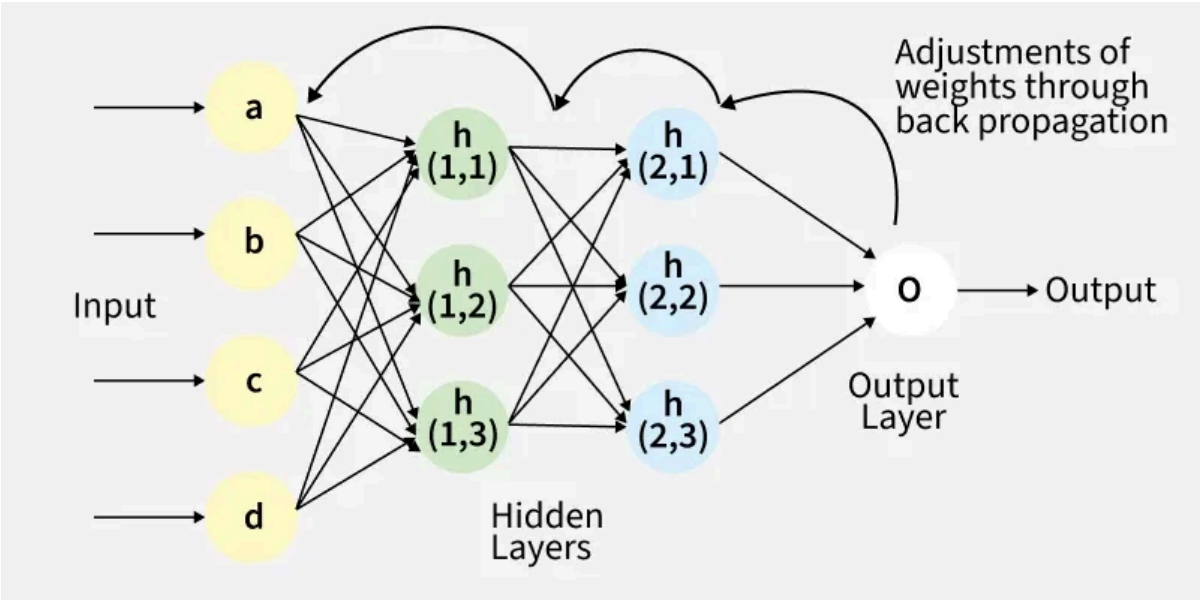
Backpropagation in Neural Network

Last Updated : 18 Sep, 2025

Back Propagation is also known as "Backward Propagation of Errors" is a method used to train neural network . Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.

It works iteratively to adjust weights and bias to minimize the cost function. In each epoch the model adapts these parameters by reducing loss by following the error gradient. It often uses optimization algorithms like **gradient descent** or **stochastic gradient descent**. The algorithm computes the gradient using the chain rule from calculus allowing it to effectively navigate complex layers in the neural network to minimize the cost function.



Fig(a) A simple illustration of how the backpropagation works by adjustments of weights

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.

