. Some people do not include the input layer when counting the number of layers and there is disagreement about whether wij {\display style w_{ij}}should be interpreted as the weight from i to j or the other way around.

Feed Forward Multi Layer Perceptron:-

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has direct connections to the neurons of the subsequent layer. In many applications, the units of these networks apply a sigmoid function as an activation function. In the mathematical theory of artificial neural networks, the Universal Approximation Theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of Rn, under mild assumptions on the activation function. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters.

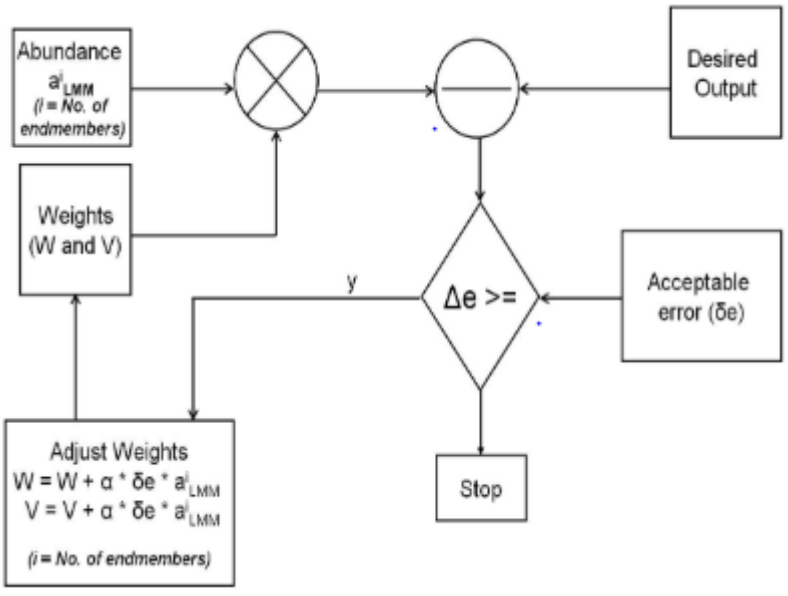


Error Back Propagation MLP:-

The Error Backward Propagation or back propagation is a common method of training artificial neural networks and used in conjunction with an optimization method such as gradient descent. The algorithm repeats a two phase cycle, propagation and weight update. When an input vector is presented to the network, it is propagated forward through the network, layer by layer, until it reaches the output layer. The output of the network is then compared to the desired output, using a loss function, and an error value is calculated for each of the neurons in the output layer. The error values are then propagated backwards, starting from the output, until each neuron has an associated error value which roughly represents its contribution to the original output.

Back propagation uses these error values to calculate the gradient of the loss function with respect to the weights in the network. In the second phase, this gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function. The importance of this process is that, as the network is trained, the neurons in the intermediate layers organize themselves in such a way that the different neurons learn to recognize different characteristics of the total input space. After training, when an arbitrary input pattern is present which contains noise or is incomplete, neurons in the hidden layer of the network will respond with an active output if the new input contains a pattern that resembles a feature that the individual neurons have learned to recognize during their training.

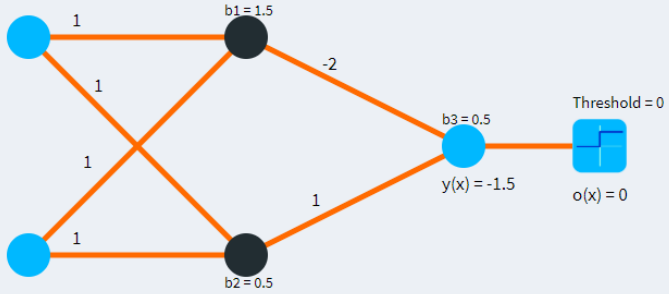
Back propagation requires a known, desired output for each input value in order to calculate the loss function gradient – it is therefore usually considered to be a supervised learning method; nonetheless, it is also used in some unsupervised networks such as auto encoders. It is a generalization of the delta rule to multi-layered feed forward networks, made possible by using the chain rule to iteratively compute gradients for each layer. Back propagation requires that the activation function used by the artificial neurons (or "nodes") be differentiable.



OUTPUT:

Select a network:

Multi-Layer Perceptron



Restart simulation

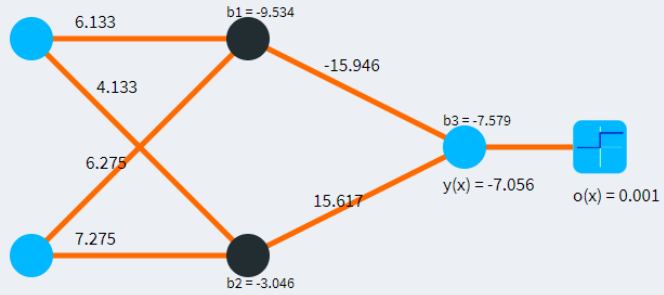
Truth Table

Input		Output			
X1	X2	Output of hidden neuron 1	Output of hidden neuron 2	Final Network Output	Expected Output
0	0	0	0	0	0
0	1	0	1	1	1
1	0	0	1	1	1
1	1	1	1	0	0

Accuracy of network: 100%

Select a network:

Error Back Propogation



Set Learning rate: 0.8

No. of iterations? → 1000000

No. of iterations completed: 1000000. Click Reset to try again with different values.

Start simulation Reset

Truth Table

Input		Output			
X1	X2	Output of hidden neuron 1	Output of hidden neuron 2	Final Network Output	Expected Output
0	0	0.000	0.045	0.001	0
0	1	0.037	0.986	0.999	1
1	0	0.032	0.962	0.999	1
1	1	0.947	1.000	0.001	0

Root mean square error: 0.091%

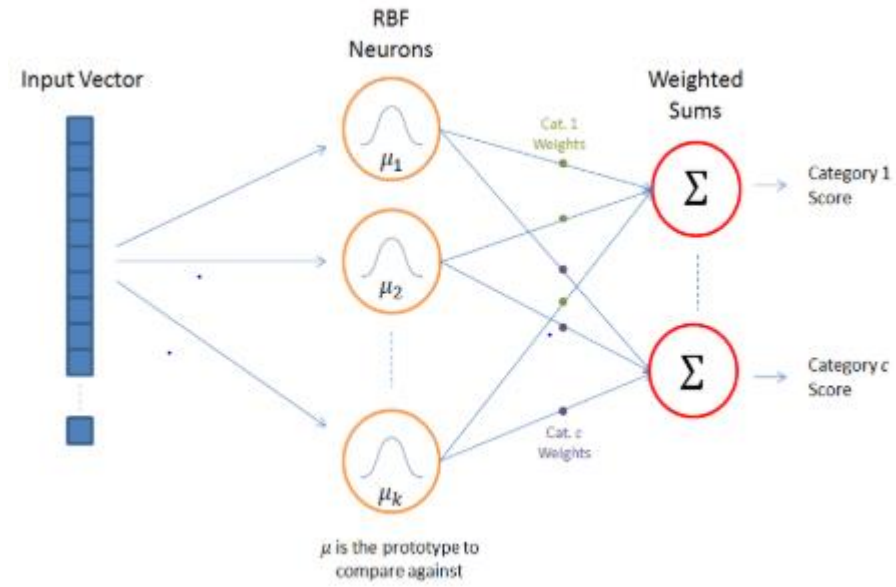
PRACTICAL: 4

AIM: Implementation of XOR Gate Using Radial Basis Function Network.

