Anjuman-I-Islam's

AKBAR PEERBHOY COLLEGE OF COMMERCE AND ECONOMICS
M.S Road Do Taki, Mumbai- 400008

Soft Computing Techniques

Submitted by

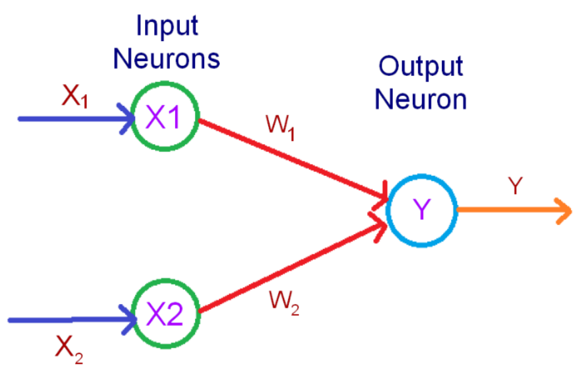
____

Guided by – Prof. Ansari Mohammed Shahid

University of Mumbai
MSC IT (Information Technology) – Semester 1
Academic Year 2024-2025

INDEX

Practical 1-A
Design a simple linear neural network model



The above diagram represents a simple neural network, having one input and one output layer; there are two neurons at the input layer and one at the output.
The input and the output layers are interconnected by weights, we are not considering the bias (b = 0)
The inputs are x_1 and x_2 to the neurons X1 and X2 respectively
The neuron X1 is connected by weight w_1 and X2 by weight w_2 to the output neuron respectively.
The threshold value for activation is 0 as represented by the activation function below

The net input to the output is

$$y_{in} = x_1 w_1 + x_2 w_2$$

In the present case we use the following activation for the output neuron

$$Y_{out} = \begin{cases} 0 \text{ if } y_{in} <= 0 \\ 1 \text{ if } y_{in} > 0 \end{cases}$$

The following Python program shows the implementation of the simple neural network

```
print("Enter Value of X1 =")
x1 = int(input())

print("Enter Value of X2 =")
x2 = int(input())

print("Enter Value of W1 =")
w1 = int(input())

print("Enter Value of W2 =")
w2 = int(input())
```

```
yin = x1*w1 + x2*w2
print("Net Input to the Output Neuron=",yin)

if yin <= 0:
        yout = 0
else :
        yout = 1

print("The Output of given neural network = ",yout)
```

The output is

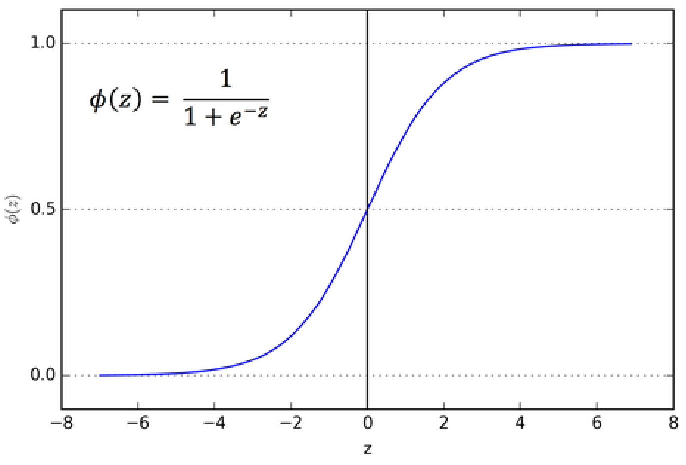
```
Enter Value of X1 =
-1
Enter Value of X2 =
11
Enter Value of W1 =
1
Enter Value of W2 =
1
Net Input to the Output Neuron= 10
The Output of given neural network = 1
```

Practical 1-B
Neural net using both binary and bipolar sigmoidal function.

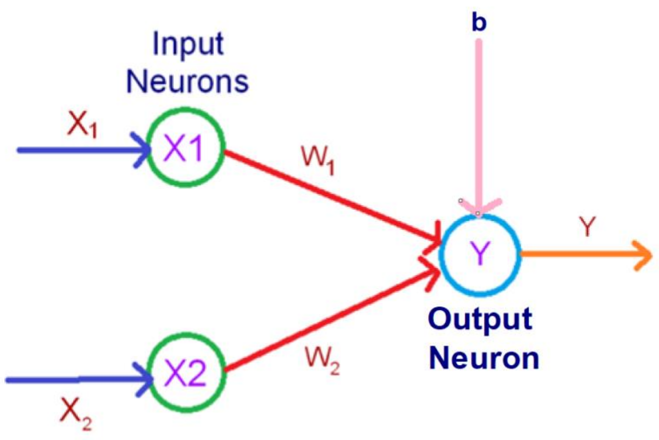
Part 1: Using Binary sigmoidal function

The function takes any real value as input and outputs values in the range 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0.

The Sigmoid Function curve looks like a S-shape.