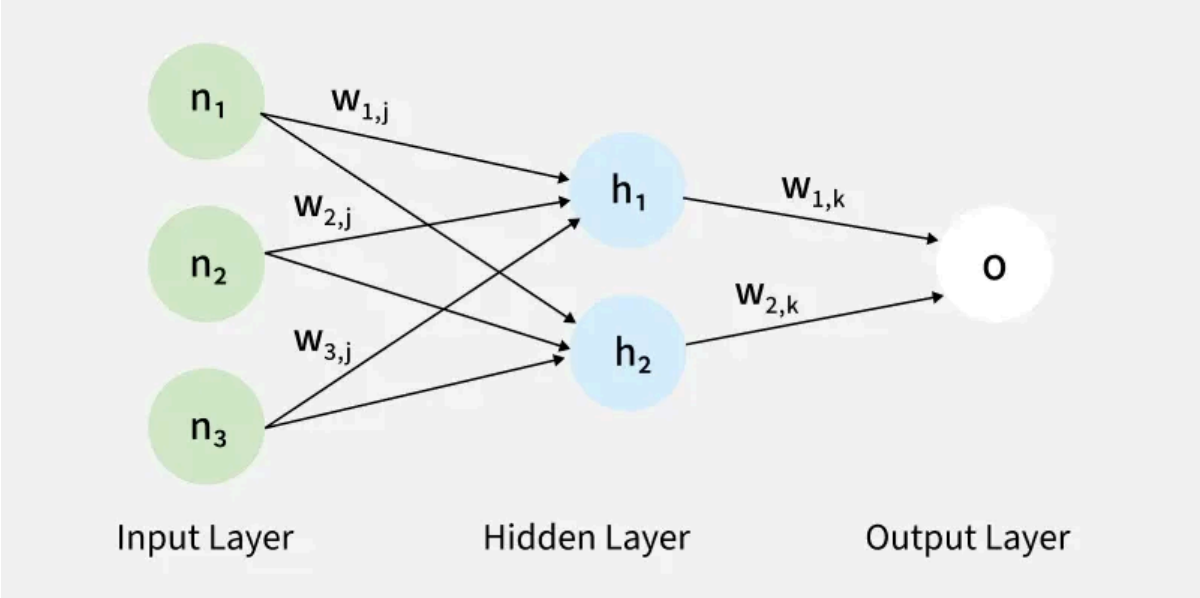
Working of Back Propagation Algorithm

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.



The forward pass using weights and biases

2. Backward Pass

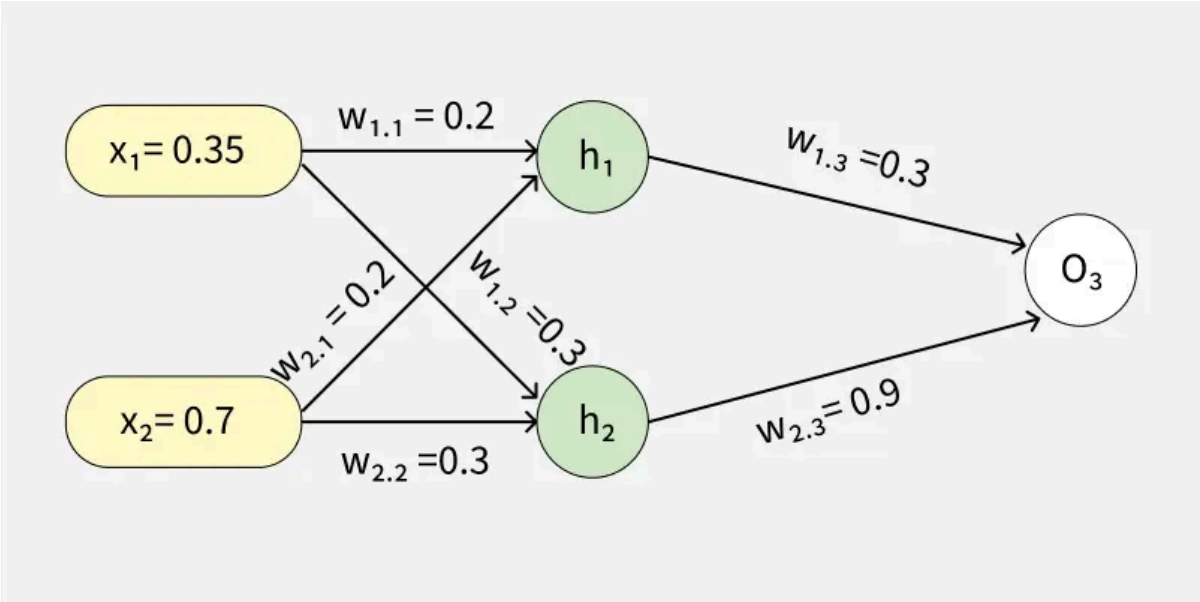
In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the Mean Squared Error (MSE) given by:

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

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.

