Next Article:
Backpropagation in Convolution…

Backpropagation in Neural Network

Last Updated : 07 Mar, 2025

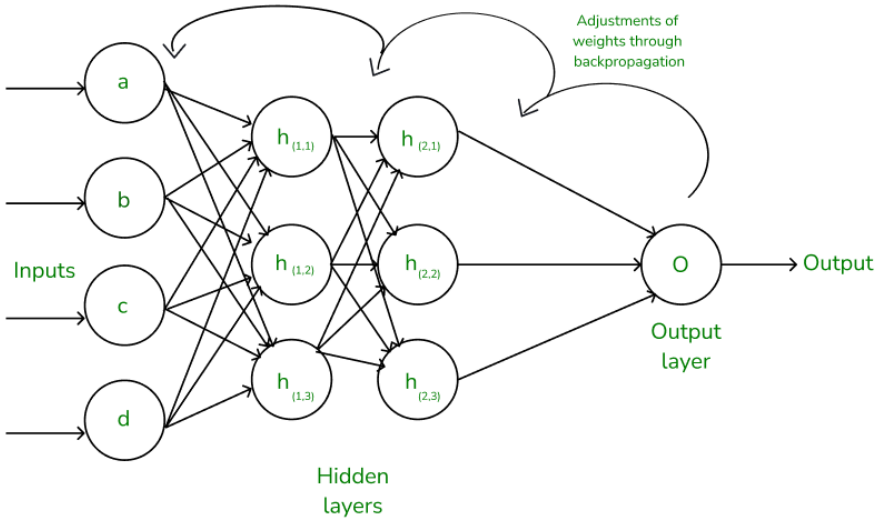
Backpropagation is also known as "*Backward Propagation of Errors*" and it is a method used to train **neural network** . Its goal is to reduce the difference between the model's predicted output and the actual output by adjusting the weights and biases in the network.

In this article we will explore what backpropagation is, why it is crucial in machine learning and how it works.

What is Backpropagation?

Backpropagation is a technique used in deep learning to train artificial neural networks particularly **feed-forward networks**. It works iteratively to adjust weights and bias to minimize the cost function.

Backpropagation 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

Backpropagation 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

2. **Scalability**: The backpropagation algorithm scales well to networks with multiple layers and complex architectures making deep learning feasible.
3. **Automated Learning**: With backpropagation the learning process becomes automated and the model can adjust itself to optimize its performance.

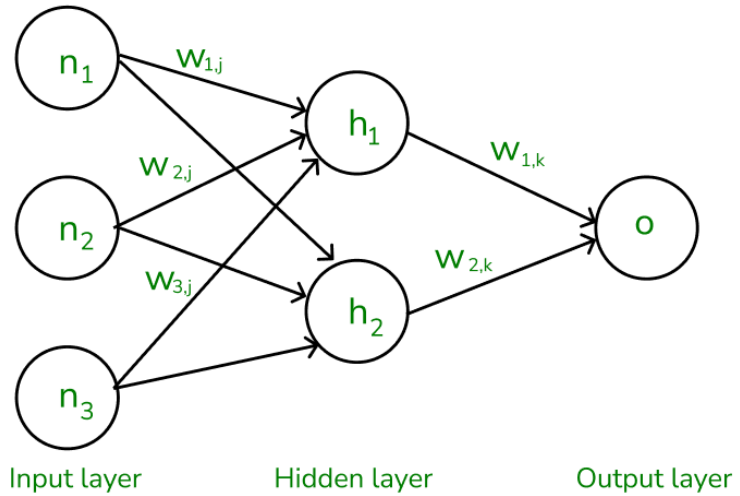
Working of Backpropagation Algorithm

The Backpropagation algorithm involves two main steps: the **Forward Pass** and the **Backward Pass**.

How Does 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 as shown in Fig. (a)) the output from h1 serves as the input to h2. Before applying an activation function, a bias is added to the weighted inputs.

linearity allowing the model to learn complex relationships in the data. Finally the outputs from the last hidden layer are passed to the output layer where an activation function such as **softmax** converts the weighted outputs into probabilities for classification.



The forward pass using weights and biases

How Does the Backward Pass Work?