Consider the following Neural Network



The Python code for the given case is as follows

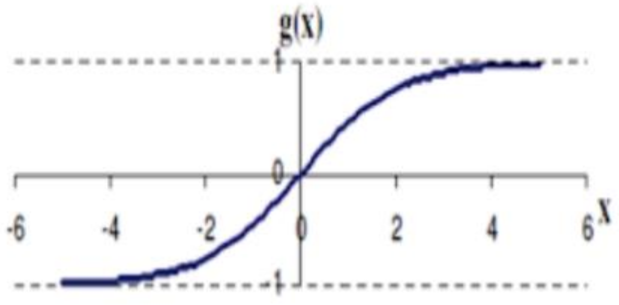
```
import numpy as np

print("Enter Value of X1 =");
x1 = int(input());

print("Enter Value of X2 =");
x2 = int(input());

print("Enter Value of W1 =");
w1 = int(input());

print("Enter Value of W2 =");
w2 = int(input());

b = 1;

yin = b + (x1*w1 + x2*w2);
print("Net Input to the Output Neuron=",yin);

output = 1/(1 + np.exp(-yin));
print("Output=", round(output, 4));
```

The output is as follows

```
Enter Value of X1 =
 -1
Enter Value of X2 =
2
Enter Value of W1 =
-2
Enter Value of W2 =
-1
Net Input to the Output Neuron= 1
Output= 0.7311
```

Irrespective of the input, the output is always between 0 and 1

Part 2: Using Bipolar sigmoidal function
The Bipolar sigmoidal function is represented in the following way



Mathematically it is given by

$$y = f(y_{in}) = \frac{2}{1 + e^{-y_{in}}} - 1$$

The python code for bipolar sigmoidal function is given by

```
import numpy as np
print("Enter Value of X1 =")
x1 = int(input())
print("Enter Value of X2 =")
x2 = int(input())
print("Enter Value of W1 =")
w1 = int(input())
print("Enter Value of W2 =")
w2 = int(input())
b = 1
yin = b + (x1*w1 + x2*w2)
print("Net Input to the Output Neuron=", yin)
```

The output is as follows

```
Enter Value of X1 =
1
Enter Value of X2 =
1
Enter Value of W1 =
1
Enter Value of W2 =
1
Net Input to the Output Neuron= 3
Output= 0.9051
```

Irrespective of the input, the output is always between -1 and 1

Generate AND/NOT function using McCulloch-Pitts neural net.

The McCulloch–Pitt neural network is considered to be the first neural network. The neurons are connected by directed weighted paths. McCulloch–Pitt neuron allows binary activation (ON or OFF), i.e., it either fires with an activation 1 or does not fire with an activation of 0.
If w > 0, then the connected path is said to be excitatory
If w < 0 ,then the connected path is said to be inhibitory.
Excitatory connections have positive weights and inhibitory connections have negative weights. Each neuron has a fixed threshold for firing. That is, if the net input to the neuron is greater than the threshold, it fires.
In the present case we implement AND-NOT function using McCulloch-Pitts Neural Network.
Binary inputs are taken, and the weights are initialized to 0
We have the following training sets and the target to implement the AND-NOT function

Training Sets		Target
X1	X2	t
0	0	0
0	1	0
1	0	1
1	1	0

The Python code is

```
w1 = 0
w2 = 0
print("For the ", 4, " inputs calculate the net input using yin = x1w1 + x2w2 ")

x1 = [0,0,1,1]
x2 = [0,1,0,1]
y = [0,0,1,0]
b = 0
print("Training Sets");
print("x1 = ",x1)
print("x2 = ",x2)
print("Target(y) :",y);

for i in range(0,4):
        w1 = w1 + x1[i] * y[i];
        w2 = w2 + x2[i] * y[i];
        b = b + y[i];
print("The updated weights are: ");
        print("W1new=",w1);
        print("w2new=",w2);
        print("bnew=",b);

print("The Final Weights are :")
print("w1new=",w1);
print("w2new=",w2);
print("bnew=",b);
```

The output of the programs is

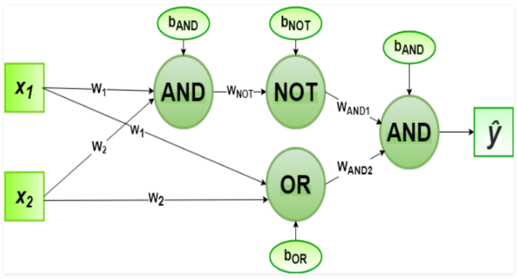
```
For the 4 inputs calculate the net input using yin = x1w1 + x2w2
Training Sets
x1 =   [0, 0, 1, 1]
x2 =   [0, 1, 0, 1]
Target(y) : [0, 0, 1, 0]
The updated weights are:
W1new= 0
w2new= 0
bnew= 0
The updated weights are:
W1new= 0
w2new= 0
bnew= 0
The updated weights are:
W1new= 1
w2new= 0
bnew= 1
The updated weights are:
W1new= 1
w2new= 0
bnew= 1
The Final Weights are :
w1new= 1
w2new= 0
bnew= 1
```

Generate XOR function using McCulloch-Pitts neural net.

