Backpropagation in Neural Network

Last Updated : 02 Nov, 2024

Backpropagation **(**short for "***Backward Propagation of Errors***") is a method used to train artificial neural networks. 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.

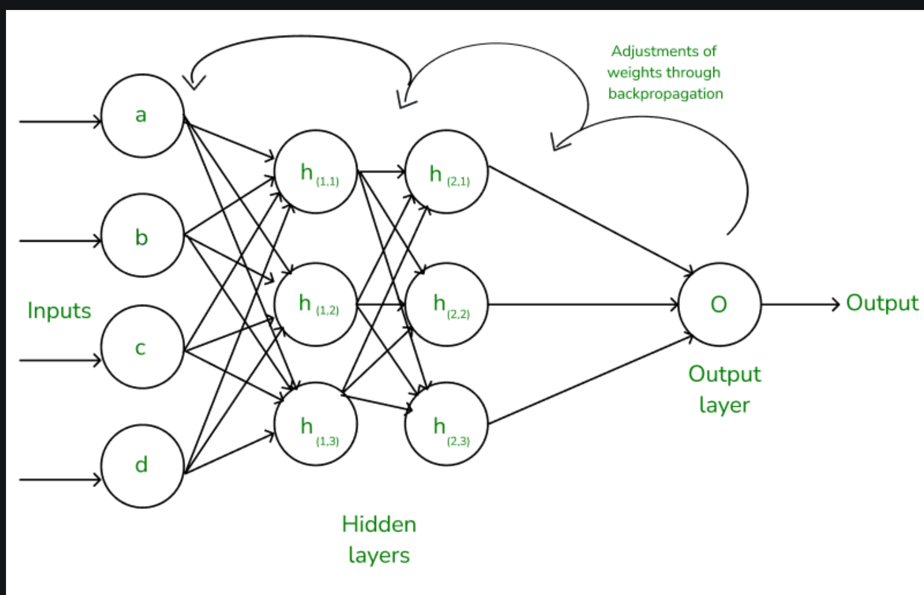
In this article, we will explore what backpropagation is, why it is crucial in machine learning, and how it works.

Table of Content

A **neural network** is a structured system composed of computing units called neurons, which enable it to compute functions. These neurons are interconnected through edges and assigned an **activation function**, along with adjustable parameters. These parameters allow the neural network to compute specific functions. Regarding activation functions, higher activation values indicate greater neuron activation in the network

Backpropagation is a powerful algorithm in deep learning, primarily used to train artificial neural networks, particularly feed-forward networks. It works iteratively, minimizing the cost function by adjusting weights and biases.

In each epoch, the model adapts these parameters, reducing loss by following the error gradient. Backpropagation often utilizes 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

Why is Backpropagation Important?

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 using the chain rule, making it possible to update weights efficiently.
2. **Scalability**: The backpropagation algorithm scales well to networks

3. **Automated Learning**: With backpropagation, the learning process becomes automated, and the model can adjust itself to optimize its performance.

Working of Backpropagation Algorithm

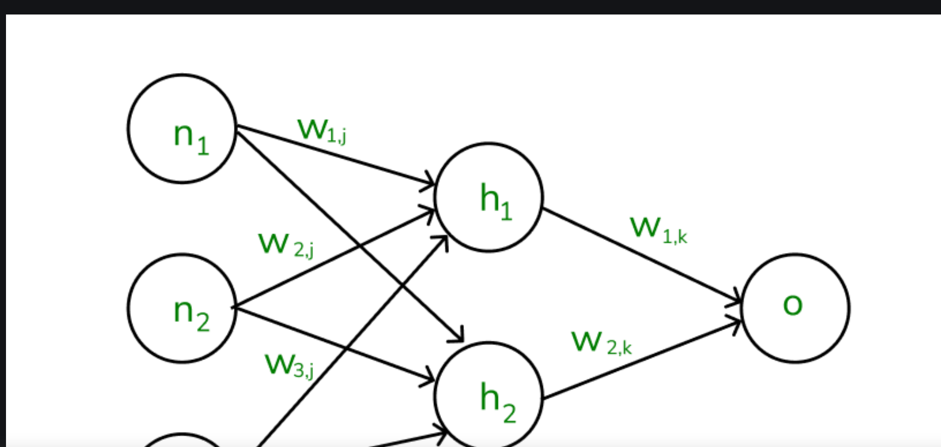
The Backpropagation algorithm involves two main steps: the **Forward Pass** and the **Backward Pass**.