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.



Example (1) of backpropagation sum

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

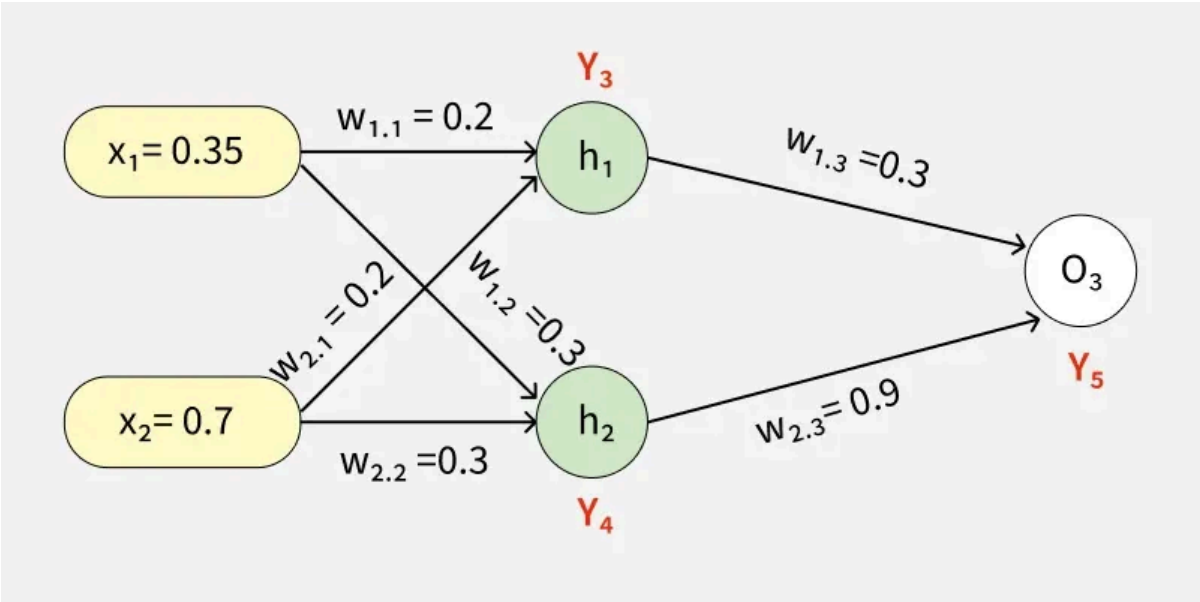
o (output): After applying the activation function to a, we get the output of the neuron:

$$o_j = \textit{activation function}(a_j)$$

2. Sigmoid Function

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

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



To find the outputs of y3, y4 and y5

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:

