ARTIFICIAL NEURAL NETWORKS (ANN) LAB

Lab Demonstrator: Shakeela Shaheen

LAB 02: BACKPROPAGATION AND ITS IMPLEMENTATION FROM SCRATCH

LEARNING OBJECTIVES:

By the end of this lab, students will:

- Understand **learning as optimization**, not magic
- Understand **why gradients are necessary for learning**
- Clearly define **differentiability** and explain why it matters
- Understand why **step / 0–1 functions cannot learn**
- Understand **continuous activation functions** at a deep level
- Apply the **chain rule step by step** to neural networks
- Derive **backpropagation mathematically**
- Translate **every mathematical symbol into NumPy code**
- Understand what deep learning libraries automate

0. WHY THIS LAB EXISTS (READ VERY CAREFULLY)

In Lab 01, you learned **what a neuron does**. But knowing what a neuron does is not knowing how a network learns.

A neural network before training is **just a random decision machine**.

Learning is not about:

- More data
- Faster computers
- Bigger GPUs

Learning is about **changing parameters in the right direction**. That direction comes from **gradients**.

This lab exists to answer one fundamental question:

How does a neural network know which direction to change its weights?

The answer is: **Backpropagation + Differentiability**

1. WHAT LEARNING REALLY IS ?

Imagine Teaching a student. A student answers a question incorrectly. You do **not** say:

"You're wrong. Fix yourself."

You say:

- Which concept was misunderstood?
- How badly was it misunderstood?
- Which earlier idea caused this mistake?

That process is **backpropagation**.

- Final mistake, traced backward
- Responsibility assigned gradually
- Bigger misunderstanding, bigger correction

Neural networks learn **exactly the same way**, except using calculus.

2. WHY LEARNING REQUIRES GRADIENTS

What Is Learning, Mathematically?

Learning means **minimizing a loss function**. Loss is a number that measures how wrong the network is.

So learning becomes:

Change parameters to reduce loss

This is an optimization problem.

How Do We Minimize a Function?

Imagine a curve (loss vs weight):

- If slope > 0, move left
- If slope < 0, move right
- If slope = 0, minimum

That slope is the **derivative**.

Without derivatives, learning has no direction.

3. WHY 0 / 1 (STEP) FUNCTIONS FAIL COMPLETELY

Step Function Again

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

What Happens During Learning?

- You change a weight slightly
- Output stays exactly the same
- Loss does not change
- Gradient = 0

So the network hears:

"No matter what you do, nothing improves."

That is **learning paralysis**.

4. WHAT DOES "DIFFERENTIABLE" MEAN? (VERY IMPORTANT)

A function is **differentiable** if: A tiny change in input causes a tiny, predictable change in output.

Mathematical Meaning

A function $f(x)$is differentiable at $x$if:

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

exists.

This limit is the **derivative**.

Think About Steering a Car

- Smooth steering wheel, small turn, small direction change
- Locked steering wheel, no matter what you do, direction doesn't change

A **step function** is a locked steering wheel.

Technically speaking,

- Differentiable means smooth curve
- Tangent line exists
- Slope exists at every point

The derivative answers:

"If I nudge the input slightly, which direction and how strongly does the output move?"

That answer is exactly what learning needs.

5. WHY CONTINUOUS FUNCTIONS FIX EVERYTHING

Continuous Activation = Volume Knob

Instead of:

- OFF or ON

We get:

- Quiet
- Medium
- Loud

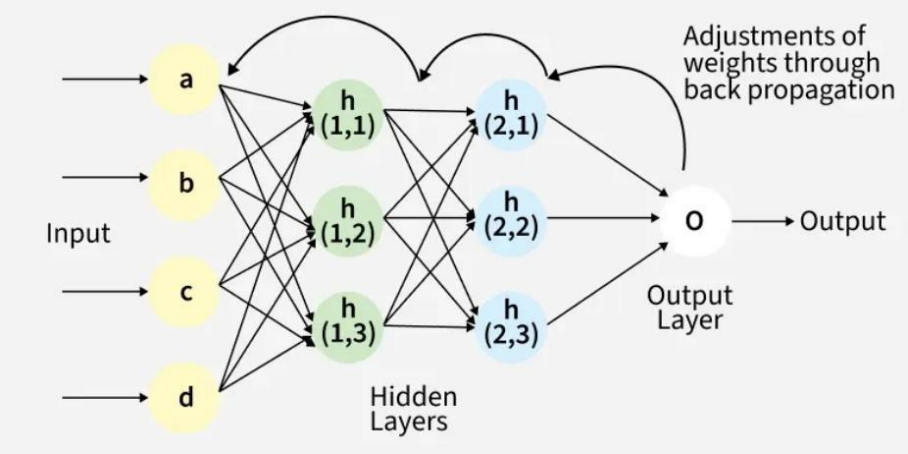
Now learning can say:

- "Reduce volume slightly"
- "Increase volume slightly"

That's gradient descent.