We need to implement XOR function using McCulloch-Pitts Neural Network.
Binary inputs are taken, and the weights are initialized to suitable values.
We have the following training sets and the target to implement the EX-OR function

Training Sets		Target
X1	X2	t
0	0	0
0	1	1
1	0	1
1	1	0

As XOR function is not variable separable, we need some different strategy to implement this function.
Consider the diagram below



The weights are initialized as
$W_1 = 1$, $W_2 = 1$,
$W_{AND1} = 1$, $W_{AND2} = 1$,
$W_{NOT} = -1$,
$b_{AND} = -1.5$, $b_{OR} = -0.5$,
$b_{NOT} = 0.5$

The Python code for the given program is

```
# importing Python library
import numpy as np

# define Unit Step Function
def unitStep(v):
    if v >= 0:
        return 1
    else:
        return 0

# design Perceptron Model
def perceptronModel(x, w, b):
    v = np.dot(w, x) + b
    y = unitStep(v)
    return y

# NOT Logic Function
# wNOT = -1, bNOT = 0.5
def NOT_logicFunction(x):
    wNOT = -1
    bNOT = 0.5
    return perceptronModel(x, wNOT, bNOT)

# AND Logic Function
# here w1 = wAND1 = 1,
# w2 = wAND2 = 1, bAND = -1.5
def AND_logicFunction(x):
    w = np.array([1, 1])
    bAND = -1.5
    return perceptronModel(x, w, bAND)

# OR Logic Function
# w1 = 1, w2 = 1, bOR = -0.5
def OR_logicFunction(x):
    w = np.array([1, 1])
    bOR = -0.5
    return perceptronModel(x, w, bOR)

# XOR Logic Function
# with AND, OR and NOT
# function calls in sequence
def XOR_logicFunction(x):
    y1 = AND_logicFunction(x)
    y2 = OR_logicFunction(x)
    y3 = NOT_logicFunction(y1)
    final_x = np.array([y2, y3])
    finalOutput = AND_logicFunction(final_x)
    return finalOutput

# testing the Perceptron Model
test1 = np.array([0, 0])
test2 = np.array([0, 1])
test3 = np.array([1, 0])
test4 = np.array([1, 1])

print("XOR({}, {}) = {}".format(0, 0, XOR_logicFunction(test1)))
print("XOR({}, {}) = {}".format(0, 1, XOR_logicFunction(test2)))
print("XOR({}, {}) = {}".format(1, 0, XOR_logicFunction(test3)))
print("XOR({}, {}) = {}".format(1, 1, XOR_logicFunction(test4)))
```

We get the following output

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

Write a program to implement Hebb's rule

Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network. It is used for pattern classification.
The Hebbian Learning Rule is a learning rule that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. The rule builds on Hebb's 1949 learning rule which states that the connections between two neurons might be strengthened if the neurons fire simultaneously. The Hebbian Rule works well as long as all the input patterns are orthogonal or uncorrelated. The requirement of orthogonality places serious limitations on the Hebbian Learning Rule.
For the present case we consider the example of OR function, using Bipolar input we have the following relations between the training sets and the target

Training Sets		Target
X1	X2	t
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

The Python code for the given case is

```
w1 = 0
w2 = 0
b = 0

x1 = [-1, -1, 1, 1]
x2 = [-1, 1, -1, 1]
y = [-1, 1, 1, 1]

def train_perceptron(x1, x2, y):
    global w1, w2, b
    for i in range(len(x1)):
        w1 += x1[i] * y[i]
        w2 += x2[i] * y[i]
        b += y[i]
        print("Epoch {}: w1_new={}, w2_new={}, b_new={}".format(i+1, w1, w2, b))

# Training with OR examples
print("Training with OR examples:")
train_perceptron(x1, x2, y)

# Display the final weights and bias
print("\nThe Final Weights are:")
print("w1_new =", w1)
print("w2_new =", w2)
print("b_new =", b)
```

The output is as follows

```
Training with OR examples:
Epoch 1: w1_new=1, w2_new=1, b_new=-1
Epoch 2: w1_new=0, w2_new=2, b_new=0
Epoch 3: w1_new=1, w2_new=1, b_new=1
Epoch 4: w1_new=2, w2_new=2, b_new=2

The Final Weights are:
w1_new = 2
w2_new = 2
b_new = 2
```

Implementing the Delta rule

The Delta rule in machine learning and neural network environments is a specific type of Backpropagation that helps to refine connectionist ML/AI networks, making connections between inputs and outputs with layers of artificial neurons.

The Delta rule is also known as the Delta learning rule, in general, Backpropagation has to do with recalculating input weights for artificial neurons using a gradient method. Delta learning does this using the difference between a target activation and an actual obtained activation. Using a linear activation function, network connections are adjusted.

Another way to explain the Delta rule is that it uses an error function to perform gradient descent learning.

A tutorial on the Delta rule explains that essentially in comparing an actual output with a targeted output, the technology tries to find a match. If there is not a match, the program makes changes. The actual implementation of the Delta rule is going to vary according to the network and its composition, but by employing a linear activation function, the Delta rule can be useful in refining some types of neural network systems with particular flavour of Backpropagation.

For the present case we take the example of OR function using Bipolar inputs

Training Sets		Target
X1	X2	t
-1	-1	-1
-1	1	1
1	-1	1
1	1	1

The Python code is as follows

```
import numpy as np

# Input data for OR function
X = np.array([
        [0, 0],
        [0, 1],
        [1, 0],
        [1, 1]
])

# Target values for OR function
Y = np.array([0, 1, 1, 1])

# Initialize weights and bias
weights = np.random.rand(2)    # Initialize with random values
bias = np.random.rand(1)

# Learning rate
learning_rate = 0.1

# Number of epochs
```

```
epochs = 10

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

def delta_rule_train(X, Y, weights, bias, learning_rate, epochs):
    for epoch in range(epochs):
        for i in range(len(X)):
            # Forward pass
            output = sigmoid(np.dot(X[i], weights) + bias)

            # Compute error
            error = Y[i] - output

            # Update weights and bias using the Delta rule
            weights += learning_rate * error * X[i]
            bias += learning_rate * error

        # Display the error for every 100 epochs
        if epoch % 100 == 0:
            total_error = np.mean(np.square(Y - sigmoid(np.dot(X, weights) + bias)))
            print("Epoch {}: Total Error: {}".format(epoch, total_error))

    return weights, bias

# Train the perceptron using the Delta rule
trained_weights, trained_bias = delta_rule_train(X, Y, weights, bias, learning_rate, epochs)

# Display the final weights and bias rounded to two decimal places
print("\nThe Final Weights are:")
print("w1_new =", round(trained_weights[0], 2))
print("w2_new =", round(trained_weights[1], 2))
print("b_new =", round(trained_bias[0], 2))
```

We get the following output

```
Epoch 0: Total Error: 0.15009771759263787

The Final Weights are:
w1_new = 0.97
w2_new = 0.67
b_new = 0.43
```

Implementing Back Propagation Algorithm

Backpropagation, short for "backward propagation of errors," is a supervised learning algorithm used for training artificial neural networks. It is a widely used optimization algorithm that helps adjust the weights of the neural network to minimize the difference between the predicted output and the actual output.

Explanation of backpropagation:

1. Forward Pass:
 - The input data is fed forward through the network, layer by layer, to generate the predicted output.
 - Each neuron in the network applies an activation function to its input, producing an output that becomes the input for the next layer.

2. Compute Loss:
 - The predicted output is compared to the actual target output, and the error (or loss) is calculated. This measures how far off the network's predictions are from the true values.

3. Backward Pass (Backpropagation):
 - The algorithm then works backward through the network to compute the gradients of the loss with respect to the weights.
 - The gradients represent how much the loss would increase or decrease if the weights were adjusted.

4. Weight Update:
 - The weights are then updated using an optimization algorithm, such as gradient descent, to minimize the loss. The learning rate determines the size of the steps taken during this update.

5. Iteration:
 - Steps 1-4 are repeated for multiple iterations (epochs) until the network has learned the underlying patterns in the training data.

The key idea behind backpropagation is to use the chain rule from calculus to compute the gradient of the loss with respect to the weights. This allows the network to adjust its weights in a way that reduces the error in predicting the target values.

The backpropagation algorithm makes training deep neural networks feasible by efficiently computing the gradients for all the weights in the network. It's a crucial component in the training of neural networks for a wide range of tasks, including image classification, natural language processing, and many others.

The Python code for the given case is

```
import numpy as np

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

def sigmoid_derivative(x):
    return x    (1 - x)

def initialize_weights(input_size, hidden_size, output_size):
    np.random.seed(42)
    weights_input_hidden = np.random.rand(input_size, hidden_size)
    weights_hidden_output = np.random.rand(hidden_size, output_size)
    return weights_input_hidden, weights_hidden_output

def forward_pass(inputs, weights_input_hidden, weights_hidden_output):
    hidden_layer_input = np.dot(inputs, weights_input_hidden)
    hidden_layer_output = sigmoid(hidden_layer_input)

    output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
    output_layer_output = sigmoid(output_layer_input)

    return hidden_layer_output, output_layer_output

def backward_pass(inputs, targets, hidden_layer_output, output_layer_output,
weights_hidden_output):
    output_error = targets - output_layer_output
    output_delta = output_error    sigmoid_derivative(output_layer_output)

    hidden_layer_error = output_delta.dot(weights_hidden_output.T)
    hidden_layer_delta = hidden_layer_error    sigmoid_derivative(hidden_layer_output)

    return output_delta, hidden_layer_delta

def update_weights(inputs, hidden_layer_output, output_delta, hidden_layer_delta,
weights_input_hidden, weights_hidden_output, learning_rate=0.1):
    weights_hidden_output += hidden_layer_output.T.dot(output_delta)    learning_rate
    weights_input_hidden += inputs.T.dot(hidden_layer_delta)    learning_rate

def train_neural_network(inputs, targets, epochs=10000):
    input_size = inputs.shape[1]
    hidden_size = 4    # You can adjust the number of hidden layer neurons
    output_size = 1

    weights_input_hidden, weights_hidden_output = initialize_weights(input_size, hidden_size,
output_size)

    for epoch in range(epochs):
        # Forward pass
        hidden_layer_output, output_layer_output = forward_pass(inputs, weights_input_hidden,
weights_hidden_output)
```

```
        # Backward pass
        output_delta, hidden_layer_delta = backward_pass(inputs, targets, hidden_layer_output,
output_layer_output, weights_hidden_output)

        # Update weights
        update_weights(inputs, hidden_layer_output, output_delta, hidden_layer_delta,
weights_input_hidden, weights_hidden_output)

    return weights_input_hidden, weights_hidden_output

def predict(inputs, weights_input_hidden, weights_hidden_output):
    _, output = forward_pass(inputs, weights_input_hidden, weights_hidden_output)
    return np.round(output)

# Example usage:
# Training data for the OR function
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [1]])

# Train the neural network
trained_weights_input_hidden, trained_weights_hidden_output = train_neural_network(inputs,
targets)

# Test the trained neural network
for input_pattern, target in zip(inputs, targets):
    prediction = predict(input_pattern, trained_weights_input_hidden,
trained_weights_hidden_output)
    print(f"{input_pattern} -> {prediction} (target: {target})")
```

We get the following output

```
[0 0] -> [0.] (target: [0])
[0 1] -> [1.] (target: [1])
[1 0] -> [1.] (target: [1])
[1 1] -> [1.] (target: [1])
```

Implementing Error Back Propagation Algorithm

"Error backpropagation" is another term for the backpropagation algorithm in the context of neural networks. It refers to the process of propagating the error backward through the network during the training phase.

In the context of neural networks, the term "error" typically refers to the difference between the predicted output of the network and the actual target output. The goal of the training process is to minimize this error, and the backpropagation algorithm is a key tool for achieving that.

Error backpropagation works as follows

1. Forward Pass:
   - Input data is fed forward through the network to generate the predicted output.
   - Each layer applies an activation function to its input, producing an output.

2. Compute Error:
   - The predicted output is compared to the actual target output, and the error (loss) is calculated.

3. Backward Pass (Backpropagation):
   - The algorithm works backward through the network to compute the gradients of the loss with respect to the weights.
   - It uses the chain rule of calculus to efficiently calculate how much each weight contributed to the error.

4. Weight Update:
   - The weights are updated using an optimization algorithm (e.g., gradient descent) based on the calculated gradients. This step aims to reduce the error by adjusting the weights in a direction that minimizes the loss.

5. Iteration:
   - Steps 1-4 are repeated for multiple iterations (epochs) until the network has learned to make accurate predictions.

The term "backpropagation" emphasizes the backward flow of information during the training process. The error calculated at the output layer is propagated backward through the network, and the weights are updated accordingly. This iterative process allows the neural network to learn from its mistakes and improve its performance over time.

Python Code

```
import numpy as np

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

def sigmoid_derivative(x):
	return x * (1 - x)

def initialize_weights(input_size, hidden_size, output_size):
	np.random.seed(42)
	weights_input_hidden = np.random.rand(input_size, hidden_size)
	weights_hidden_output = np.random.rand(hidden_size, output_size)
	return weights_input_hidden, weights_hidden_output

def forward_pass(inputs, weights_input_hidden, weights_hidden_output):
	hidden_layer_input = np.dot(inputs, weights_input_hidden)
	hidden_layer_output = sigmoid(hidden_layer_input)

	output_layer_input = np.dot(hidden_layer_output, weights_hidden_output)
	output_layer_output = sigmoid(output_layer_input)

	return hidden_layer_output, output_layer_output

def calculate_error(targets, output_layer_output):
	return targets - output_layer_output

def backward_pass(inputs, hidden_layer_output, output_layer_output, error,
weights_hidden_output):
	output_delta = error * sigmoid_derivative(output_layer_output)

	hidden_layer_error = output_delta.dot(weights_hidden_output.T)
	hidden_layer_delta = hidden_layer_error * sigmoid_derivative(hidden_layer_output)

	return output_delta, hidden_layer_delta

def update_weights(inputs, hidden_layer_output, output_delta, hidden_layer_delta,
weights_input_hidden, weights_hidden_output, learning_rate=0.1):
	weights_hidden_output += hidden_layer_output.T.dot(output_delta) * learning_rate
	weights_input_hidden += inputs.T.dot(hidden_layer_delta) * learning_rate

def train_neural_network(inputs, targets, epochs=10000):
	input_size = inputs.shape[1]
	hidden_size = 4    # You can adjust the number of hidden layer neurons
	output_size = 1

	weights_input_hidden, weights_hidden_output = initialize_weights(input_size,
hidden_size, output_size)

	for epoch in range(epochs):
		# Forward pass
		hidden_layer_output, output_layer_output = forward_pass(inputs,
weights_input_hidden, weights_hidden_output)
```

```
            # Calculate error
            error = calculate_error(targets, output_layer_output)

            # Backward pass
            output_delta, hidden_layer_delta = backward_pass(inputs, hidden_layer_output,
output_layer_output, error, weights_hidden_output)

            # Update weights
            update_weights(inputs, hidden_layer_output, output_delta, hidden_layer_delta,
weights_input_hidden, weights_hidden_output)

      return weights_input_hidden, weights_hidden_output

def predict(inputs, weights_input_hidden, weights_hidden_output):
      _, output = forward_pass(inputs, weights_input_hidden, weights_hidden_output)
      return np.round(output)

# Example usage:
# Training data for the OR function
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [1]])

# Train the neural network
trained_weights_input_hidden, trained_weights_hidden_output = train_neural_network(inputs,
targets)

# Test the trained neural network
for input_pattern, target in zip(inputs, targets):
      prediction = predict(input_pattern, trained_weights_input_hidden,
trained_weights_hidden_output)
      print(f"{input_pattern} -> {prediction} (target: {target})")
```

We get the following output

```
[0 0] -> [0.] (target: [0])
[0 1] -> [1.] (target: [1])
[1 0] -> [1.] (target: [1])
[1 1] -> [1.] (target: [1])
```

Implementing Hopfield Network

A Hopfield network is a type of recurrent artificial neural network, primarily used for associative memory and pattern recognition. It was introduced by John Hopfield in 1982. Hopfield networks are single-layer networks with symmetric connections, meaning that the weights are the same for connections going from one neuron to another and from the second neuron back to the first.

Key characteristics of Hopfield networks include:

1. Recurrence: Hopfield networks are fully connected and recurrent, meaning each neuron is connected to every other neuron, including itself. The recurrent connections allow the network to store and retrieve patterns.

2. Symmetric Weights: The connection weights in a Hopfield network are symmetric. If there is a connection from neuron i to neuron j, there is also a connection from neuron j to neuron i, and the weights are the same in both directions.

3. Binary Activation: Neurons in a Hopfield network typically use binary activation, where the states are either +1 or -1. The network operates in discrete time steps.

4. Energy Function: The network is designed to minimize an energy function during the learning process. The energy function is associated with the patterns the network is trained on.

Hopfield networks have applications in content-addressable memory and pattern recognition. They can be trained to store and retrieve patterns, and given a partial or noisy input, they can reconstruct the closest stored pattern. However, Hopfield networks have limitations, such as being prone to spurious patterns and having limitations in terms of storage capacity.

The update rule for a Hopfield network is typically based on the Hebbian learning rule, which adjusts weights based on the correlation between the activities of connected neurons.

While Hopfield networks have been influential in the development of neural network concepts, more advanced and scalable models, such as feedforward neural networks and recurrent neural networks, are commonly used for modern applications in machine learning and artificial intelligence.

The Python code for the given case is

```
import numpy as np
```

```
class HopfieldNetwork:
    def __init__(self, size):
        self.size = size
        self.weights = np.zeros((size, size))

    def train(self, patterns):
        for pattern in patterns:
            pattern = np.reshape(pattern, (self.size, 1))
            self.weights += np.outer(pattern, pattern)
            np.fill_diagonal(self.weights, 0)

    def predict(self, input_pattern, max_iter=100):
        input_pattern = np.reshape(input_pattern, (self.size, 1))

        for _ in range(max_iter):
            output_pattern = np.sign(np.dot(self.weights, input_pattern))
            if np.array_equal(input_pattern, output_pattern):
                return output_pattern.flatten()
            input_pattern = output_pattern

        raise RuntimeError("Hopfield Network did not converge within the maximum number of
iterations.")

# Example usage:
# Define patterns for training
pattern1 = np.array([1, -1, 1, -1])
pattern2 = np.array([-1, -1, -1, 1])
patterns = [pattern1, pattern2]

# Create a Hopfield Network and train it
hopfield_net = HopfieldNetwork(size=len(pattern1))
hopfield_net.train(patterns)

# Test the Hopfield Network with a noisy input pattern
noisy_pattern = np.array([1, 1, 1, 1])
predicted_pattern = hopfield_net.predict(noisy_pattern)

# Display results
print("Noisy Pattern:", noisy_pattern)
print("Predicted Pattern:", predicted_pattern)
```

We get the following output

```
Noisy Pattern: [1 1 1 1]
Predicted Pattern: [ 1.  0.  1. -1.]
```

Implementing Radial Basis function

A Radial Basis Function (RBF) is a real-valued function whose output depends on the distance between the input and a fixed point in space, often referred to as the center. RBFs are commonly used in various fields, including machine learning, mathematics, and signal processing. In the context of machine learning, RBFs are often used as activation functions in certain types of neural networks or as kernels in support vector machines (SVMs).

In the context of neural networks, a Radial Basis Function Network (RBFN) is a type of artificial neural network that uses radial basis functions as activation functions.

The architecture typically consists of three layers:

1. Input Layer: Neurons in the input layer represent the input features.

2. Hidden Layer with Radial Basis Functions: Neurons in the hidden layer apply radial basis functions to transform the input. Each neuron in the hidden layer is associated with a center, and its output is a function of the distance between the input and the center.

3. Output Layer: Neurons in the output layer combine the outputs of the hidden layer to produce the final network output.

The RBFN learns the centers and widths of the radial basis functions during the training process.

In SVMs, RBFs are commonly used as kernels in the kernelized version of the algorithm. The RBF kernel measures the similarity (or distance) between data points in the input space.

The use of radial basis functions allows these models to capture complex, non-linear relationships in the data.

The Python code for the given case is

```
importnumpy as np
fromscipy.spatial.distance import cdist
```

```
classRadialBasisFunctionNetwork:
def __init__(self, input_size, hidden_size, output_size):
self.input_size = input_size
self.hidden_size = hidden_size
self.output_size = output_size
self.centers = None
self.width = None
self.weights_hidden_output = np.random.rand(hidden_size, output_size)

definitialize_centers(self, inputs):
            # Use k-means clustering to initialize centers
indices = np.random.choice(len(inputs), self.hidden_size, replace=False)
self.centers = inputs[indices]

defcalculate_width(self):
            # Set the width as the maximum distance between centers
distances = cdist(self.centers, self.centers, 'euclidean')
self.width = np.max(distances) / np.sqrt(2 * self.hidden_size)

defradial_basis_function(self, x, c, width):
returnnp.exp(-np.linalg.norm(x - c) / (2 * width**2))

def train(self, inputs, targets, learning_rate=0.01, epochs=1000):
self.initialize_centers(inputs)
self.calculate_width()

for epoch in range(epochs):
for i in range(len(inputs)):
                        # Calculate activations for hidden layer (RBF layer)
hidden_layer_output = np.array([self.radial_basis_function(inputs[i], c, self.width) for c in
self.centers])

                        # Calculate output layer (linear)
output_layer_output = np.dot(hidden_layer_output, self.weights_hidden_output)

                        # Update weights using gradient descent
self.weights_hidden_output += learning_rate * np.outer(hidden_layer_output, targets[i] -
output_layer_output)

def predict(self, inputs):
hidden_layer_output = np.array([self.radial_basis_function(inputs, c, self.width) for c in
self.centers])
output_layer_output = np.dot(hidden_layer_output, self.weights_hidden_output)
returnoutput_layer_output

# Example usage:
# Define input patterns and corresponding targets
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
targets = np.array([[0], [1], [1], [0]])

# Create an RBF Network and train it
```

```
rbf_net = RadialBasisFunctionNetwork(input_size=2, hidden_size=4, output_size=1)
rbf_net.train(inputs, targets)

# Test the RBF Network
forinput_pattern, target in zip(inputs, targets):
predicted_output = rbf_net.predict(input_pattern)
print(f"Input: {input_pattern}, Target: {target}, Predicted Output: {predicted_output}")
```

We get the following output

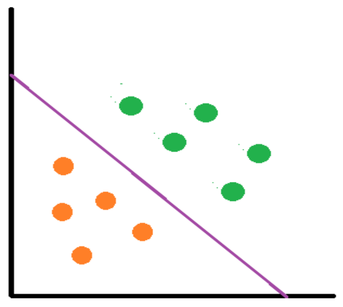
```
Input: [0 0], Target: [0], Predicted Output: [0.0012516]
Input: [0 1], Target: [1], Predicted Output: [0.99879928]
Input: [1 0], Target: [1], Predicted Output: [0.99879736]
Input: [1 1], Target: [0], Predicted Output: [0.00115243]
```

Write a program for Linear separation

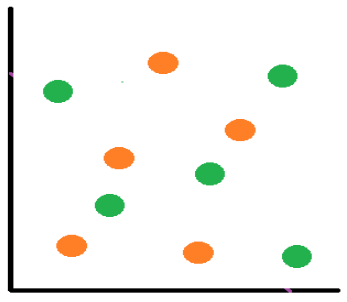
Linear separable means that there is a hyperplane, which splits the input data into two half-spaces such that all points of the first class should be in one half-space and other points of the second class should be in the other half-space.

In 2-D, it means that there is a line, which separates points of one class from points of the other class.

For example**:** In the following image, if orange circles represent points from one class and green circles represent points from the other class, then these points are linearly separable.



In the following image, the two classes represented by orange and green circles cannot be separated by a line



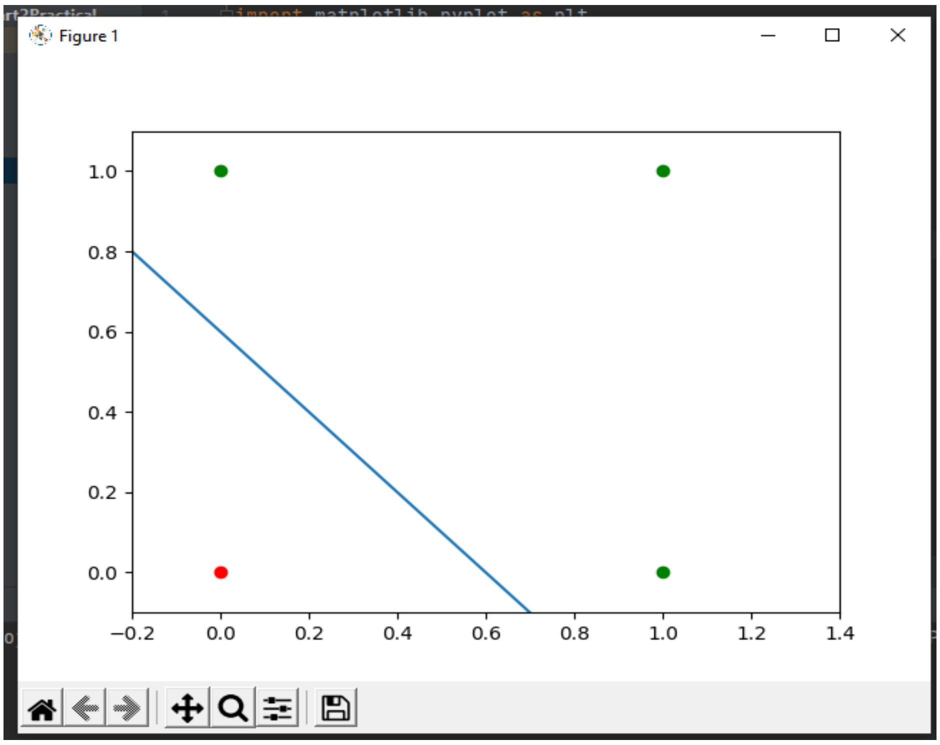
Linear separability is an important concept in neural networks.
In ANN a decision line is drawn to separate positive and negative responses. The decision line may also be called as the decision-making Line or decision-support Line or linear-separable line. The necessity of the linear separability concept was felt to clarify classify the patterns based upon their output responses

The Logic functions AND, OR etc are linearly separable, while XOR and EX-NOR are not linearly separable.

The following program shows the linear separability of OR functions.

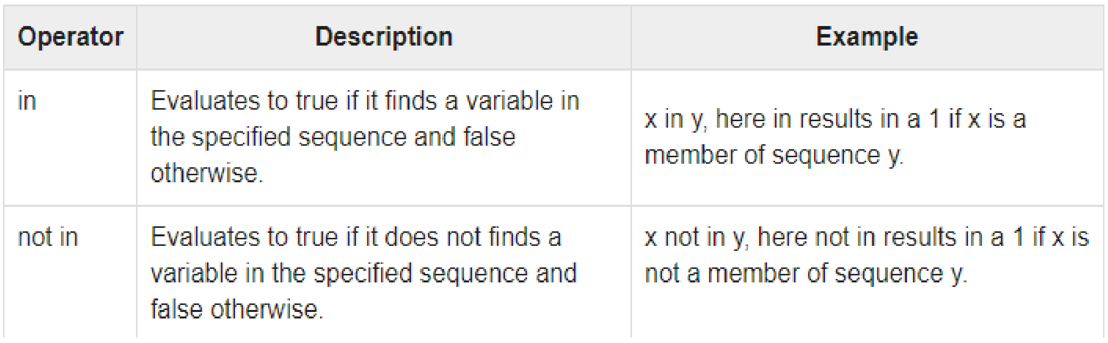
```
import matplotlib.pyplot as plt

import numpy as np


fig, ax = plt.subplots()

xmin, xmax = -0.2, 1.4

X = np.arange(xmin, xmax, 0.1)

ax.scatter(0, 0, color="r")

ax.scatter(0, 1, color="g")

ax.scatter(1, 0, color="g")

ax.scatter(1, 1, color="g")

ax.set_xlim([xmin, xmax])

ax.set_ylim([-0.1, 1.1])

m = -1

ax.plot(X, m * X + 0.6, label="decision boundary")

plt.show()

#plt.plot()
```

We get the following plot



Green dots show output as 1 and red shows output as 0

Membership and Identity Operators is, is not



Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

# Membership Operators (in and not in)

```
# Lists
fruits = ["apple", "banana", "cherry"]

# Check if an element is in the list
if "banana" in fruits:
    print("banana is in the list")

# Check if an element is not in the list
if "orange" not in fruits:
    print("orange is not in the list")

# Strings
text = "Hello, world"

# Check if a substring is in the string
if "world" in text:
    print("The word 'world' is in the string")
```

# Identity Operators (is and is not)

```
# Numbers
x = 10
y = 10

# Check if two variables reference the same object
if x is y:
    print("x and y are the same object")

# Check if two variables do not reference the same object
if x is not "Hello":
    print("x is not 'Hello'")

# Lists
list1 = [1, 2, 3]
list2 = [1, 2, 3]

# Check if two lists are not the same object (even though they have the same contents)
if list1 is not list2:
    print("list1 and list2 are not the same object")
```

Find ratios using fuzzy logic

Ratio

We used the ratio function above to calculate the Levenshtein distance similarity ratio between the two strings (sequences).

Partial Ratio

FuzzyWuzzy also has more powerful functions to help with matching strings in more complex situations. This function allows us to perform substring matching. This works by taking the shortest string and matching it with all substrings that are of the same length.

Token Sort Ratio

FuzzyWuzzy also has token functions that tokenize the strings, change capitals to lowercase, and remove punctuation. This function sorts the strings alphabetically and then joins them together. Then, the fuzz.ratio() is calculated. This can come in handy when the strings we are comparing are the same in spelling but are not in the same order.

Token Set Ratio

This function is similar to the token sort ratio, except it takes out the common tokens before calculating the fuzz ration between the new strings. This function is the most helpful when applied to a set of strings with a significant difference in lengths.

```
# Python code showing all the ratios together,

# make sure you have installed fuzzywuzzy module


from fuzzywuzzy import fuzz

from fuzzywuzzy import process


s1 = "I like softcomputing"

s2 = "I like hardcomputing"

print ("FuzzyWuzzy Ratio: ", fuzz.ratio(s1, s2))

print ("FuzzyWuzzy PartialRatio: ", fuzz.partial_ratio(s1, s2))

print ("FuzzyWuzzy TokenSortRatio: ", fuzz.token_sort_ratio(s1, s2))

print ("FuzzyWuzzy TokenSetRatio: ", fuzz.token_set_ratio(s1, s2))

print ("FuzzyWuzzy WRatio: ", fuzz.WRatio(s1, s2),'\n\n')
```

We get the following output

```
FuzzyWuzzy Ratio:    80
FuzzyWuzzy PartialRatio:    80
FuzzyWuzzy TokenSortRatio:    45
FuzzyWuzzy TokenSetRatio:    80
FuzzyWuzzy WRatio:    80
```