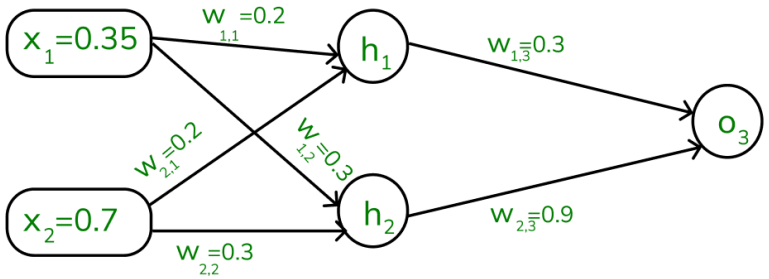
In the backward pass the error (the difference between the predicted and actual output) is propagated back through the network to adjust the

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

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 backpropagation.

Example of Backpropagation in Machine Learning

Let's walk through an example of backpropagation 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

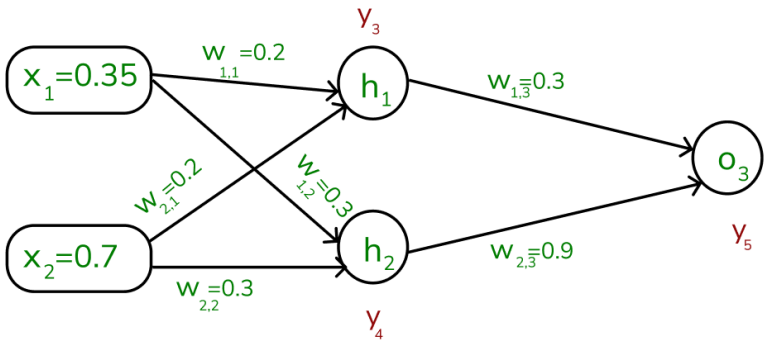
The weighted sum at each node is calculated using:

$$a_j = \sum(w_i, j * x_i)$$

Where,

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

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



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:

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

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

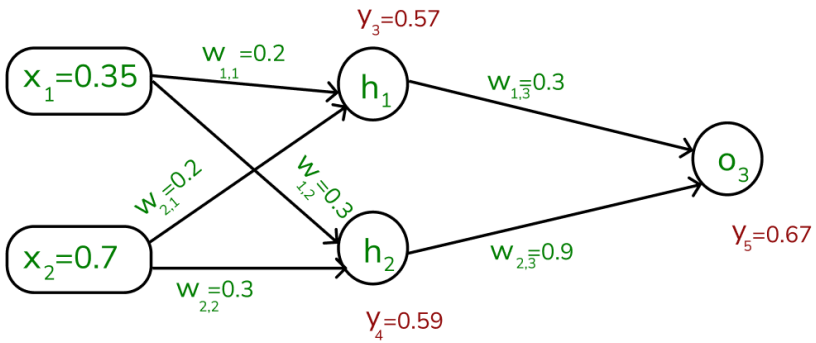
$$y_3 = 0.56$$

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

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

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

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



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:

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

$$Error = 0.5 - 0.67 = -0.17$$

Backpropagation

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:

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

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

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:

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:

$$w_{2,3}(\text{new}) = -0.22184 + 0.9 = 0.67816$$

For weights from input to hidden layer:

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

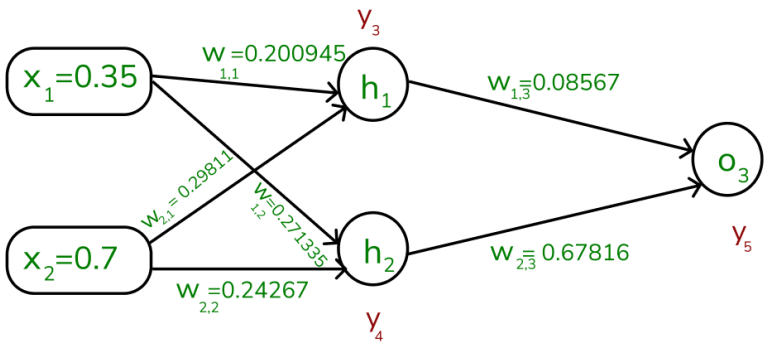
New weight:

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

Similarly other weights are updated:

- $w_{1,2}(\text{new}) = 0.271335$
- $w_{1,3}(\text{new}) = 0.08567$
- $w_{2,1}(\text{new}) = 0.29811$
- $w_{2,2}(\text{new}) = 0.24267$

The updated weights are illustrated below



Through backward pass the weights are updated

After updating the weights the forward pass is repeated yielding:

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

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 backpropagation iteratively updates weights by minimizing errors until the network accurately predicts the output.