6. BACKPROPAGATION:

Backpropagation, short for Backward Propagation of Errors, is a key algorithm used to train neural networks by minimizing the difference between predicted and actual outputs. It works by propagating errors backward through the network, using the chain rule of calculus to compute gradients and then iteratively updating the weights and biases. Combined with optimization techniques like gradient descent, backpropagation enables the model to reduce loss across epochs and effectively learn complex patterns from data.



Backpropagation is the process of figuring out how much each weight is responsible for the final error.

Back Propagation plays a critical role in how neural networks improve over time. Here's why:

1. **Efficient Weight Update**: It computes the gradient of the loss function with respect to each weight using the chain rule making it possible to update weights efficiently.
2. **Scalability**: The Back Propagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.
3. **Automated Learning**: With Back Propagation the learning process becomes automated and the model can adjust itself to optimize its performance.

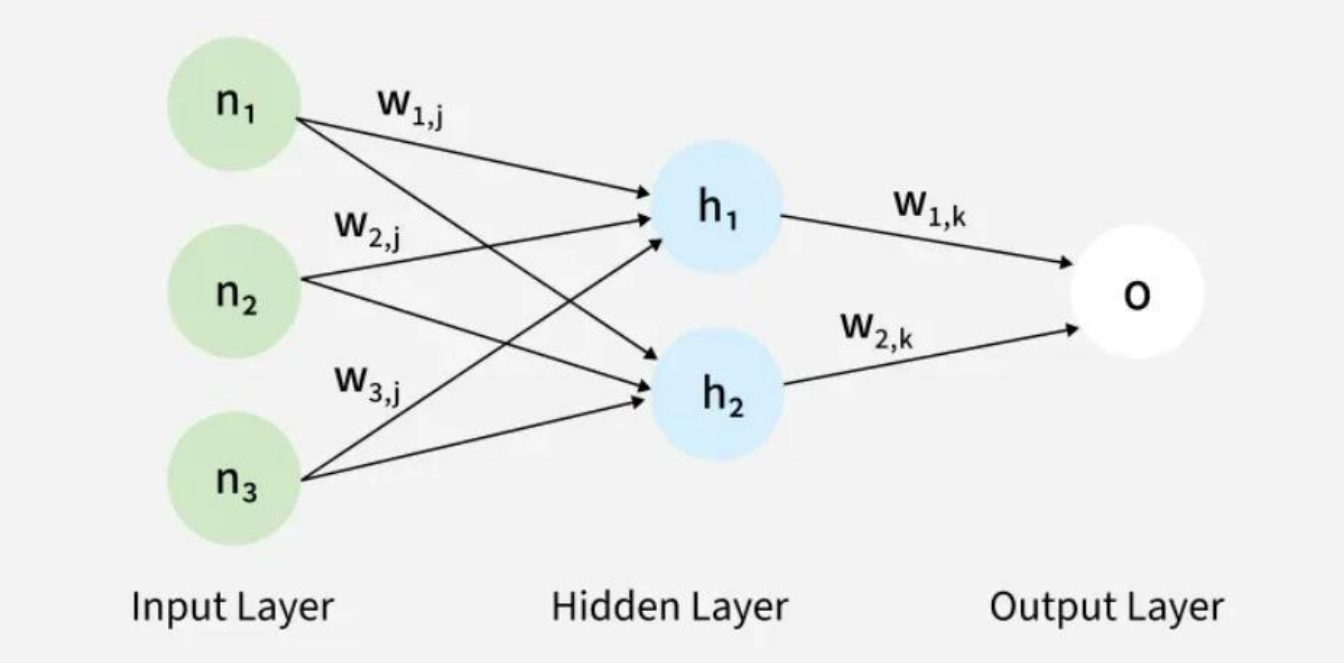
6.1: WORKING OF BACK PROPAGATION ALGORITHM

The Back Propagation algorithm involves two main steps: the Forward Pass and the Backward Pass.