Theory:

A Radial Basis Function Network (RBFN**)** is a particular type of neural network. The RBFN approach is more intuitive than MLP. An RBFN performs classification by measuring the input's similarity to examples from the training set. Each RBFN neuron stores a "prototype", which is just one of the examples from the training set. When we want to classify a new input, each neuron computes the Euclidean distance between the input and its prototype. Thus, if the input more closely resembles the class A prototypes than the class B prototypes, it is classified as class A.

RBF Network Architecture:



It consist of an input vector, a layer of RBF neurons, and an output layer with one node per category or class of data.

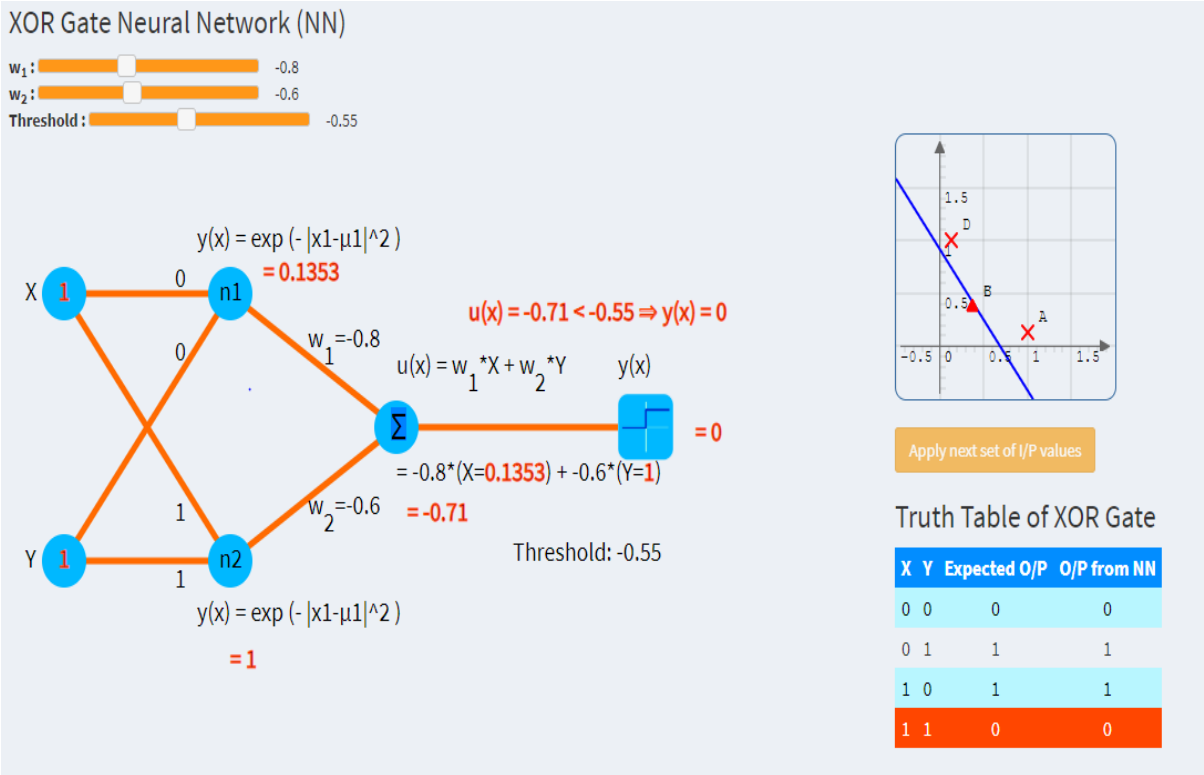
RBF Neuron Activation Function:

Each RBF neuron computes a measure of the similarity between the input and its prototype vector (taken from the training set). Input vectors which are more similar to the prototype return a result closer to 1. There are different possible choices of similarity functions, but the most popular is based on the Gaussian. Below is the equation for a Gaussian with a one-dimensional input.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Where x is the input, mu is the mean, and sigma is the standard deviation.

OUTPUT:


XOR Gate Neural Network (NN)
w1 : -0.8
w2 : -0.6
Threshold : -0.55
y(x) = exp (- |x1-μ1|^2)
= 0.1353
X
1
0
n1
0
w1 =-0.8
u(x) = -0.71 < -0.55 ⇒ y(x) = 0
u(x) = w1*X + w2*Y
y(x)
Σ
= 0
= -0.8*(X=0.1353) + -0.6*(Y=1)
1
w2 =-0.6
= -0.71
Y
1
n2
1
Threshold: -0.55
y(x) = exp (- |x1-μ1|^2)
= 1
1.5
D
0.5
B
A
-0.5
0
0.5
1
1.5
Apply next set of I/P values
Truth Table of XOR Gate
X Y Expected O/P O/P from NN
0 0 0 0
0 1 1 1
1 0 1 1
1 1 0 0

PRACTICAL: 5

AIM: To understand the perceptron learning rule that can be applied for supervised learning of neural network.

Theory:

In machine learning, the perceptron is an algorithm for supervised learning of binary classifiers. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. The algorithm allows for online learning, in that it processes elements in the training set one at a time.
Perceptrons are trained on examples of the desired behaviour. The desired behaviour can be summarized by a set of input, output pairs

p1t1, p2t2, p3t3, p4t4...pntn

where **p** is input to the network and **t** is the corresponding correct (target) output. The objective is to reduce the error **e**, which is the difference **t-a** between the neuron response **a**, and the target vector **t**. The perceptron learning rule calculates desired changes to the perceptron's weights and biases given an input vector **p**, and the associated error **e**. The target vector t must contain values of either **-1** or **1**, as perceptrons (with signum activation functions) can only output such values.
As each iteration goes on, the perceptron has a better chance of producing the correct outputs. The perceptron rule is proven to converge on a solution in a finite number of iterations if a solution exists.
If a bias is not used, learning algorithm works to find a solution by altering only the weight vector **w** to point toward input vectors to be classified as **1**, and away from vectors to be classified as **-1.** This results in a decision boundary that is perpendicular to **w**, and which properly classifies the input vectors.
There are three conditions that can occur for a single neuron once an input vector **p** is presented and the network's response **a** is calculated:

CASE 1: If an input vector is presented and the output of the neuron is correct (**a** = **t**, and **e** = **t - a** = **0**), then the weight vector **w** is not altered.