How Does the Forward Pass Work?

In the **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.

Each hidden layer applies an activation function like [ReLU (Rectified Linear Unit)](), which returns the input if it's positive and zero otherwise. This adds non-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?

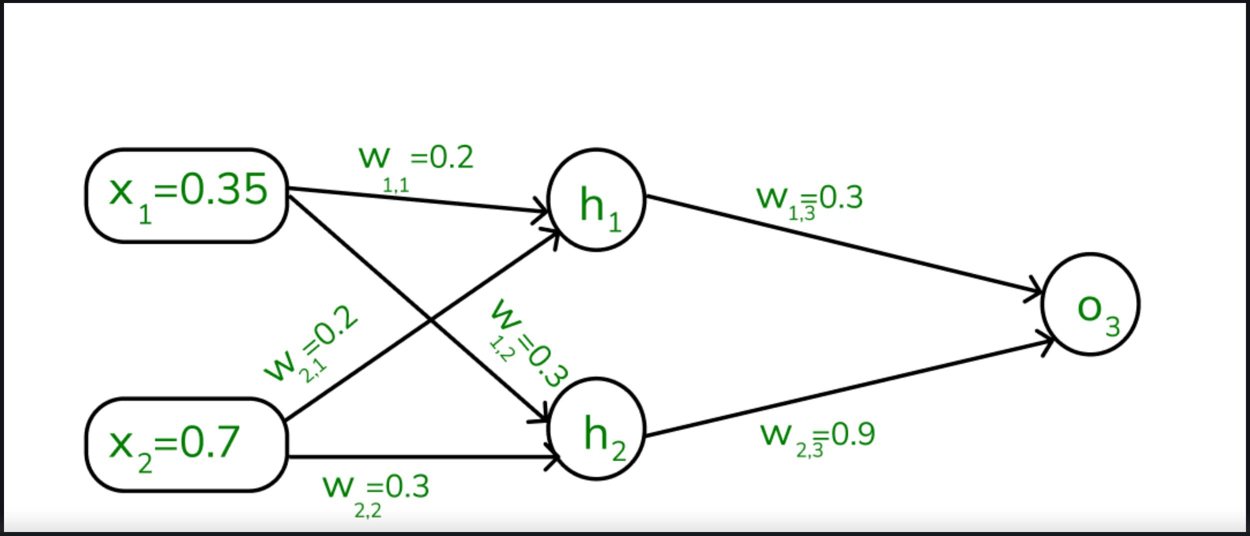
In the backward pass, the error (the difference between the predicted and actual output) is propagated back through the network to adjust the weights and biases. One common method for error calculation is the Mean Squared Error (MSE), given by:

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

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.



Forward Propagation

1. Initial Calculation

The weighted sum at each node is calculated using:

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

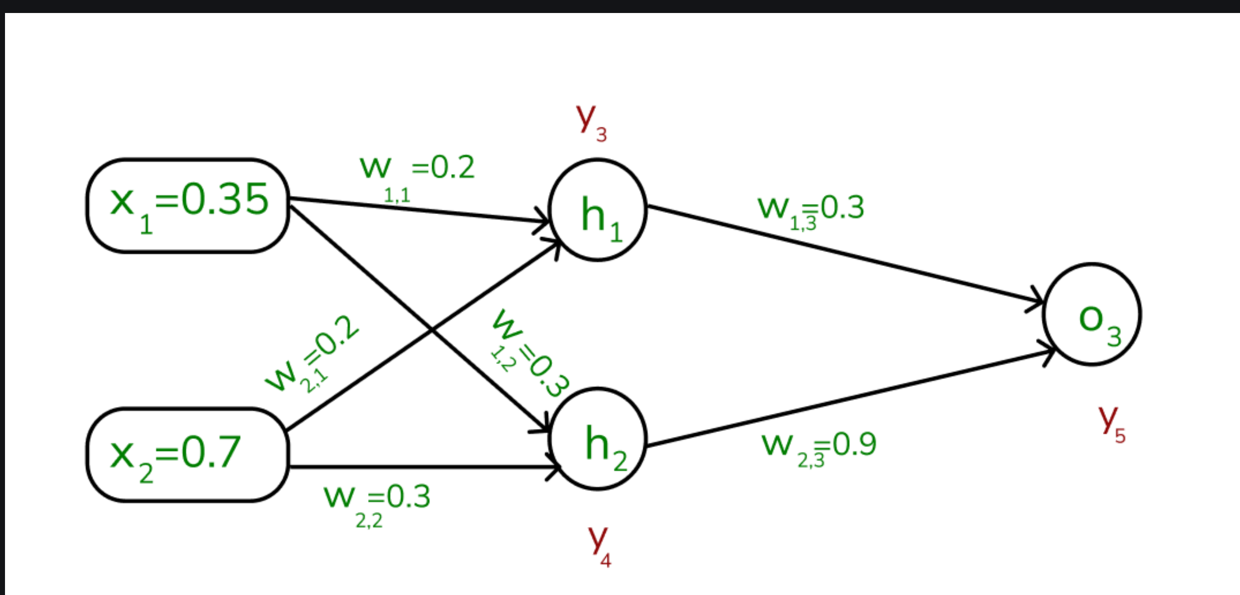
Where,

- 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,

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,

$$a_1 = (w_{1,1}x_1) + (w_{2,1}x_2)$$

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

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

$$y_3 = 0.56$$

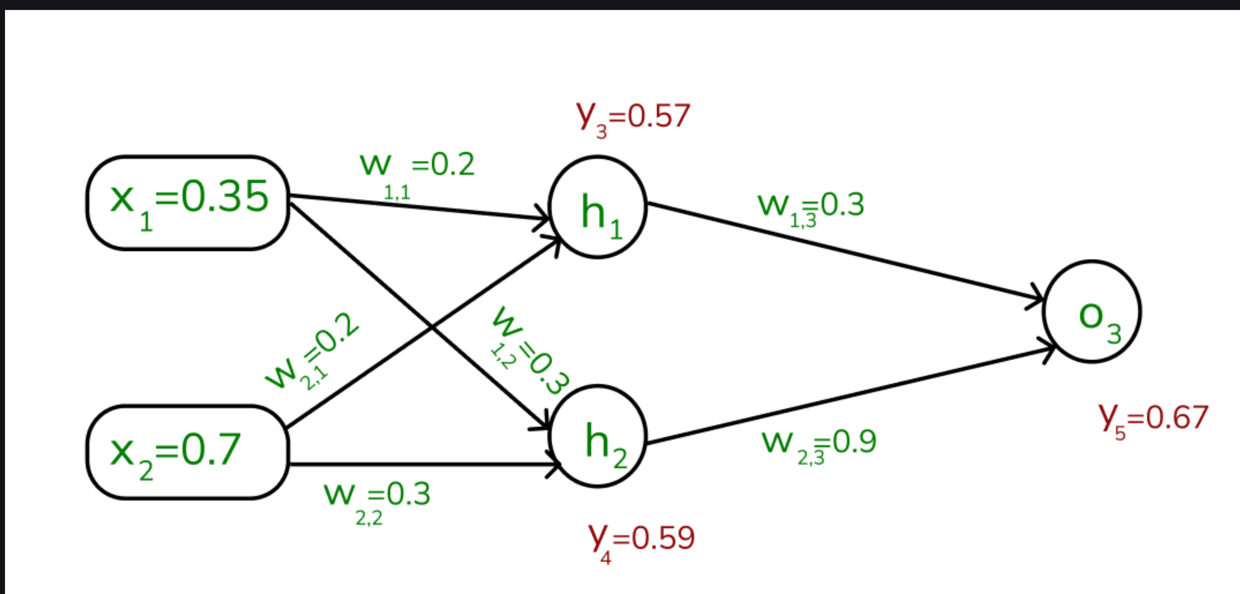
Similarly find the values of y_4 at **h_2** and y_5 at O_3 ,

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

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

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

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



Values of y3, y4 and y5

4. Error Calculation

Note that, our actual output is 0.5 but we obtained 0.67.

To calculate the error, we can use the below formula:

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

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

Using this error value, we will be backpropagating.

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:

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

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

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:

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

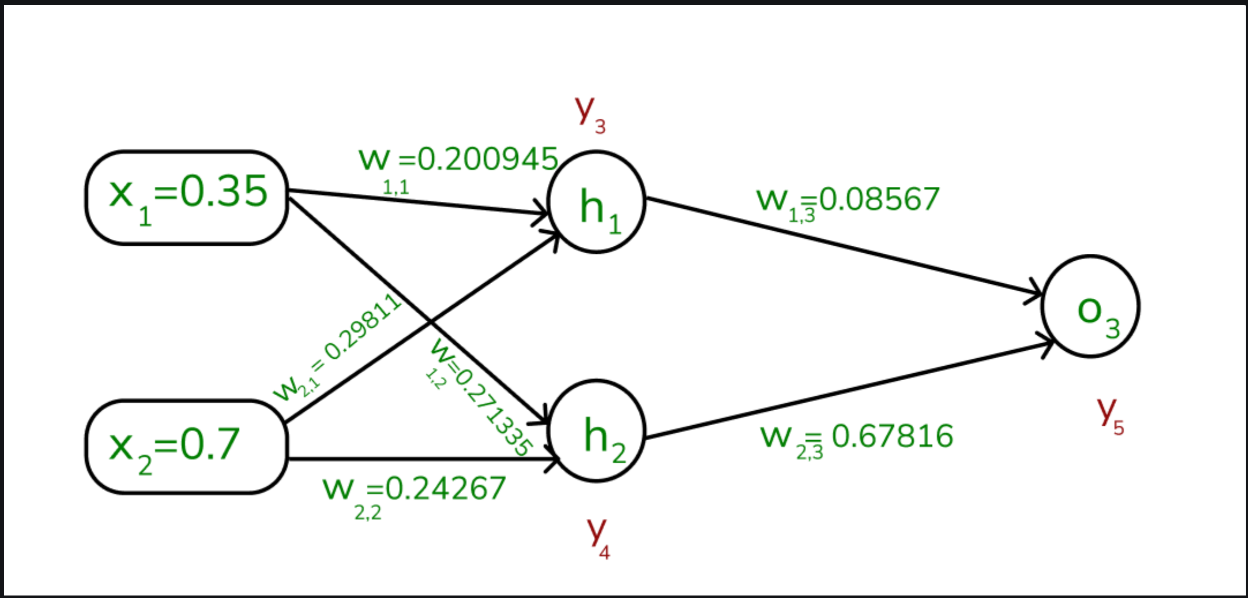
For weights from input to hidden layer:

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

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

Final Forward Pass:

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

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

$$= 0.5 - 0.61 = -0.11$$

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

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:

- **Input layer** with 2 inputs,
- **Hidden layer** with 4 neurons,
- **Output layer** with 1 output neuron.

Key steps:

1. **Forward pass:** The inputs are passed through the network, activating the hidden and output layers using the sigmoid function.
2. **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.
3. **Training:** The network is trained over 10,000 epochs using the backpropagation algorithm with a learning rate of 0.1, progressively reducing the error.

This implementation highlights how backpropagation adjusts weights and biases to minimize the loss and improve predictions over time.

```
import numpy as np

class NeuralNetwork:
```

```
        # Initialize weights
        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)

        # Initialize the biases
        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 feedforward(self, X):
        # Input to hidden
        self.hidden_activation = np.dot(X, self.weights_input_hidden)
+ self.bias_hidden
        self.hidden_output = self.sigmoid(self.hidden_activation)

        # Hidden to output
        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

    def backward(self, X, y, learning_rate):
        # Compute the output layer error
        output_error = y - self.predicted_output
        output_delta = output_error *
self.sigmoid_derivative(self.predicted_output)

        # Compute the hidden layer error
        hidden_error = np.dot(output_delta,
self.weights_hidden_output.T)
        hidden_delta = hidden_error *
self.sigmoid_derivative(self.hidden_output)

        # Update weights and biases
        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
```

```
            if epoch % 4000 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss:{loss}")

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)

# Test the trained model
output = nn.feedforward(X)
print("Predictions after training:")
print(output)
```

Output:

```
Epoch 0, Loss:0.2680427627058641З
Epoch 4000, Loss:0.012477301332301533
Epoch 8000, Loss:0.0029801470220045504

Predictions after training:
[[0.02330965]
 [0.95658721]
 [0.95049451]
 [0.05896647]]
```

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 via error derivatives.
- **Simplicity and Flexibility:** Its straightforward design suits a range of tasks, from basic feedforward to complex convolutional or recurrent networks.
- **Efficiency:** Backpropagation accelerates learning by directly updating

- **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.

Conclusion

Backpropagation is the engine that drives neural network learning. 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, such as using ReLU activation or optimizing learning rates, have been developed to address these issues.

Comment More info Advertise with us

Next Article

Backpropagation in Convolutional Neural Networks

Similar Reads

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

How Does Gradient Descent and Backpropagation Work Together?

How to Update Bias and Bias's Weight Using Backpropagation…

Answer: In backpropagation, biases are updated by applying the chain rule to the loss function with respect to the bias parameters in each layer during...