1. Forward Pass Work

In forward pass the input data is fed into the input layer. These inputs combined with their respective weights are passed to hidden layers. For example in a network with two hidden layers (h1 and h2) the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

Each hidden layer computes the weighted sum (`a`) of the inputs then applies an activation function like ReLU (Rectified Linear Unit) to obtain the output (`o`). The output is passed to the next layer where an activation function such as softmax converts the weighted outputs into probabilities for classification.



2. Backward Pass

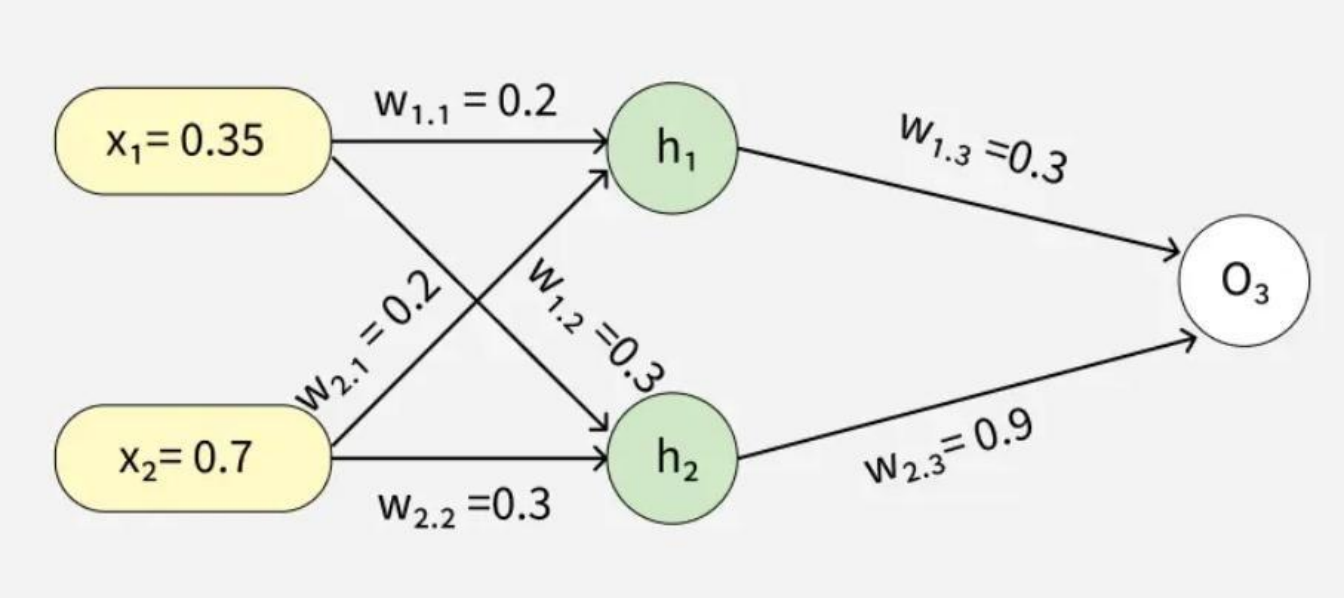
In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the Mean Squared Error (MSE) given by:

$$\textbf{MSE} = (\textbf{Predicted Output} - \textbf{Actual Output})^2$$

Once the error is calculated the network adjusts weights using gradients which are computed with the chain rule. These gradients indicate how much each weight and bias should be adjusted to minimize the error in the next iteration. The backward pass continues layer by layer ensuring that the network learns and improves its performance. The activation function through its derivative plays a crucial role in computing these gradients during Back Propagation.

6.2: EXAMPLE OF BACK PROPAGATION IN MACHINE LEARNING

Let's walk through an example of Back Propagation in machine learning. Assume the neurons use the sigmoid activation function for the forward and backward pass. The target output is 0.5 and the learning rate is 1.