CASE 2: If the neuron output is **-1** and should have been **1** (**a** = **-1** and **t** = **1**, and **e** = **t - a** = **2**), the input vector **p** is added to the weight vector **w**. This makes the weight vector point closer to the input vector, increasing the chance that the input vector will be classified as a **1** in the future.

CASE 3: If the neuron output is **1** and should have been **-1** (**a** = **1** and **t** = **-1**, and **e** = **t - a** = **-2**), the input vector **p** is subtracted from the weight vector **w**. This makes the weight vector point farther away from the input vector, increasing the chance that the input vector is classified as a **-1** in the future.

The perceptron learning rule can be written more succinctly in terms of the error **e** = **t** - **a**, and the change to be made to the weight vector **w**:

CASE 1. If **e** = **0**, then make a change in w equal to **0**.
CASE 2. If **e** = **1**, then make a change in w equal to $\mathbf{2p^T}$.

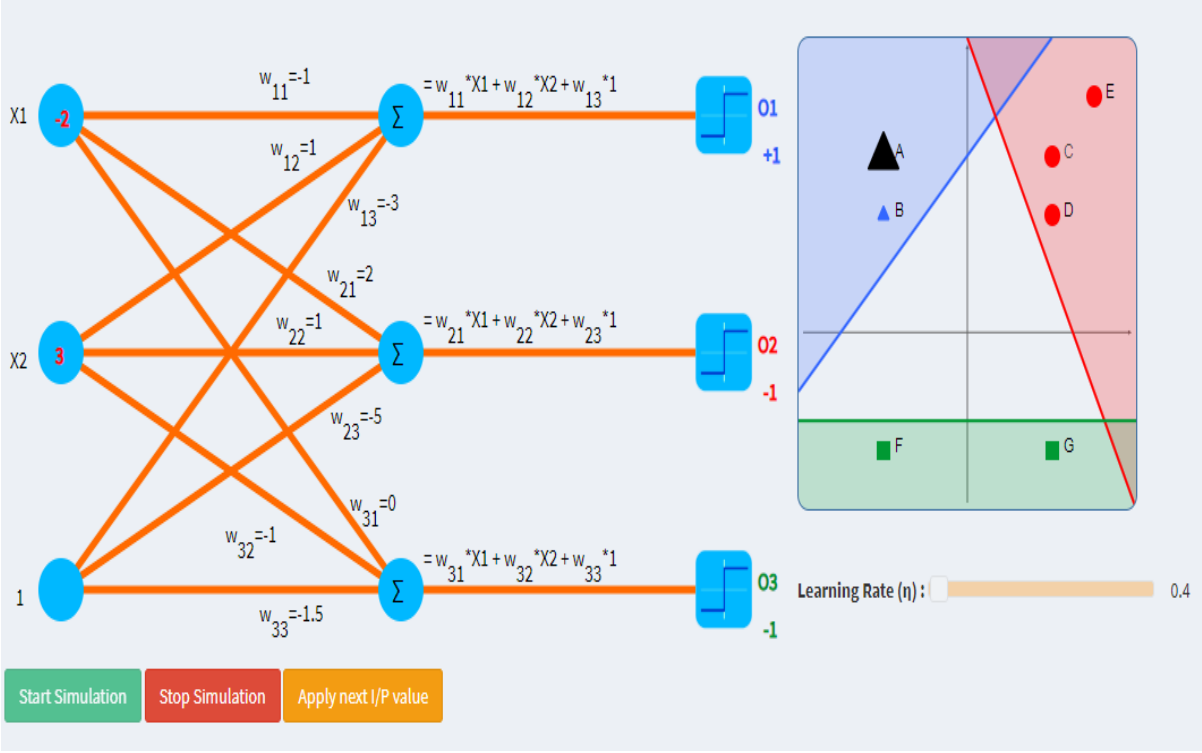
CASE 3. If **e** = **-1**, then make a change in w equal to $\mathbf{-2p^T}$.

According to **Perceptron** Learning Rule,

$$\mathbf{W_{new} = W_{old} + \Delta w}$$

where $(\mathbf{\Delta w = e * p^T})$ or $\mathbf{\Delta W_i = \eta\ (\ D_i - O_i\ )\ X}$

OUTPUT:



Calculations:

$$O = \text{sgn}(W \times X)$$

$$O = \text{sgn}\left(\begin{bmatrix} -1 & 1 & -3 \\ 2 & 1 & -5 \\ 0 & -1 & -1.5 \end{bmatrix} \times \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}\right) = \text{sgn}\left(\begin{bmatrix} 2 \\ -6 \\ -4.5 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

$$O = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, D = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

According to **Perceptron Learning Rule** : $\Delta W_i = \eta (D_i - O_i) X$

Hence, $W_{i,new} = W_{i,old} + \eta (D_i - O_i) X$

The calculations for weight vector for each classifier neuron are as shown below:

In the carousel, the cards with green background indicate the corresponding **weight vector has not changed** whereas the cards with red background indicate the corresponding **weight vector has changed.**

Hence, the new weight vectors are (Refer to the carousel above) :

$$W_{1,new} = \begin{bmatrix} -1 & 1 & -3 \end{bmatrix} \quad W_{2,new} = \begin{bmatrix} 2 & 1 & -5 \end{bmatrix} \quad W_{3,new} = \begin{bmatrix} 0 & -1 & -1.5 \end{bmatrix}$$

Thus, the new weight matrix becomes:

$$\begin{bmatrix} -1 & 1 & -3 \\ 2 & 1 & -5 \\ 0 & -1 & -1.5 \end{bmatrix}$$

The weight matrix **hasn't changed** and hence the graph remains unchanged.

PRACTICAL: 6

AIM: To understand the Hebbian learning rule that can be applied for supervised learning of neural network.

Theory:

Hebbian learning is one of the oldest learning algorithms and is based in large part on the dynamics of biological systems.

The general idea is that any two cells or systems of cells that are repeatedly active at the same time will tend to become 'associated' so that activity in one facilitates activity in the other.

A synapse between two neurons is strengthened when the neurons on either side of the synapse (input and output) have highly correlated outputs.

In essence, when an input neuron fires, if it frequently leads to the firing of the output neuron, the synapse is strengthened. Following the analogy to an artificial system, the tap weight is increased with a high correlation between two sequential neurons.

From the point of view of artificial neurons and artificial neural networks, Hebb's principle can be described as a method of determining how to alter the weights between model neurons. The weight between two neurons increases if the two neurons activate simultaneously, and reduces if they activate separately. Nodes that tend to be either both positive or both negative at the same time have strong positive weights, while those that tend to be opposite have strong negative weights.

