EXERCISE-4(B)

AIM: BUILD AN ARTIFICIAL NEURAL NETWORK BY IMPLEMENTING THE BACK PROPAGATION ALGORITHM AND TEST THE SAME USING APPROPRIATE DATA SETS.

DESCRIPTION:

An **Artificial Neuron Network (neural network)** is a computational model that mimics the way **nerve cells work in the human brain**. ANN algorithm accepts only **numeric and structured data.** ANNs can learn and model **non-linear and complicated interactions**, which is critical since many of the relationships between **inputs and outputs in real life** are non-linear and complex.

Architecture:

There are three layers in the network architecture:

The Input Layer,

The Hidden Layer (more than one), and

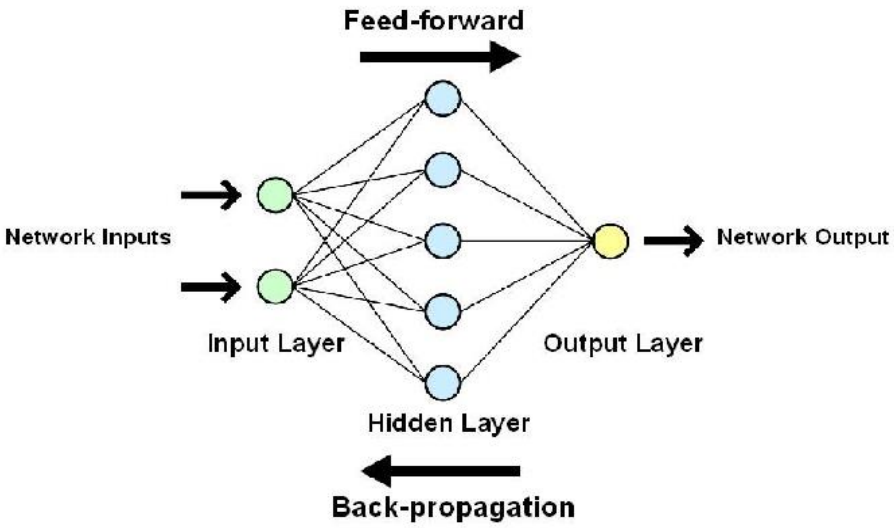
The Output Layer.

Because of the numerous layers are sometimes referred to as the **MLP (Multi-Layer Perceptron).**

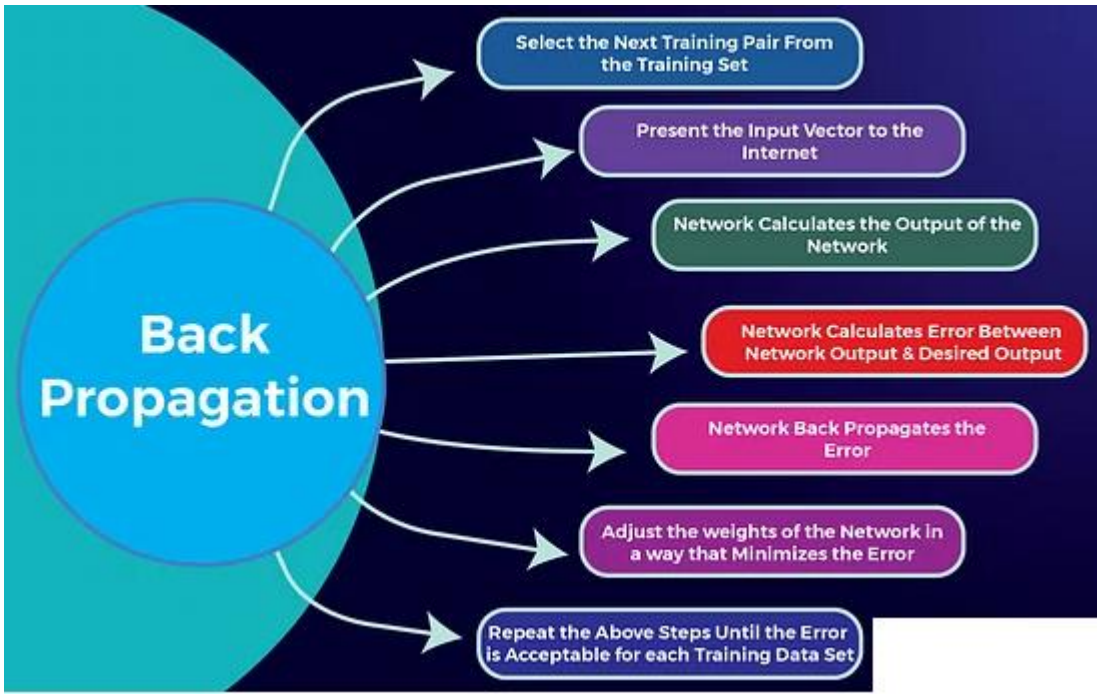
The **Multi-Layer-Perceptron** was **first introduced** by **M. Minsky and S. Papert in 1969.** It is an **extended Perceptron** and has **one ore more hidden neuron layers between its input and output layers. A Multi-Layer-Perceptron is able to solve every logical operation, including the XOR problem.**

Multilayer perceptron's (MLPs) are **feed forward neural networks** trained with the **standard back propagation algorithm.** Back propagation is the most important step for training artificial neural networks. **By using the Back propagation, is a procedure to repeatedly adjust the weights so as to minimize the difference between actual output and desired output.**

The Architecture of ANN is :



Back Propagation Algorithm Steps:

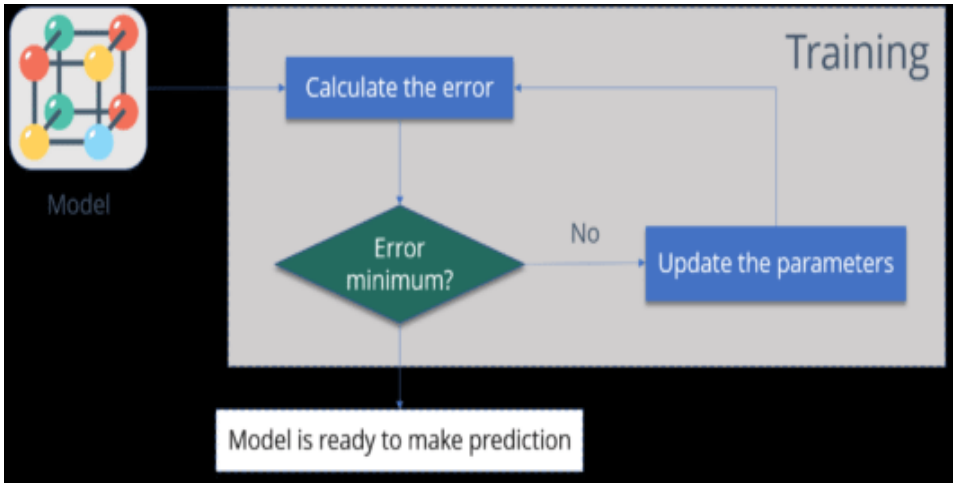


In an **Artificial Neural Network**, the **values of weights and biases** are **randomly initialized**. Due to **random initialization**, the **neural network probably has errors for the given inputs.** So, We need to **reduce error values** as much as possible. So, for **reducing these error values,** we need a mechanism that can **compare the desired output of the neural network** with the **Target network's output .**

Suppose, that **consists of errors** and **adjusts its weights and biases** such that it **gets closer to the desired output after each iteration**. For this, we **train the network** such that it **back propagates and updates the weights and biases**. This is the concept of the **back propagation algorithm.**

Below are the steps that an artificial neural network follows to **gain maximum accuracy** and **minimize error values**

One way to train our model is called as Back propagation. Consider the diagram below:



Let us summarize the steps for you:

1. **Calculate the error** – How far is your model output from the actual output.
2. **Minimum Error** – Check whether the error is minimized or not.
3. **Update the parameters** – If the error is huge then, update the parameters (weights and biases). After that again check the error. Repeat the process until the error becomes minimum.
4. **Model is ready to make a prediction** – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.