Forward Propagation

1. Initial Calculation

The weighted sum at each node is calculated using:

$$aj = \sum(wi, j * xi)$$

Where,

- a_j is the weighted sum of all the inputs and weights at each node
- $w_{i,j}$ represents the weights between the i^{th}input and the j^{th} neuron

- x_i represents the value of the i^{th} input

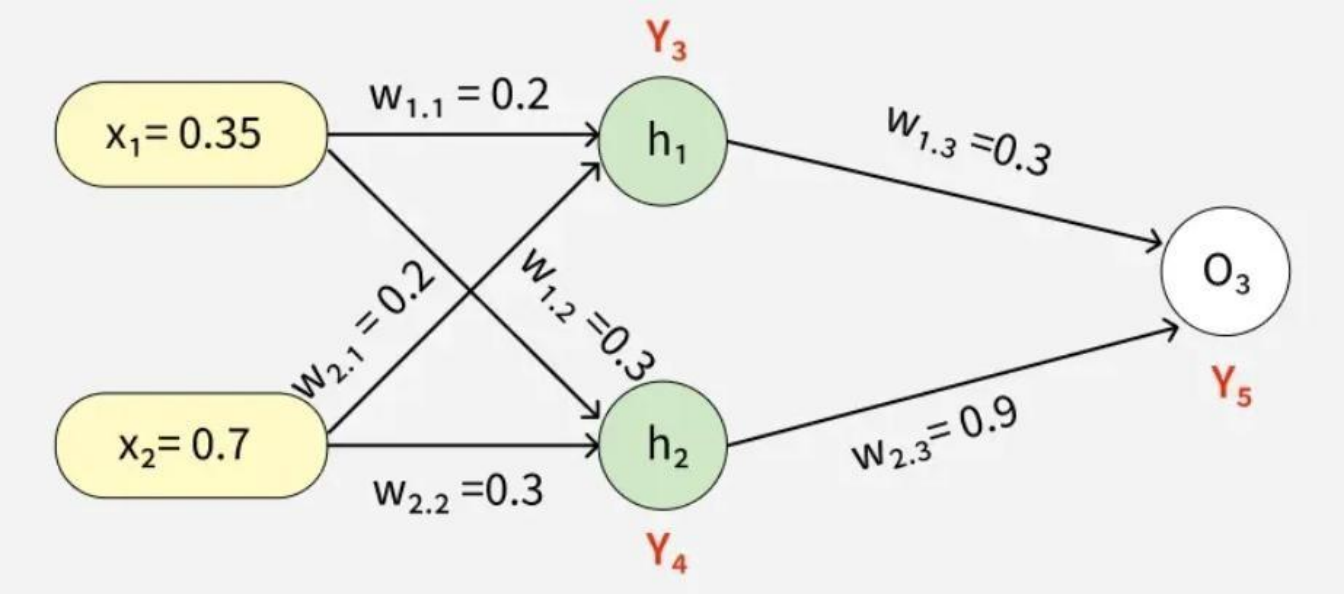
O (output): After applying the activation function to a, we get the output of the neuron:

o_j = *activation function*(a_j)

2. Sigmoid Function

The sigmoid function returns a value between 0 and 1, introducing non-linearity into the model.

$$y_j = \frac{1}{1 + e^{-aj}}$$



3. Computing Outputs

At h1 node

$$\begin{aligned} a_1 &= (w_{1,1}x_1) + (w_{2,1}x_2) \\ &= (0.2 * 0.35) + (0.2 * 0.7) \\ &= 0.21 \end{aligned}$$

Once we calculated the a_1 value, we can now proceed to find the y_3 value:

$$y_j = F(a_j) = \frac{1}{1 + e^{-a_1}}$$

$$y_3 = F(0.21) = \frac{1}{1 + e^{-0.21}}$$

$$y_3 = 0.56$$

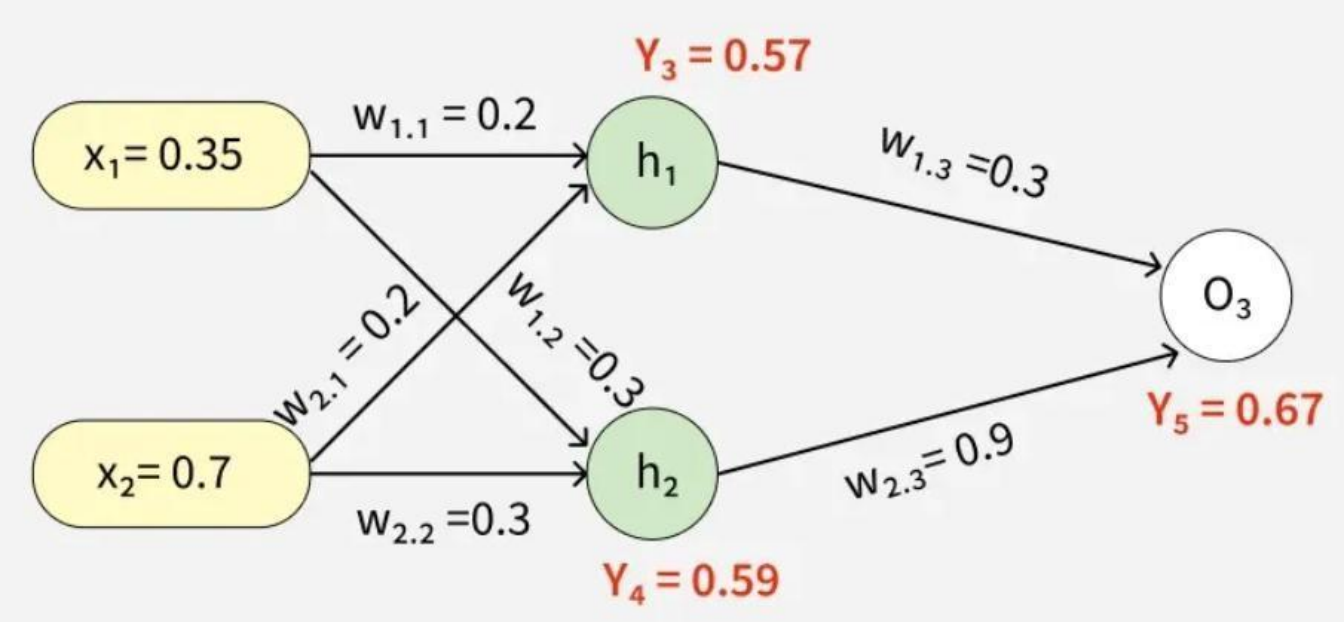
Similarly find the values of y_4 at **h2** and y_5 at O_3

$$a_2 = (w_{1,2} * x_1) + (w_{2,2} * x_2) = (0.3 * 0.35) + (0.3 * 0.7) = 0.315$$

$$y_4 = F(0.315) = \frac{1}{1 + e^{-0.315}}$$

$$a3 = (w_{1,3} * y_3) + (w_{2,3} * y_4) = (0.3 * 0.57) + (0.9 * 0.59) = 0.702$$

$$y_5 = F(0.702) = \frac{1}{1 + e^{-0.702}} = 0.67$$



4. Error Calculation

Our actual output is 0.5 but we obtained 0.67. To calculate the error we can use the below formula:

$$Error_j = y_{target} - y_5$$

$$=> 0.5 - 0.67 = -0.17$$

Using this error value we will be backpropagating.

Back Propagation

1. Calculating Gradients

The change in each weight is calculated as:

$$\Delta w_{ij} = \eta \times \delta_j \times O_j$$

Where:

- δ_j is the error term for each unit,
- η is the learning rate.

Why multiply by learning rate (η)?

Because:

- Gradient gives **direction**, not step size
- Learning rate controls **how far we move**

Gradient = direction to walk and η = step size

Why multiply by output of previous neuron (Oi)?

Because:

- A weight only matters **if a signal passed through it**
- If input was zero → weight had no effect

This term answers:

"How much was this weight *used*?"

Why multiply by δj (delta)?

Because:

- δj measures **how responsible neuron j is for the error**
- If neuron didn't cause the mistake, weight shouldn't change

This term answers:

"How much blame does this neuron deserve?"

Mathematically:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

So the *real* goal is:

$$\frac{\partial L}{\partial w}$$

Backpropagation is **just a clever way to compute this derivative efficiently**.

$$\delta_j = o_j(1-o_j)(t_j - o_j) \qquad \text{if } j \text{ is an output unit}$$

$$\delta_j = o_j(1-o_j)\sum_k \delta_k w_{kj} \qquad \text{if } j \text{ is a hidden unit}$$

2. Output Unit Error

For O3:

$$\delta_5 = y_5(1 - y_5)(y_{target} - y_5)$$

$$= 0.67(1 - 0.67)(-0.17) = -0.0376$$

Why $y_5(1 - y_5)$?

That is:

Derivative of sigmoid

Why derivative?
Because:

- We're asking: *"If I slightly change the input of this neuron, how much does its output change?"*

Neurons near saturation (0 or 1) learn **slowly**.