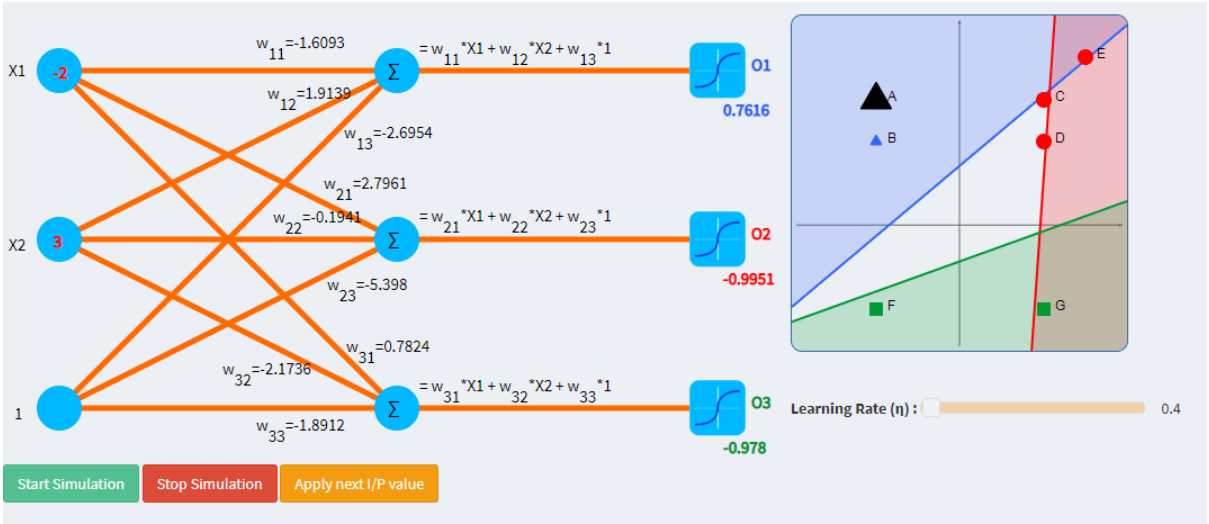
According to Hebbian Rule,

$\Delta W_i = \eta (O_i) X$

or the change in the synaptic weights of the i^{th} neuron W_i is equal to a learning rate η times the i^{th} output O_i times the input X. Often cited is the case of a linear neuron output y or O,

$$y = \sum_j w_j x_j,$$

OUTPUT:



Here, the activation function used is **tan sigmoid**:

$$f(x) = \frac{2}{1+e^{-x}} - 1$$

To understand what calculations are happening check this popup and to understand the representations of vectors and matrices click here or hover your mouse over them.

Calculations:

$O = f(W \times X)$

$$O = f\left(\begin{bmatrix} -1 & 1 & -3 \\ 2 & 1 & -5 \\ 0 & -1 & -1.5 \end{bmatrix} X \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}\right) = f\left(\begin{bmatrix} 2 \\ -6 \\ -4.5 \end{bmatrix}\right) = \begin{bmatrix} 0.7616 \\ -0.9951 \\ -0.978 \end{bmatrix}$$

$$O = \begin{bmatrix} 0.7616 \\ -0.9951 \\ -0.978 \end{bmatrix}, D = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

According to **Hebbian Learning Rule** : $\Delta W_i = \eta (O_i) X$

Hence, $W_{i,new} = W_{i,old} + \eta (O_i) X$

The calculations for weight vector for each classifier neuron are as shown below:

Hence, the new weight vectors are (Refer to the carousel above) :

$$W_{1,new} = \begin{bmatrix} -1.6093 & 1.9139 & -2.6954 \end{bmatrix} \quad W_{2,new} = \begin{bmatrix} 2.7961 & -0.1941 & -5.398 \end{bmatrix} \quad W_{3,new} = \begin{bmatrix} 0.7824 & -2.1736 & -1.8912 \end{bmatrix}$$

Thus, the new weight matrix becomes:

$$\begin{bmatrix} -1.6093 & 1.9139 & -2.6954 \\ 2.7961 & -0.1941 & -5.398 \\ 0.7824 & -2.1736 & -1.8912 \end{bmatrix}$$

The weight matrix has **changed** and hence the graph will also change.

PRACTICAL: 7

AIM: To understand the correlation learning rule that can be applied for supervised learning of neural network.

Theory:

The correlation learning rule is based on a similar principle to the Hebbian learning rule. It assumes that weights between simultaneously responding neurons should be largely positive, and weights between neurons with opposite reaction should be largely negative. Contrary to the Hebbian rule, the correlation rule is supervised learning. Instead of an actual response, $\mathbf{O_i}$, the desired response, $\mathbf{D_i}$, is used for the weight-change calculation.

According to **Correlation** Rule,

$$\Delta w_{ij} = \eta \, (D_i) \, X_j$$

where $\mathbf{D_i}$ is the desired value of the output signal. This training algorithm usually starts with the initialization of weights to zero.

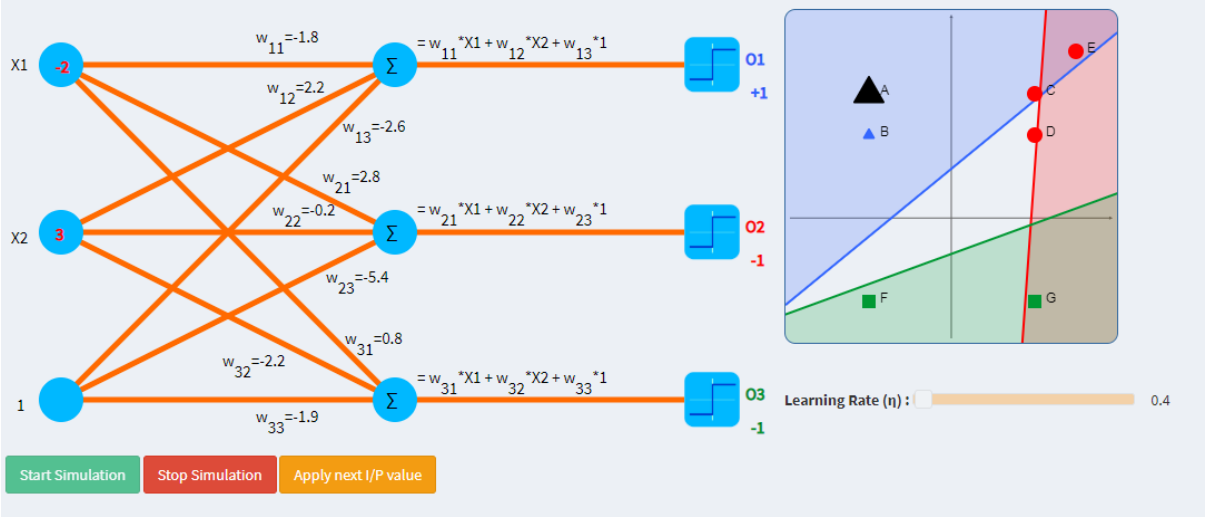
This simple rule states that if $\mathbf{D_i}$ is the desired response due to **X**, the corresponding weight increase is proportional to their product. The rule typically applies to record data in memory networks with binary response neurons. It can be interpreted as a special case of the Hebbian rule with a binary activation function and for $\mathbf{O_i = D_i}$.

However, Hebbian learning is performed in an unsupervised environment, while correlation learning is supervised.

While keeping this basic difference in mind, we can observe that Hebbian rule weight adjustment and correlation rule weight adjustment become identical.

Similar to the Hebbian learning rule, this learning rule also requires the weight initialization $\mathbf{w = 0}$.

OUTPUT:



Calculations:

$$O = f(W \times X)$$

$$O = f\left(\begin{bmatrix} -1 & 1 & -3 \\ 2 & 1 & -5 \\ 0 & -1 & -1.5 \end{bmatrix} X \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}\right) = f\left(\begin{bmatrix} 2 \\ -6 \\ -4.5 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

$$O = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}, D = \begin{bmatrix} 1 \\ -1 \\ -1 \end{bmatrix}$$

According to **Correlation Learning Rule** : $\Delta W_i = \eta (D_i) X$

Hence, $W_{i,new} = W_{i,old} + \eta (D_i) X$

The calculations for weight vector for each classifier neuron are as shown below:

Hence, the new weight vectors are (Refer to the carousel above) :

$$W_{1,new} = \begin{bmatrix} -1.8 & 2.2 & -2.6 \end{bmatrix} \quad W_{2,new} = \begin{bmatrix} 2.8 & -0.2 & -5.4 \end{bmatrix} \quad W_{3,new} = \begin{bmatrix} 0.8 & -2.2 & -1.9 \end{bmatrix}$$

Thus, the new weight matrix becomes:

$$\begin{bmatrix} -1.8 & 2.2 & -2.6 \\ 2.8 & -0.2 & -5.4 \\ 0.8 & -2.2 & -1.9 \end{bmatrix}$$

The weight matrix has **changed** and hence the graph will also change.

PRACTICAL: 8

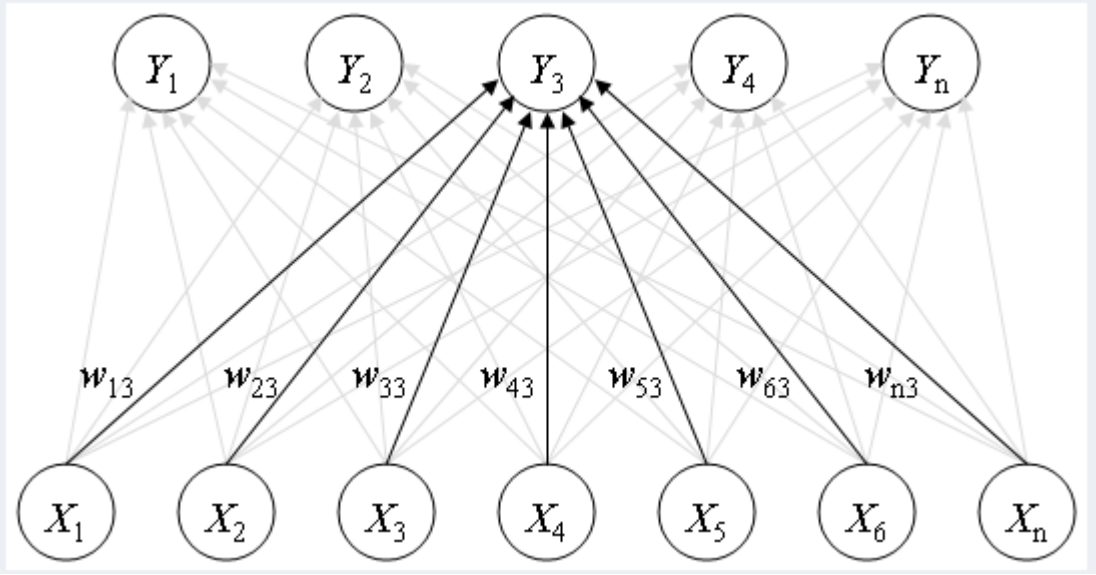
AIM: To understand the importance of Kohonen's Self Organizing Maps and the application of the same in Machine Learning, and to demonstrate the implementation of Kohonen's SOM over a set of 2 dimensional data.

Theory:

The Self-Organizing Map is one of the most popular neural network models. It belongs to the category of competitive learning networks. The Self-Organizing Map is based on unsupervised learning, which means that no human intervention is needed during the learning and that little needs to be known about the characteristics of the input data. We could, for example, use the SOM for clustering data without knowing the class memberships of the input data.

Self-organizing neural networks are used to cluster input patterns into groups of similar patterns. They're called "maps" because they assume a topological structure among their cluster units; effectively mapping weights to input data. The artificial neural network introduced by the Finnish professor Teuvo Kohonen in the 1980s is sometimes called a **Kohonen map** or **network**. The Kohonen network is probably the best example, because it's simple, yet introduces the concepts of self-organization and unsupervised learning easily. Each weight is representative of a certain input. Input patterns are shown to all neurons simultaneously.

The structure of a self-organizing map involves m output neurons, which correspond to m output clusters, and n input neurons which correspond to the n-dimensionality of the dataset.



The weight vectors define each cluster. Input patterns are compared to each cluster, and associated with the cluster it best matches. The comparison is usually based on the square of the minimum Euclidean distance. When the best match is found, the associated cluster gets its weights updated. KSOM is based on the Winner-Takes-All learning rule.