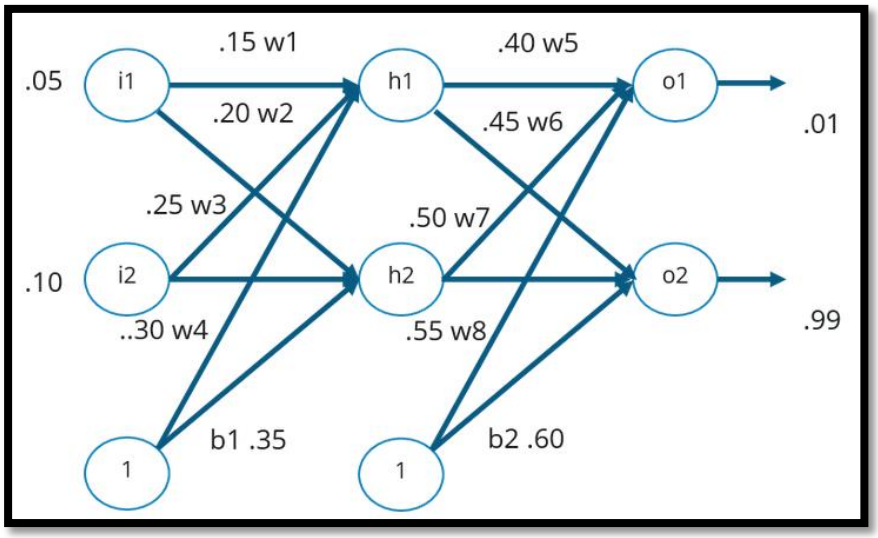
It involves below steps:

1. Install the Packages:

(a) Numpy: Numpy Python library is used for including any type **of mathematical operation in the code**. It is the fundamental package for scientific calculation in Python. It also supports to **add large, multidimensional arrays and matrices**. **So, in Python, we can import it as:**

import numpy as np

Here we have taken with One Example



The above network contains the following:

1. **Two inputs :** X1 and X2
2. **Two hidden neurons :** h1 and h2
3. **Two output neurons :** o1 and o2
4. **Two biases :** b1 and b2

Below the steps involved the Back Propagation

Step – 1: Forward Propagation

Step – 2: Backward Propagation

Step – 3: Putting all the values together and calculating the updated weight value

Step-1: Forward Propagation:

1. **First we have to initialize the Weights:** **ie, here we have to read the input from the user**

   Here we have initialized two inputs : X1 and X2

   For Eg: input_data = np.array(input("Enter input data separated by commas: ").split(','), dtype=float)

2. **Next, we have to read the Number of Hidden Layers from user.**

   For Eg: n_hidden_layers = int(input("Enter number of hidden layers: "))

3. **Next, we have to read the Number of Output Neurons from user.**

   For Eg: n_output_neurons = int(input("Enter number of output neurons: "))

4. **Next, Read the Bias Values from User**

   For Eg: bias_values = []

   for i in range(n_hidden_layers+1):

   bias_values.append(np.array(input(f"Enter bias values for layer {i+1} separated by commas: ").split(','), dtype=float))

5. **Next, we have to give the weights for the given inputs X1 and X2.**

   For Eg : for i in range(n_hidden_layers):

   hidden_layer_sizes.append(int(input(f"Enter number of neurons in hidden layer {i+1}: ")))

   if i == 0:

   weights.append(np.array(input(f"Enter weights for input layer to hidden layer {i+1} separated by commas: ").split(','), dtype=float).reshape(input_layer_size, hidden_layer_sizes[i]))

   else:

   weights.append(np.array(input(f"Enter weights for hidden layer {i} to hidden layer {i+1} separated by commas: ").split(','), dtype=float).reshape(hidden_layer_sizes[i-1], hidden_layer_sizes[i]))

   weights.append(np.array(input(f"Enter weights for hidden layer {n_hidden_layers} to output layer separated by commas: ").split(','), dtype=float).reshape(hidden_layer_sizes[-1], n_output_neurons))

6. **Next ,we have to Initialize the Learning Rate and Number of Iterations**

   learning_rate = 0.1

   num_iterations = 1000

7. **Next, Read the Target Output from user**

   For Eg: target = np.array(input(f"Enter target output separated by commas: ").split(','), dtype=float)

Next, we have to Train the Network (Feed Forward Propagation)