3 min read

Computing Gradients with Backpropagation for Arbitrary Loss and…

Backpropagation allows a network to learn from its mistakes by adjusting the weights based on errors. It computes gradients efficiently using the chai...

4 min read

A single neuron neural network in Python

Neural networks are the core of deep learning, a field that has practical applications in many different areas. Today neural networks are used for...

3 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

Importance of Convolutional Neural Network | ML

Convolutional Neural Network as the name suggests is a neural network that makes use of convolution operation to classify and predict. Let's analyz...

2 min read

Neural Network Advances

We know that our world is changing quickly but there are lot of concrete technology advances that you might not hear a lot about in the newspaper ...

3 min read

For loop take much time for completing iterations and in ML practise we have to optimize the time so we can use for loops. But then you must be...

2 min read

Implementation of Artificial Neural Network for AND Logic Gate with 2...

Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three...

4 min read

Implementation of Artificial Neural Network for OR Logic Gate with 2-...

Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three...

4 min read

Implementation of Artificial Neural Network for NAND Logic Gate with ...

Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three...

4 min read

Implementation of Artificial Neural Network for NOR Logic Gate with 2...

Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three...

4 min read

Implementation of Artificial Neural Network for XOR Logic Gate with 2-...

Artificial Neural Network (ANN) is a computational model based on the biological neural networks of animal brains. ANN is modeled with three...

4 min read

Implementation of Artificial Neural Network for XNOR Logic Gate with ...

Difference between Neural Network And Fuzzy Logic

Neural Network: Neural network is an information processing system that is inspired by the way biological nervous systems such as brain process...

2 min read

ANN - Implementation of Self Organizing Neural Network (SONN) from...

Prerequisite: ANN | Self Organizing Neural Network (SONN) Learning Algorithm To implement a SONN, here are some essential consideration-...

4 min read

ANN - Self Organizing Neural Network (SONN)

Self Organizing Neural Network (SONN) is an unsupervised learning model in Artificial Neural Network termed as Self-Organizing Feature Maps or...

2 min read

ANN - Self Organizing Neural Network (SONN) Learning Algorithm

Prerequisite: ANN | Self Organizing Neural Network (SONN) In the Self Organizing Neural Network (SONN), learning is performed by shifting the...

3 min read

Adjusting Learning Rate of a Neural Network in PyTorch

Learning Rate is an important hyperparameter in Gradient Descent. Its value determines how fast the Neural Network would converge to minima. Usuall...

7 min read

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

Company
About Us
Legal
Privacy Policy
Careers
In Media
Contact Us
GFG Corporate Solution
Placement Training Program

Explore
Job-A-Thon Hiring Challenge
Hack-A-Thon
GfG Weekly Contest
Offline Classes (Delhi/NCR)
DSA in JAVA/C++
Master System Design
Master CP
GeeksforGeeks Videos
Geeks Community

Languages
Python
Java
C++
PHP
GoLang
SQL
R Language
Android Tutorial

DSA
Data Structures
Algorithms
DSA for Beginners
Basic DSA Problems
DSA Roadmap
DSA Interview Questions
Competitive Programming

Data Science & ML
Data Science With Python
Data Science For Beginner
Machine Learning
ML Maths
Data Visualisation
Pandas
NumPy

Web Technologies
HTML
CSS
JavaScript
TypeScript
ReactJS
NextJS
NodeJs

Python Programming Examples
Django Tutorial
Python Projects
Python Tkinter
Web Scraping
OpenCV Tutorial
Python Interview Question

DevOps

Git
AWS
Docker
Kubernetes
Azure
GCP
DevOps Roadmap

School Subjects

Mathematics
Physics
Chemistry
Biology
Social Science
English Grammar

Databases

SQL
MYSQL
PostgreSQL
PL/SQL
MongoDB

Competitive Exams

JEE Advanced
UGC NET
UPSC
SSC CGL
SBI PO

GATE CS Notes
Operating Systems
Computer Network
Database Management System
Software Engineering
Digital Logic Design
Engineering Maths

System Design

High Level Design
Low Level Design
UML Diagrams
Interview Guide
Design Patterns
OOAD
System Design Bootcamp
Interview Questions

Commerce

Accountancy
Business Studies
Economics
Management
HR Management
Finance
Income Tax

Preparation Corner

Company-Wise Recruitment Process
Resume Templates
Aptitude Preparation
Puzzles
Company-Wise Preparation
Companies
Colleges

More Tutorials

Software Development
Software Testing
Product Management
Project Management
Linux