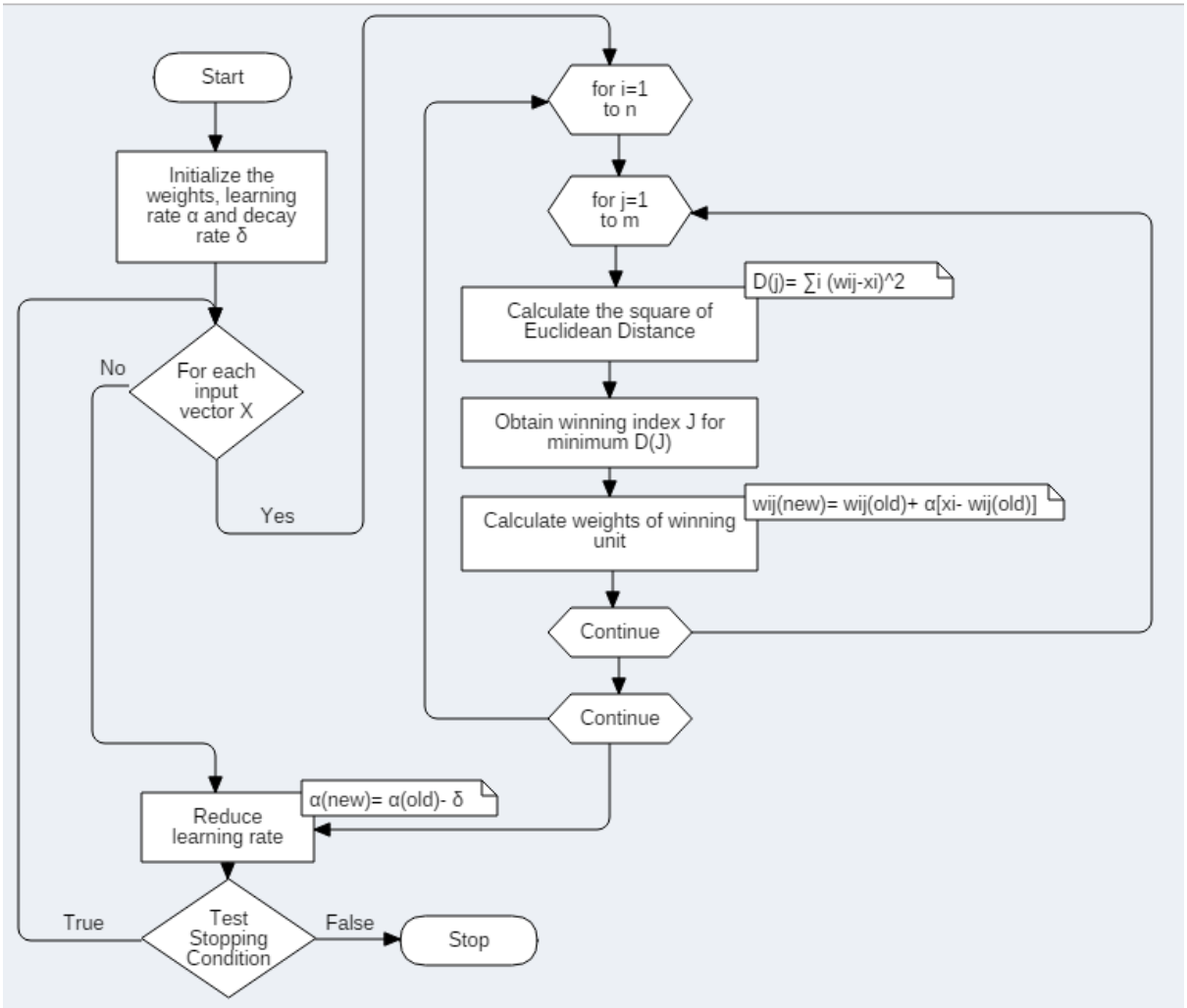
The learning rate α alpha is slowly decreased with each epoch. The size or radius of the neighbourhood around a cluster unit can also decrease during the later epochs.

The formation of a map occurs in two stages:

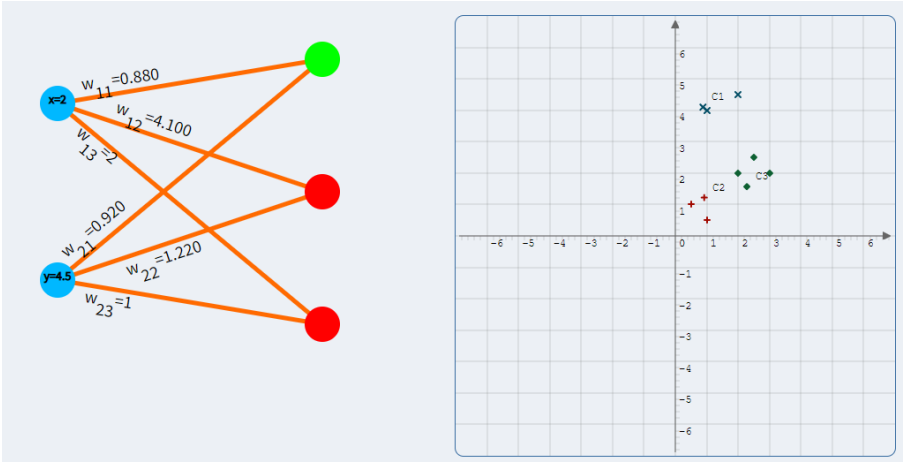
1. The initial formation of the correct order
2. The final convergence

The second stage takes much longer and usually occurs when the learning rate gets smaller. The initial weights can be random values.

Following is the flowchart which shows the algorithm used in KSOM:



OUTPUT:



Samples & their clusters:

Sample	Cluster
(2, 2)	3
(2.5, 2.5)	3
(3, 2)	3
(0.5, 1)	2
(1, 0.5)	2
(1, 4)	1
(2, 4.5)	1

Calculations:

Input sample into consideration: (2,4.5)

The initial weight matrix:

$$\begin{bmatrix} 0.600 & 4.000 \\ 0.920 & 1.220 \\ 2.280 & 1.568 \end{bmatrix}$$

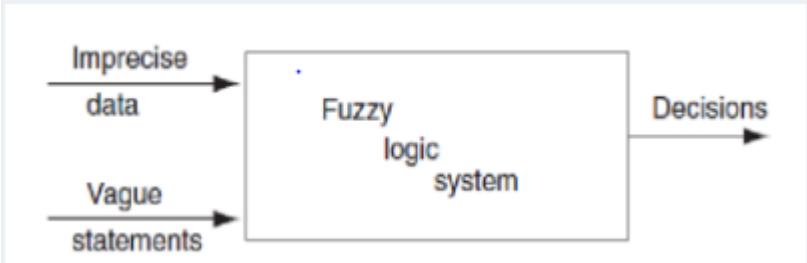
PRACTICAL: 9

AIM: To understand the concept of Fuzzy Logic, Fuzzy Sets and Fuzzification.

Theory:

Introduction to Fuzzy Logic:

Fuzzy Logic is a form of multi-valued logic to deal with reasoning that is approximate rather than precise. This is in contradiction with **crisp logic** that deals with precise values. Also, binary sets have binary or Boolean logic (either 0 or 1), which finds a solution to a particular set of problems. Fuzzy logic variables may have a truth value that ranges between 0 and 1 and is not constrained to the two truth values of classic propositional logic.
Also, as linguistic variables are used in fuzzy logic, these degrees have to be managed by specific type of functions.
Fuzzy logic is a mathematical tool for dealing with uncertainty. It provides a technique to handle imprecision and information granularity. The fuzzy theory provides a mechanism for representing linguistic constructs such as **high, low, tall, short, many**. All these terms are called as **linguistic variables** which represent the uncertainty in the system. In general, fuzzy logic provides an inference structure that enables appropriate human reasoning capabilities. On the contrary, the traditional binary set theory describes crisp events, that is, events that either do or do not occur. It uses probability theory to explain if an event occurs, measuring the chance with which a given event is expected to occur. The theory of fuzzy logic is based upon the notion of relative graded membership and so are the functions of cognitive processes. The utility of fuzzy sets lies in their ability to model uncertain or ambiguous data and to provide suitable decisions as in **Fig1**.



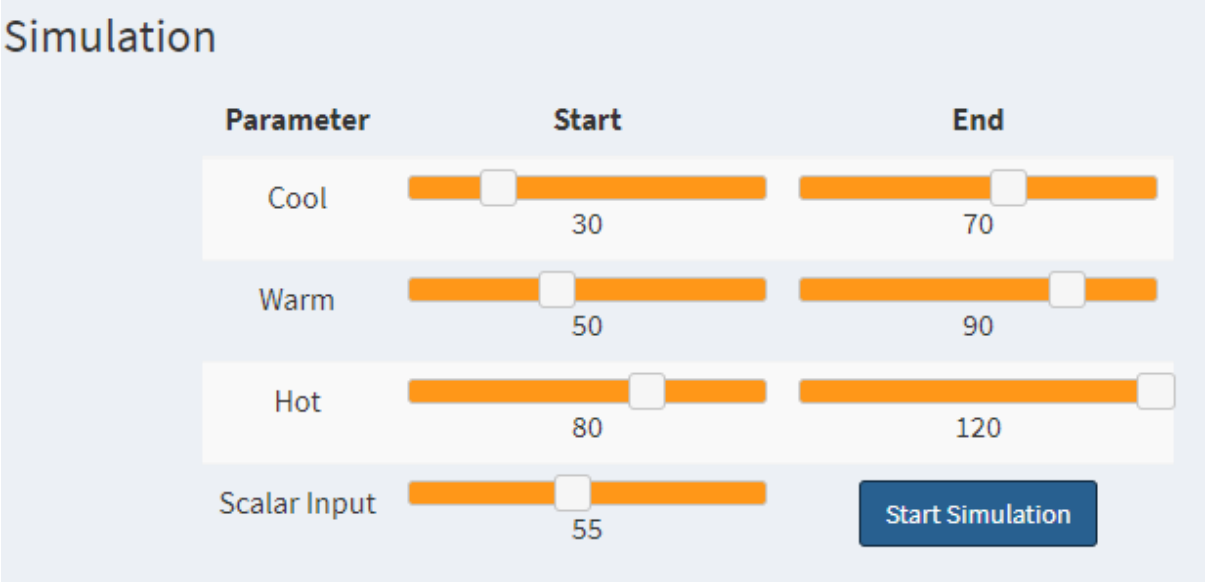
Fuzzy Set:

Fuzzy Sets are sets whose elements have degrees of membership. For example, a **classic set** can be written as { 1, 2, 3, 4 } whereas a ,**Fuzzy Set** can be written as { (1,0.4), (2,0.7), (3,0.1), (4,0.2)} wherein every pair **(X,Y), X represents the value of the element whereas Y represents the degree of membership of the element in the set.**

Fuzzification:

Fuzzification is the process of changing a real scalar value into a fuzzy value. This is done by the help of fuzzifiers (membership functions).
A **membership function (MF)** is a curve that defines how each point in the input space is mapped to a membership value (or degree of membership) between 0 and 1.

OUTPUT:


Simulation
Parameter
Start
End
Cool
30
70
Warm
50
90
Hot
80
120
Scalar Input
55
Start Simulation


Degree Of Truth
1
0
10
20
30
40
50
60
70
80
90
100
110
120
Temperature

Membership in Cool set

(Cool_End - Scalar_Value) × 200 ÷ (Cool_End - Cool_Start)
= (70-55) × 200 ÷ (70-30)
= 15 × 200 ÷ 40
= 75

Membership in Warm set

(Scalar_Value - Warm_Start) × 200 ÷ (Warm_End - Warm_Start)
= (55-50) × 200 ÷ (90-50)
= 5 × 200 ÷ 40
= 25

The temperature is 75% Cool and 25% Warm

PRACTICAL: 10

AIM: To understand the concept of Defuzzification.

Theory:

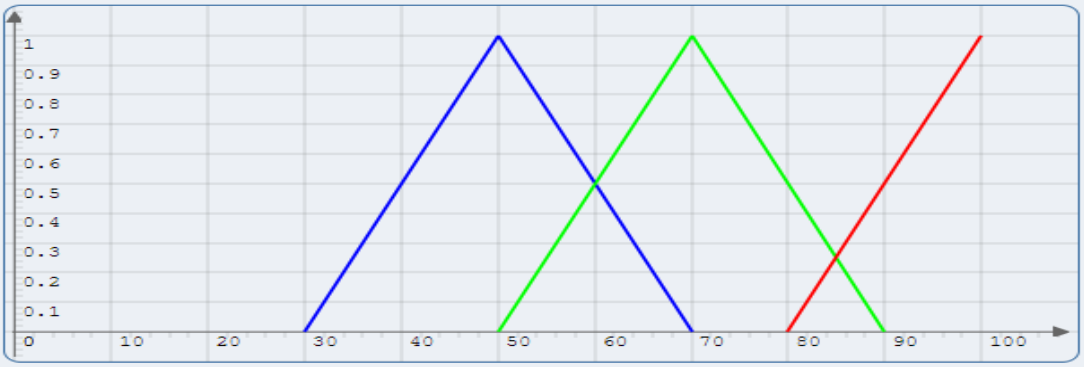
Defuzzification:

Defuzzification is the process of producing a quantifiable result in Crisp logic, given fuzzy sets and corresponding membership degrees. It is the process that maps a fuzzy set to a crisp set. It is typically needed in fuzzy control systems. These will have a number of rules that transform a number of variables into a fuzzy result, that is, the result is described in terms of membership in fuzzy sets. Defuzzification is the conversion of a fuzzy quantity to a precise quantity, just as fuzzification is the conversion of a precise quantity to a fuzzy quantity.