- **Here I1, I2 are the Input Layers and W1, W2, W3, W4 are the corresponding Weights.**
- **From the above Network, First we can Calculate the Net H1 and Net H2.**

$$net\,H_1 = I_1.W_1 + I_2.W_2 + b_1 \quad .....(1)$$

$$net\,H_2 = I_1.W_3 + I_2.W_4 + b_1 \quad .....(2)$$

After Calculating net H1 and net H2. compute Out H1 and Out H2 as follows:

$$Out\,H_1 = \frac{1}{1+e^{-net H_1}} \quad .....(3)$$

$$Out\,H_2 = \frac{1}{1+e^{-net H_2}} \quad .....(4)$$

Similarly, we can Calculating net O1 and net O2.

$$net\,O_1 = W_5.out\,H_1 + W_6.out\,H_2 + b_2 \quad .....(5)$$

$$net\,O_2 = W_7.out\,H_1 + W_8.out\,H_2 + b_2 \quad .....(6)$$

From the above Equations, we can calculate Out O1 and Out O2 as follows:

$$Out\,O_1 = \frac{1}{1+e^{-netO_1}} \quad .....(7)$$

$$Out\,O_2 = \frac{1}{1+e^{-netO2}} \quad .....(8)$$

 # Feedforward

```
hidden_layers = [input_data]
for j in range(len(hidden_layer_sizes)):
    hidden_layer = sigmoid(np.dot(hidden_layers[j], weights[j]) + bias_values[j])
    hidden_layers.append(hidden_layer)
output = sigmoid(np.dot(hidden_layers[-1], weights[-1]) + bias_values[-1])
```

Step-2: Back Propagation:

- When we got the Error. i.e, Target Output O1 and Actual Output O1.
- Similarly, Target Output O2 and Actual Output O2.
- So that we have to move Backward and we have to adjust the weights of W5, W6, W7, W8.
- Now we have to Calculate the Output Values.
- If we get the Expected Output and Actual Outputs are Same, We will stop.

Next, we can Calculating the Total Error $$E_T = E_{o_1} + E_{o_2} \quad .....(9)$$

<u>For Eg:</u> # Backpropagation

```
error = output - target
delta = error * sigmoid_derivative(output)
deltas = [delta]
for j in range(len(hidden_layer_sizes)-1, -1, -1):
    delta = np.dot(deltas[-1], weights[j+1].T) * sigmoid_derivative(hidden_layers[j+1])
    deltas.append(delta)
deltas.reverse()
```

<u>If, Error Comes, we need to Update the Weights and Biases :</u>

Here, we need to use the Formula for Gradient Descent to update the Weights.

$$^{*}W_x = W_x - a\left(\frac{\partial Error}{\partial W_x}\right)$$

Old weight — Derivative of Error with respect to weight — New weight — Learning rate

Suppose we have to update the Weight W5:

$$W_5^{+} = W_5 - a\left(\frac{\partial Error}{\partial W_5}\right)$$

This can be Written as :

$$\frac{\partial E_{Total}}{\partial_{W_5}} = \frac{\partial E_{Total}}{\partial_{OutO_1}} \cdot \frac{\partial_{OutO_1}}{\partial_{netO_1}} \cdot \frac{\partial_{netO_1}}{\partial_{W_5}} \quad .....(10)$$

The Final Calculation is :

$$\frac{\partial E_{Total}}{\partial w_5} = \frac{\partial E_{Total}}{\partial out_{O1}} * \frac{\partial out_{O1}}{\partial net_{O1}} * \frac{\partial net_{O1}}{\partial w_5}$$

$$= -\left[Target_{O1} - output_{O1}\right] * out_{O1}\left[1 - out_{O1}\right] * out_{H1}$$

```
# Update weights and biases
    for j in range(len(hidden_layer_sizes)+1):
        if j == 0:
            layer_input = input_data
        else:
            layer_input = hidden_layers[j]
        dtran = np.transpose(deltas[j].reshape(-1,1))
        x = np.dot(layer_input.reshape(-1,1), dtran)
        #print(x.shape,weights[j].shape)
        weights[j] -= learning_rate * x
```

Step – 3: Putting all the values together and calculating the updated weight value

```
    # Print final output and weights
        print(f"Final output: {output}")
        print(f"Target output: {target}")
        for i in range(len(weights)):
                print(f"Layer {i+1} weights: {weights[i]}")
        for i in range(len(bias_values)):
                print(f"Layer {i+1} biases: {bias_values[i]}")
```

PROGRAM:

```
#Importing lib for data pre-processing and algorithm building
        import numpy as np
# Define sigmoid activation function
        def sigmoid(x):
            return 1 / (1 + np.exp(-x))
# Define derivative of sigmoid function
        def sigmoid_derivative(x):
            return x * (1 - x)
# Read input from user
input_data = np.array(input("Enter input data separated by commas: ").split(','), dtype=float)
#Read number of hidden layers from user
n_hidden_layers = int(input("Enter number of hidden layers: "))
# Read number of output neurons from user
n_output_neurons = int(input("Enter number of output neurons: "))
# Read bias values from user
bias_values = []
for i in range(n_hidden_layers+1):
    bias_values.append(np.array(input(f"Enter bias values for layer {i+1} separated by
    commas: ").split(','), dtype=float))
# Define neural network architecture
input_layer_size = input_data.shape[0]
hidden_layer_sizes = []
weights = []
# Assign the weights for the given inputs
for i in range(n_hidden_layers):
    hidden_layer_sizes.append(int(input(f"Enter number of neurons in hidden layer
   {i+1}: ")))
    if i == 0:
        weights.append(np.array(input(f"Enter weights for input layer to hidden layer


   {i+1} separated by commas: ").split(','), dtype=float).reshape(input_layer_size,
```

```
    hidden_layer_sizes[i]))
    else:
        weights.append(np.array(input(f"Enter weights for hidden layer {i} to hidden
        layer {i+1} separated by commas: ").split(','),
    dtype=float).reshape(hidden_layer_sizes[i-1], hidden_layer_sizes[i]))
        weights.append(np.array(input(f"Enter weights for hidden layer {n_hidden_layers}
        to output layer separated by commas: ").split(','),
        dtype=float).reshape(hidden_layer_sizes[-1], n_output_neurons))
# Set hyperparameters
learning_rate = 0.1
num_iterations = 1000
# Read target output from user
target = np.array(input(f"Enter target output separated by commas: ").split(','),
dtype=float)
# Train the neural network
for i in range(num_iterations):
 # Feed forward Network
    hidden_layers = [input_data]
    for j in range(len(hidden_layer_sizes)):
        hidden_layer = sigmoid(np.dot(hidden_layers[j], weights[j]) + bias_values[j])
        hidden_layers.append(hidden_layer)
    output = sigmoid(np.dot(hidden_layers[-1], weights[-1]) + bias_values[-1])
    # Backpropagation
    error = output - target
    delta = error * sigmoid_derivative(output)
    deltas = [delta]
    for j in range(len(hidden_layer_sizes)-1, -1, -1):
        delta = np.dot(deltas[-1], weights[j+1].T) *
                sigmoid_derivative(hidden_layers[j+1])
        deltas.append(delta)
    deltas.reverse()
```

```
# Update weights and biases
    for j in range(len(hidden_layer_sizes)+1):
        if j == 0:
            layer_input = input_data
        else:
            layer_input = hidden_layers[j]
        dtran = np.transpose(deltas[j].reshape(-1,1))
        x = np.dot(layer_input.reshape(-1,1), dtran)
        #print(x.shape,weights[j].shape)
        weights[j] -= learning_rate * x
# Print final output and weights
print(f"Final output: {output}")
print(f"Target output: {target}")
for i in range(len(weights)):
    print(f"Layer {i+1} weights: {weights[i]}")
for i in range(len(bias_values)):
    print(f"Layer {i+1} biases: {bias_values[i]}")
```

INPUT/OUTPUT:

```
Final output: [0.0725931  0.93353271]
Target output: [0.01 0.99]
Layer 1 weights: [[0.24500364 0.29141985]
 [0.44000729 0.48283971]]
Layer 2 weights: [[-2.6646643   1.64760408]
 [-2.5732862   1.75091177]]
Layer 1 biases: [0.35]
Layer 2 biases: [0.6]
```

CONCLUSION: Program is executed successfully without any error.