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

Backpropagation Implementation in Python for XOR Problem

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

1. Defining Neural Network

- **Input layer** with 2 inputs
- **Hidden layer** with 4 neurons
- **Output layer** with 1 output neuron
- Using Sigmoid function as activation function

```
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.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)
```

- **`def __init__(self, input_size, hidden_size, output_size):`**: constructor to initialize the neural network
- **`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

- **`self.bias_output = np.zeros((1, self.output_size))`**: initializes bias for output layer

2. Defining Feed Forward Network

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

```
    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
```

- **`self.hidden_activation = np.dot(X, self.weights_input_hidden) + self.bias_hidden`**: calculates activation for 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

3. Defining Backward Network

In Backward pass (Backpropagation) 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.

```
    def backward(self, X, y, learning_
        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_input_hidden += np.dot(X.T,
hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta,
axis=0, keepdims=True) * learning_rate
```

- **`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

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

```
    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}")
```

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

5. Testing Neural Network

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

```
print("Predictions after training:")
print(output)
```

- `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

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 Backpropagation for Neural Network Training

The key benefits of using the backpropagation algorithm are:

- **Ease of Implementation:** Backpropagation is beginner-friendly requiring no prior neural network knowledge and simplifies programming by adjusting weights with error derivatives.
- **Simplicity and Flexibility:** Its straightforward design suits a range of tasks from basic feedforward to complex convolutional or recurrent networks.

especially in deep networks.

- **Generalization:** It helps models generalize well to new data improving prediction accuracy on unseen examples.
- **Scalability:** The algorithm scales efficiently with larger datasets and more complex networks making it ideal for large-scale tasks.

Challenges with Backpropagation

While backpropagation is powerful it does face some challenges:

1. **Vanishing Gradient Problem**: In deep networks the gradients can become very small during backpropagation 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

Backpropagation is a technique that makes neural network learn. By propagating errors backward and adjusting the weights and biases neural networks can gradually improve their predictions. Though it has some limitations like vanishing gradients many techniques like ReLU activation or optimizing learning rates have been developed to address these issues.

Next Article

Backpropagation in Convolutional Neural Networks

Similar Reads

Backpropagation in Neural Network

Backpropagation is also known as "Backward Propagation of Errors" and it is a method used to train neural network . Its goal is to reduce the difference...

10 min read

Backpropagation in Convolutional Neural Networks

Convolutional Neural Networks (CNNs) have become the backbone of many modern image processing systems. Their ability to learn hierarchical...

4 min read

Dropout in Neural Networks

The concept of Neural Networks is inspired by the neurons in the human brain and scientists wanted a machine to replicate the same process. This...

3 min read

Applications of Neural Network

Activation functions in Neural Networks

While building a neural network, one key decision is selecting the Activation Function for both the hidden layer and the output layer. Activation functions...

8 min read

Batch Size in Neural Network

Batch size is a hyperparameter that determines the number of training records used in one forward and backward pass of the neural network. In th...

5 min read

Effect of Bias in Neural Network

Neural Network is conceptually based on actual neuron of brain. Neurons are the basic units of a large neural network. A single neuron passes single...

3 min read

Backpropagation in Data Mining

Backpropagation is an algorithm that backpropagates the errors from the output nodes to the input nodes. Therefore, it is simply referred to as the...

6 min read

Back Propagation through time - RNN

Introduction: Recurrent Neural Networks are those networks that deal with sequential data. They predict outputs using not only the current inputs but...

5 min read

A feedback system in neural networks is a mechanism where the output is fed back into the network to influence subsequent outputs, often used to...

6 min read

Software Engineering
Digital Logic Design
Engineering Maths
Software Development
Software Testing

System Design
Bootcamp
Interview Questions

Company-Wise Preparation
Aptitude Preparation
Puzzles