Defuzzification Methods:

1. **Max-Membership Principle**
   This method is also known as height method and is limited to peak output functions. This method is given by the algebraic expression:
   $\boldsymbol{\mu}(z^*) >= \boldsymbol{\mu}(z)$ for all $z \in Z$.
2. **Centroid Method**
   This method is also known as center of mass, center of area or center of gravity. It is the most commonly used defuzzification method. The defuzzified output z* is given by
   $z^* = \int \mu(z).zdz / \int \mu(z)dz$
3. **Weighted Average Method**
   This method is valid for symmetrical output membership functions only. Each membership function is weighted by its maximum membership value. The output in the case is given by
   $z^* = \sum \mu(z').z' / \sum \mu(z')$;
   where z' is the maximum value of the membership function.
4. **Mean-Max Membership**
   This method is also knows as middle of the maxima. This is closely related to the max-membership method, except that the locations of the maximum membership can be nonunique. The output here is given by
   $z^* = \sum z' / n$;
   where z' is the maximum value of the membership function
5. **Center of Sums**
   This method employs the algebraic sum of the individual fuzzy subsets instead of their union. The calculations here are very fast, but the main drawback is that the intersecting areas are added twice. The defuzzified value z* is given by
   $z^* = \int z^* \sum \mu(z).zdz / \int \sum \mu(z)dz$
6. **Center of Largest Area**
   This method can be adopted whent the output of at least two convex fuzzy subsets which are not overlapping. The output in this case is baised towwards a side of one membership function. When output fuzzy st has at least two convex regions, then the center of gravity of the convex fuzzy subregion having the largest are is used to obtain the defuzzified value z*. The value is given by
   $z^* = \int \mu c(z).zdz / \int \sum \mu c(z)dz$

OUTPUT:



Parameter	Start	End
Cool	30	70
Warm	50	90
Hot	80	100

Start Simulation

Method 1 | Method 2 | Method 3 | Method 4 | Method 5 | Method 6

Max-Membership Method

This method is also known as height method and is limited to peak output functions. This method is given by the algebraic expression $\mu(z^*) >= \mu(z)$ for all $z \in Z$.

Here we have to consider the maximum value of the set having the maximum membership value.

MAX {1, 1, 1}
Since all three sets have equal maximum membership value, there are 3 possible Defuzzified Values.
MAX {(Cool_Start + Cool_End) ÷ 2 , (Warm_Start + Warm_End) ÷ 2 , (Hot_End)}
MAX {(30 + 70) ÷ 2 , (50 + 90) ÷ 2 , 100}
MAX {50 , 70 , 100}

The Defuzzified value is 50 or 70 or 100.

Method 1 | **Method 2** | Method 3 | Method 4 | Method 5 | Method 6

Centroid Method

This method is also known as center of mass, center of area or center of gravity . It is the most commonly used defuzzification method. Basically it is the centroid of the area under the graph plotted by the intersection of all the sets. The defuzzified output z* is given by
$z^* = \int \mu(z).zdz / \int \mu(z)dz$

DATA:

	Start	End
Cool	30	70
Warm	50	90

SOLUTION:

Total Area = [(cool_end + cool_start) ÷ 2 + (warm_end+warm_start) ÷ 2 + (hot_end+hot_start) ÷ 2] ÷ (1 + 1 + 1)
Total Area = [(70 + 30) ÷ 2 + (90 + 50) ÷ 2 + (100 + 80) ÷ 2] ÷ 3
Total Area: 70.00
Overlapping Area: 5.00
Ans = Total Area - Overlapping Area
Ans = 70.00 - 5.00
Ans = 65.00
Overlapping Area: 1.25
Ans = Total Area - Overlapping Area
Ans = 65.00 - 1.25
Ans = 63.75

The Defuzzified value is : 63.75

Method 1 | Method 2 | **Method 3** | Method 4 | Method 5 | Method 6

Weighted Average Method

This method is valid for symmetrical output membership functions only. Each membership function is weighted by its maximum membership value. The output in the case is given by
$z^* = \sum \mu(z').z' / \sum \mu(z')$;
where z' is the maximum value of the membership function.

As the membership function for hot is not symmetrical we are ignoring it in this method.

DATA:

	Start	End
Cool	30	70
Warm	50	90

SOLUTION:

a = (cool_start+cool_end) ÷ 2
a = (30+70) ÷ 2
a = 50
b = (warm_start+warm_end) ÷ 2
b = (50+90) ÷ 2
b = 70
value = (a+b)÷ 2
value = (50+70) ÷ 2
value = 60.00

The Defuzzified value is : 60.00

Method 1 | Method 2 | Method 3 | **Method 4** | Method 5 | Method 6

Mean-Max Membership

This method is also knows as middle of the maxima. This is closely related to the max-membership method, except that the locations of the maximum membership can be nonunique. The output here is given by

$z^* = \sum z' / n$;

where z' is the maximum value of the membership function

DATA:

	Start	End
Cool	30	70
Warm	50	90
Hot	80	100

SOLUTION:

a = (cool_start+cool_end) ÷ 2
a = (30+70) ÷ 2
a = 50
b = (warm_start+warm_end) ÷ 2
b = (50+90) ÷ 2
b = 70
c = (hot_start+hot_end) ÷ 2
c = (80+100) ÷ 2
c = 90
value = (a+b+c) ÷ 3
value = (50+70+90) ÷ 2
value = 70.00

The Defuzzified value is : 70.00

Method 1 | Method 2 | Method 3 | Method 4 | **Method 5** | Method 6

Center of Sums

This method employs the algebraic sum of the individual fuzzy subsets instead of their union. The calculations here are very fast, but the main drawback is that the intersecting areas are added twice. The defuzzified value z^* is given by

$z^* = \int z^* \sum \mu(z).zdz / \int \sum \mu(z)dz$

DATA:

	Start	End
Cool	30	70
Warm	50	90

SOLUTION:

a = (cool_end - cool_start) * (cool_end + cool_start) ÷ 2
a = (70 - 30) * (70+30) ÷ 2
a = 2000
b = (warm_end-warm_start) × (warm_end+warm_start) ÷ 2
b = (90-50) × (90+50) ÷ 2
b = 2800
value = (a+b) ÷ [(cool_end-cool_start)+(warm_end-warm_start)]
value = (2000+2800) ÷ [(70-30)+(90-50)]
value = 60.00

The Defuzzified value is : 60.00

Method 1 | Method 2 | Method 3 | Method 4 | Method 5 | **Method 6**

Center of Largest Area

This method can be adopted whent the output of at least two convex fuzzy subsets which are not overlapping. The output in this case is baised towwards a side of one membership function. When output fuzzy st has at least two convex regions, then the center of gravity of the convex fuzzy subregion having the largest are is used to obtain the defuzzified value z*. The value is given by

$z^* = \int \mu c(z).zdz / \int \sum \mu c(z)dz$

DATA:

	Start	End
Cool	30	70
Warm	50	90
Hot	80	100

SOLUTION:

Ans = (warm_end + warm_start) ÷ 2
Ans = (90 + 50) ÷ 2
Ans = 70.00

The Defuzzified value is : 70.00