INTERPRETATION

δ_5 = **how much O3 is responsible for the error**

- Negative → output must go down
- Magnitude → how strong the correction should be

3. Hidden Unit Error

For h1:

$$\delta_3 = y_3(1 - y_3)(w_{1,3} \times \delta_5)$$

$$= 0.56(1 - 0.56)(0.3 \times -0.0376) = -0.0027$$

For h2:

$$\delta_4 = y_4(1 - y_4)(w_{2,3} \times \delta_5)$$

$$= 0.59(1 - 0.59)(0.9 \times -0.0376) = -0.0819$$

4. Weight Updates

For the weights from hidden to output layer:

$$\Delta w_{2,3} = 1 \times (-0.0376) \times 0.59 = -0.022184$$

New weight:

$$w_{2,3}(\text{new}) = -0.022184 + 0.9 = 0.877816$$

For weights from input to hidden layer:

$$\Delta w_{1,1} = 1 \times (-0.0027) \times 0.35 = 0.000945$$

New weight:

$$w_{1,1}(\text{new}) = 0.000945 + 0.2 = 0.200945$$

Similarly other weights are updated:

- $w_{1,2}(\text{new}) = 0.273225$
- $w_{1,3}(\text{new}) = 0.086615$
- $w_{2,1}(\text{new}) = 0.269445$
- $w_{2,2}(\text{new}) = 0.18534$

The updated weights are illustrated below

Challenges

While Back Propagation is useful it does face some challenges:

- **Vanishing Gradient Problem**: In deep networks the gradients can become very small during Back Propagation making it difficult for the network to learn. This is common when using activation functions like sigmoid or tanh.
- **Exploding Gradients**: The gradients can also become excessively large causing the network to diverge during training.
- **Overfitting:** If the network is too complex it might memorize the training data instead of learning general patterns.

7. Back Propagation Implementation in Python for XOR Problem

This code demonstrates how Back Propagation is used in a neural network to solve the XOR problem. The neural network consists of:

1. Defining Neural Network

We define a neural network as Input layer with 2 inputs, Hidden layer with 4 neurons, Output layer with 1 output neuron and use **Sigmoid** function as activation function.

- **self.input_size = input_size**: stores the size of the input layer
- **self.hidden_size = hidden_size:** stores the size of the hidden layer

- **self.weights_input_hidden = np.random.randn(self.input_size, self.hidden_size)**: initializes weights for input to hidden layer
- **self.weights_hidden_output = np.random.randn(self.hidden_size, self.output_size)**: initializes weights for hidden to output layer
- **self.bias_hidden = np.zeros((1, self.hidden_size)):** initializes bias for hidden layer
- **self.bias_output = np.zeros((1, self.output_size)):** initializes bias for output layer

```
import numpy as np


class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

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

        self.bias_hidden = np.zeros((1, self.hidden_size))
        self.bias_output = np.zeros((1, self.output_size))

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)
```

2. Defining Feed Forward Network

In Forward pass inputs are passed through the network activating the hidden and output layers using the sigmoid function.

- **self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden**: calculates activation for hidden layer
- **self.hidden_output= self.sigmoid(self.hidden_activation)**: applies activation function to hidden layer
- **self.output_activation= np.dot(self.hidden_output, self.weights_hidden_output) +**

self.bias_output: calculates activation for output layer

- **self.predicted_output = self.sigmoid(self.output_activation):** applies activation function to output layer

```
def feedforward(self, X):
    self.hidden_activation = np.dot(
        X, self.weights_input_hidden) + self.bias_hidden
    self.hidden_output = self.sigmoid(self.hidden_activation)

    self.output_activation = np.dot(
        self.hidden_output, self.weights_hidden_output) + self.bias_output
    self.predicted_output = self.sigmoid(self.output_activation)

    return self.predicted_output
```

3. Defining Backward Network

In Backward pass or Back Propagation the errors between the predicted and actual outputs are computed. The gradients are calculated using the derivative of the sigmoid function and weights and biases are updated accordingly.

- **output_error = y - self.predicted_output:** calculates the error at the output layer
- **output_delta = output_error * self.sigmoid_derivative(self.predicted_output):** calculates the delta for the output layer
- **hidden_error = np.dot(output_delta, self.weights_hidden_output.T):** calculates the error at the hidden layer
- **hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output):** calculates the delta for the hidden layer
- **self.weights_hidden_output += np.dot(self.hidden_output.T, output_delta) * learning_rate:** updates weights between hidden and output layers
- **self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate:** updates weights between input and hidden layers

```
def backward(self, X, y, learning_rate):
    output_error = y - self.predicted_output
    output_delta = output_error * \
        self.sigmoid_derivative(self.predicted_output)

    hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
    hidden_delta = hidden_error * self.sigmoid_derivative(self.hidden_output)

    self.weights_hidden_output += np.dot(self.hidden_output.T,
                                         output_delta) * learning_rate
    self.bias_output += np.sum(output_delta, axis=0,
                               keepdims=True) * learning_rate
    self.weights_input_hidden += np.dot(X.T, hidden_delta) * learning_rate
    self.bias_hidden += np.sum(hidden_delta, axis=0,
                               keepdims=True) * learning_rate
```

4. Training Network

The network is trained over 10,000 epochs using the Back Propagation algorithm with a learning rate of 0.1 progressively reducing the error.

- **output = self.feedforward(X):** computes the output for the current inputs
- **self.backward(X, y, learning_rate):** updates weights and biases using Back Propagation
- **loss = np.mean(np.square(y - output)):** calculates the mean squared error (MSE) loss

```
def train(self, X, y, epochs, learning_rate):
    for epoch in range(epochs):
        output = self.feedforward(X)
        self.backward(X, y, learning_rate)
        if epoch % 4000 == 0:
            loss = np.mean(np.square(y - output))
            print(f"Epoch {epoch}, Loss:{loss}")
```

5. Testing Neural Network

- **X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]):** defines the input data
- **y = np.array([[0], [1], [1], [0]]):** defines the target values
- **nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1):** initializes the neural network
- **nn.train(X, y, epochs=10000, learning_rate=0.1):** trains the network
- **output = nn.feedforward(X)**: gets the final predictions after training

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

output = nn.feedforward(X)
print("Predictions after training:")
print(output)
```

LAB TASKS

You are working as a **Machine Learning Intern** at a healthcare startup developing a **simple neural network** to predict whether a patient's cognitive response score indicates early risk of a neurological disorder.

The system takes **two normalized test scores** as input and produces a **single risk score** between 0 and 1 using a neural network.

During testing, the system produces an incorrect prediction.
Your task is to **manually analyze how the network learns from its mistake** using **backpropagation**.

Network Given

Architecture

- **Inputs:**
  $x_1 = 0.35, x_2 = 0.7$
- **Hidden Layer:**
  - $h_1 \rightarrow$ output y_3
  - $h_2 \rightarrow$ output y_4
- **Output Layer:**
  - $o_1 \rightarrow$ output y_5
- **Activation Function:** Sigmoid
- **Learning Rate:** $\eta = 1$
- **Target Output:** $y_{target} = 0.5$

Connection	Weight
(w_{1,1})	0.2
(w_{2,1})	0.2
(w_{1,2})	0.3
(w_{2,2})	0.3
(w_{1,3})	0.3
(w_{2,3})	0.9

Task 1: Forward Pass Analysis (Understanding the Prediction)

1. Compute the **weighted sum** at each hidden neuron.
2. Apply the **sigmoid activation function**.
3. Compute the **final output** of the network.
4. Compare the predicted output with the target value.

Why is the network's prediction considered incorrect?

Task 2: Error Calculation (Identifying the Mistake)

1. Compute the **output error**:

$$Error = y_{target} - y_5$$

What does the **sign of the error** tell you about the prediction?

Task 3: Output Neuron Responsibility (δ_5)

1. Compute the **error term for the output neuron** using:

$$\delta_5 = y_5(1 - y_5)(y_{target} - y_5)$$

Why do we multiply the error with the derivative of the sigmoid function?

Task 4: Hidden Neuron Responsibility (δ_3 and δ_4)

1. Compute the error term for each hidden neuron:

$$\delta_j = y_j(1 - y_j)(w_{j,output} \times \delta_5)$$

- Why do hidden neurons not directly use the target value?
- Why does a larger outgoing weight result in a larger hidden error?

Task 5: Weight Updates (Learning from Mistakes)

1. Update the weights between:
    - Hidden → Output layer
    - Input → Hidden layer
2. Use:

$$\Delta w = \eta \times \delta \times input$$

- Why do some weights change more than others?
- Why are some weight updates very small?

Task 6: Interpretation & Reflection (Critical Thinking)

Answer the following in **your own words**:

1. Explain backpropagation as a **process of blame assignment**.
2. What would happen if:
    - Learning rate was **very large**?
    - Learning rate was **very small**?
3. Why is backpropagation called **"backward" propagation**?