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

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

$$y_3 = 0.56$$

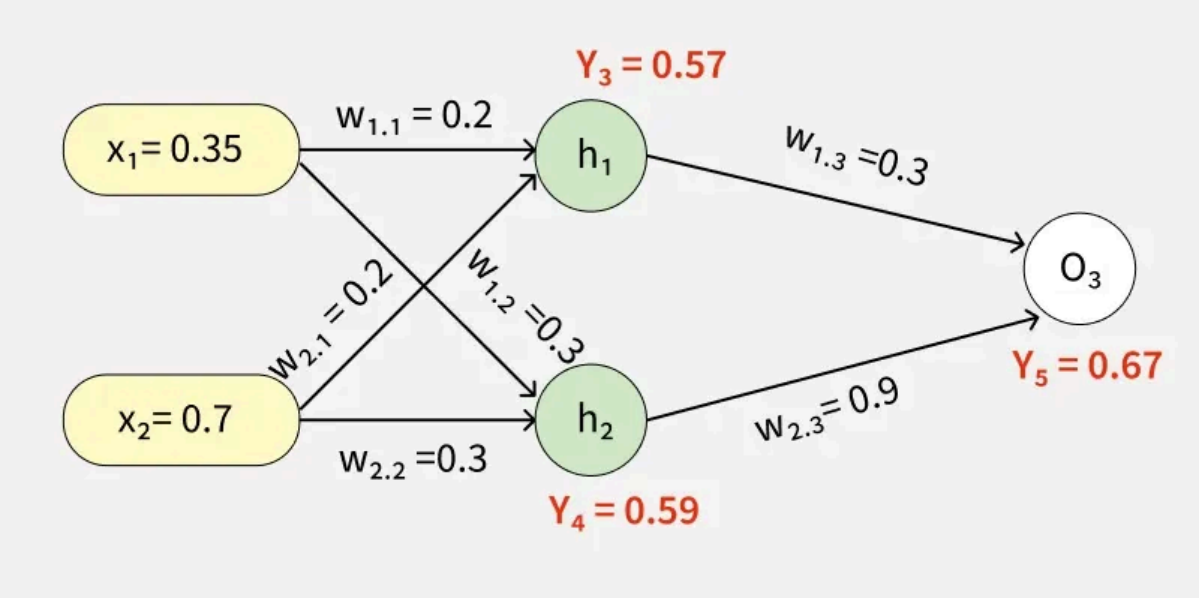
Similarly find the values of y_4 at **h_2** and y_5 at O_3

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

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

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

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



Values of y3, y4 and y5

4. Error Calculation

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

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

$$\Rightarrow 0.5 - 0.67 = -0.17$$

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.

2. Output Unit Error

For O3:

$$\begin{aligned}\delta_5 &= y_5(1 - y_5)(y_{target} - y_5) \\ &= 0.67(1 - 0.67)(-0.17) = -0.0376\end{aligned}$$

3. Hidden Unit Error

For h1:

$$\begin{aligned}\delta_3 &= y_3(1 - y_3)(w_{1,3} \times \delta_5) \\ &= 0.56(1 - 0.56)(0.3 \times -0.0376) = -0.0027\end{aligned}$$

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:

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

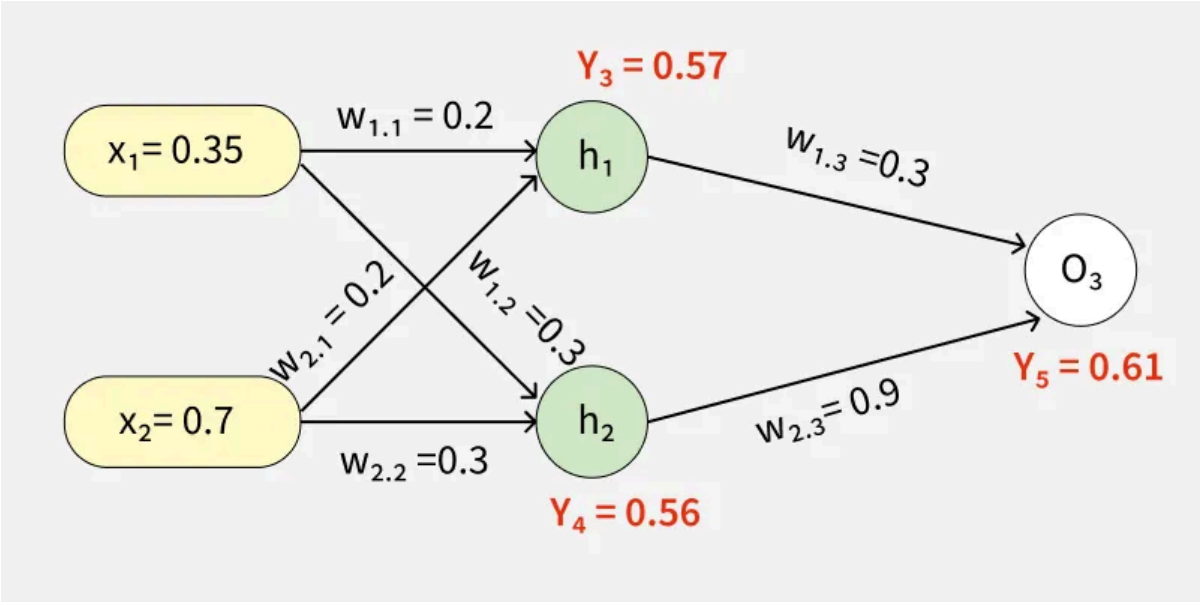
New weight:

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

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



Through backward pass the weights are updated

After updating the weights the forward pass is repeated hence giving:

- $y_3 = 0.57$
- $y_4 = 0.56$
- $y_5 = 0.61$

Since $y_5 = 0.61$ is still not the target output the process of calculating the error and backpropagating continues until the desired output is reached.

This process demonstrates how Back Propagation iteratively updates weights by minimizing errors until the network accurately predicts the output.

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

$$= 0.5 - 0.61 = -0.11$$

This process is said to be continued until the actual output is gained by the neural network.

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)
```

Output:

```
Epoch 0, Loss:0.27132035388864456
Epoch 4000, Loss:0.12865253014284214
Epoch 8000, Loss:0.006592119352308818
Predictions after training:
[[0.03837966]
 [0.93898139]
 [0.94188724]
 [0.07318271]]
```

- The output shows the training progress of a neural network over 10,000 epochs. Initially the loss was high (0.2713) but it gradually decreased as the network learned reaching a low value of 0.0066 by epoch 8000.
- The final predictions are close to the expected XOR outputs: approximately 0 for [0, 0] and [1, 1] and approximately 1 for [0, 1] and [1, 0] indicating that the network successfully learned to approximate the XOR function.

Advantages of Back Propagation for Neural Network Training

The key benefits of using the Back Propagation algorithm are:

1. **Ease of Implementation:** Back Propagation is beginner-friendly requiring no prior neural network knowledge and simplifies programming by adjusting weights with error derivatives.
2. **Simplicity and Flexibility:** Its straightforward design suits a range of tasks from basic feedforward to complex convolutional or recurrent networks.
3. **Efficiency**: Back Propagation accelerates learning by directly updating weights based on error especially in deep networks.
4. **Generalization:** It helps models generalize well to new data improving prediction accuracy on unseen examples.
5. **Scalability:** The algorithm scales efficiently with larger datasets and more complex networks making it ideal for large-scale tasks.

Challenges with Back Propagation

While Back Propagation is useful it does face some challenges:

1. **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.
2. **Exploding Gradients**: The gradients can also become excessively large causing the network to diverge during training.

3. **Overfitting:** If the network is too complex it might memorize the training data instead of learning general patterns.

Backward Propagation in Neural Networks:

Visit Course

Comment

More info

Explore

Advanced Techniques

Machine Learning Practice

GeeksforGeeks
Sanchhaya Education Private Limited

Corporate & Communications Address:
A-143, 7th Floor, Sovereign Corporate Tower, Sector- 136, Noida, Uttar Pradesh (201305)

Registered Address:
K 061, Tower K, Gulshan Vivante Apartment, Sector 137, Noida, Gautam Buddh Nagar, Uttar Pradesh, 201305

GET IT ON Google Play
Download on the App Store

Company
About Us
Legal
Privacy Policy
Contact Us
Advertise with us
GFG Corporate Solution
Campus Training Program

Explore
POTD
Job-A-Thon
Community
Blogs
Nation Skill Up

Tutorials
Programming Languages
DSA
Web Technology
AI, ML & Data Science
DevOps
CS Core Subjects
Interview Preparation
GATE
Software and Tools

Courses
IBM Certification
DSA and Placements
Web Development
Programming Languages
DevOps & Cloud
GATE
Trending Technologies

Videos
DSA
Python
Java

Preparation Corner
Aptitude
Puzzles
GfG 160

C++
Web Development
Data Science
CS Subjects

DSA 360
System Design

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved