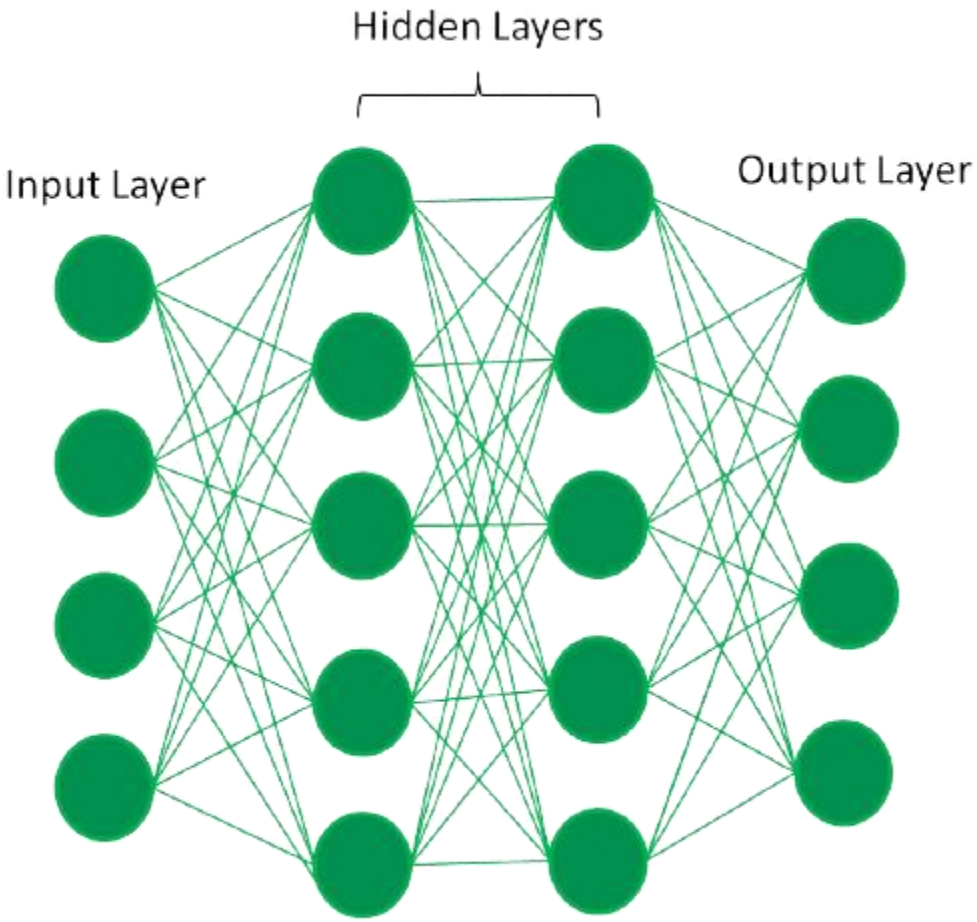
How Deep Learning Works?

Neural network consists of layers of interconnected nodes or neurons that collaborate to process input data. In a **fully connected deep neural network** data flows through multiple layers where each neuron performs nonlinear transformations, allowing the model to learn intricate representations of the data.

In a deep neural network the **input layer** receives data which passes through **hidden layers** that transform the data using nonlinear functions. The final **output layer** generates the model's prediction.

For more details on neural networks refer to this article: What is a Neural Network?



Fully Connected Deep Neural Network

Difference between Machine Learning and Deep Learning

Machine learning and Deep Learning both are subsets of artificial intelligence but there are many similarities and differences between them.

Machine Learning	Deep Learning
Apply statistical algorithms to learn the hidden patterns and relationships in the dataset.	Uses artificial neural network architecture to learn the hidden patterns and relationships in the dataset.
Can work on the smaller amount of dataset	Requires the larger volume of dataset compared to machine learning
Better for the low-label task.	Better for complex task like image processing, natural language processing, etc.
Takes less time to train the model.	Takes more time to train the model.

Machine Learning	Deep Learning
A model is created by relevant features which are manually extracted from images to detect an object in the image.	Relevant features are automatically extracted from images. It is an end-to-end learning process.
Less complex and easy to interpret the result.	More complex, it works like the black box interpretations of the result are not easy.
It can work on the CPU or requires less computing power as compared to deep learning.	It requires a high-performance computer with GPU.

Evolution of Neural Architectures

The journey of deep learning began with the **perceptron**, a single-layer neural network introduced in the 1950s. While innovative, perceptrons could only solve linearly separable problems hence failing at more complex tasks like the XOR problem.

This limitation led to the development of **Multi-Layer Perceptrons (MLPs)**. It introduced hidden layers and non-linear activation functions. MLPs trained using **backpropagation** could model complex, non-linear relationships marking a significant leap in neural network capabilities. This evolution from perceptrons to MLPs laid the groundwork for advanced architectures like CNNs and RNNs, showcasing the power of layered structures in solving real-world problems.

Types of neural networks

1. **Feedforward neural networks (FNNs)** are the simplest type of ANN, where data flows in one direction from input to output. It is used for basic tasks like classification.
2. **Convolutional Neural Networks (CNNs)** are specialized for processing grid-like data, such as images. CNNs use convolutional layers to detect spatial hierarchies, making them ideal for computer vision tasks.
3. **Recurrent Neural Networks (RNNs)** are able to process sequential data, such as time series and natural language. RNNs have loops to retain information over time, enabling applications like language modeling and speech recognition. Variants like LSTMs and GRUs address vanishing gradient issues.

4. **Generative Adversarial Networks (GANs)** consist of two networks—a generator and a discriminator—that compete to create realistic data. GANs are widely used for image generation, style transfer and data augmentation.
5. **Autoencoders** are unsupervised networks that learn efficient data encodings. They compress input data into a latent representation and reconstruct it, useful for dimensionality reduction and anomaly detection.
6. **Transformer Networks** has revolutionized NLP with self-attention mechanisms. Transformers excel at tasks like translation, text generation and sentiment analysis, powering models like GPT and BERT.

Deep Learning Applications

1. Computer vision

In computer vision, deep learning models enable machines to identify and understand visual data. Some of the main applications of deep learning in computer vision include:

- **Object detection and recognition:** Deep learning models are used to identify and locate objects within images and videos, making it possible for machines to perform tasks such as self-driving cars, surveillance and robotics.
- **Image classification:** Deep learning models can be used to classify images into categories such as animals, plants and buildings. This is used in applications such as medical imaging, quality control and image retrieval.
- **Image segmentation:** Deep learning models can be used for image segmentation into different regions, making it possible to identify specific features within images.

2. Natural language processing (NLP)

In NLP, deep learning model enable machines to understand and generate human language. Some of the main applications of deep learning in NLP include:

- **Automatic Text Generation:** Deep learning model can learn the corpus of text and new text like summaries, essays can be automatically generated using these trained models.
- **Language translation:** Deep learning models can translate text from one language to another, making it possible to communicate with people from different linguistic backgrounds.
- **Sentiment analysis:** Deep learning models can analyze the sentiment of a piece of text, making it possible to determine whether the text is positive, negative or neutral.

- **Speech recognition:** Deep learning models can recognize and transcribe spoken words, making it possible to perform tasks such as speech-to-text conversion, voice search and voice-controlled devices.

3. Reinforcement learning

In reinforcement learning, deep learning works as training agents to take action in an environment to maximize a reward. Some of the main applications of deep learning in reinforcement learning include:

- **Game playing:** Deep reinforcement learning models have been able to beat human experts at games such as Go, Chess and Atari.
- **Robotics:** Deep reinforcement learning models can be used to train robots to perform complex tasks such as grasping objects, navigation and manipulation.
- **Control systems:** Deep reinforcement learning models can be used to control complex systems such as power grids, traffic management and supply chain optimization.

Advantages of Deep Learning

1. **High accuracy:** Deep Learning algorithms can achieve state-of-the-art performance in various tasks such as image recognition and natural language processing.
2. **Automated feature engineering:** Deep Learning algorithms can automatically discover and learn relevant features from data without the need for manual feature engineering.
3. **Scalability:** Deep Learning models can scale to handle large and complex datasets and can learn from massive amounts of data.
4. **Flexibility:** Deep Learning models can be applied to a wide range of tasks and can handle various types of data such as images, text and speech.
5. **Continual improvement:** Deep Learning models can continually improve their performance as more data becomes available.

Disadvantages of Deep Learning

Deep learning has made significant advancements in various fields but there are still some challenges that need to be addressed. Here are some of the main challenges in deep learning:

1. **Data availability**: It requires large amounts of data to learn from. For using deep learning it's a big concern to gather as much data for training.

2. **Computational Resources**: For training the deep learning model, it is computationally expensive because it requires specialized hardware like GPUs and TPUs.
3. **Time-consuming:** While working on sequential data depending on the computational resource it can take very large even in days or months.
4. I**nterpretability:** Deep learning models are complex, it works like a black box. It is very difficult to interpret the result.
5. **Overfitting:** when the model is trained again and again it becomes too specialized for the training data leading to overfitting and poor performance on new data.

Feedforward Neural Network

Last Updated : 19 Mar, 2025

-
-
-

Feedforward Neural Network (FNN) is a type of artificial neural network in which information flows in a single direction—from the input layer through hidden layers to the output layer—without loops or feedback. It is mainly used for pattern recognition tasks like image and speech classification.

For example in a credit scoring system banks use an FNN which analyze users' financial profiles—such as income, credit history and spending habits—to determine their creditworthiness.

Each piece of information flows through the network's layers where various calculations are made to produce a final score.

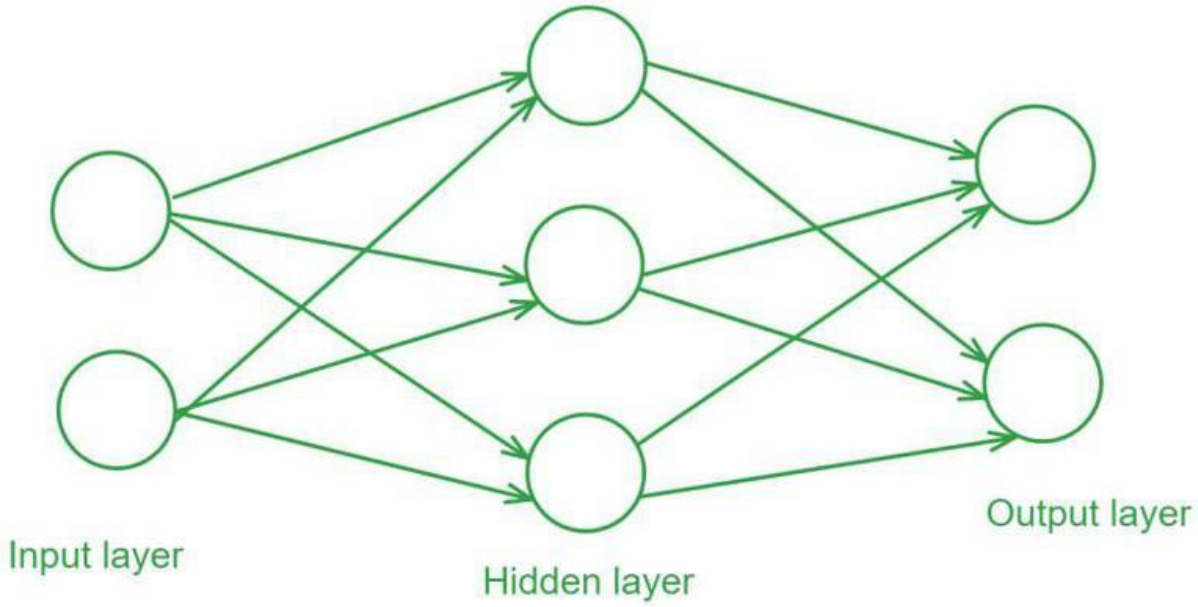
Structure of a Feedforward Neural Network

Feedforward Neural Networks have a structured layered design where data flows sequentially through each layer.

1. **Input Layer**: The input layer consists of neurons that receive the input data. Each neuron in the input layer represents a feature of the input data.
2. **Hidden Layers**: One or more hidden layers are placed between the input and output layers. These layers are responsible for learning the complex patterns in the data. Each neuron in a hidden layer applies a weighted sum of inputs followed by a non-linear activation function.

3. **Output Layer**: The output layer provides the final output of the network. The number of neurons in this layer corresponds to the number of classes in a classification problem or the number of outputs in a regression problem.

Each connection between neurons in these layers has an associated weight that is adjusted during the training process to minimize the error in predictions.



Feed Forward Neural Network

Activation Functions

Activation functions introduce non-linearity into the network enabling it to learn and model complex data patterns.

Common activation functions include:

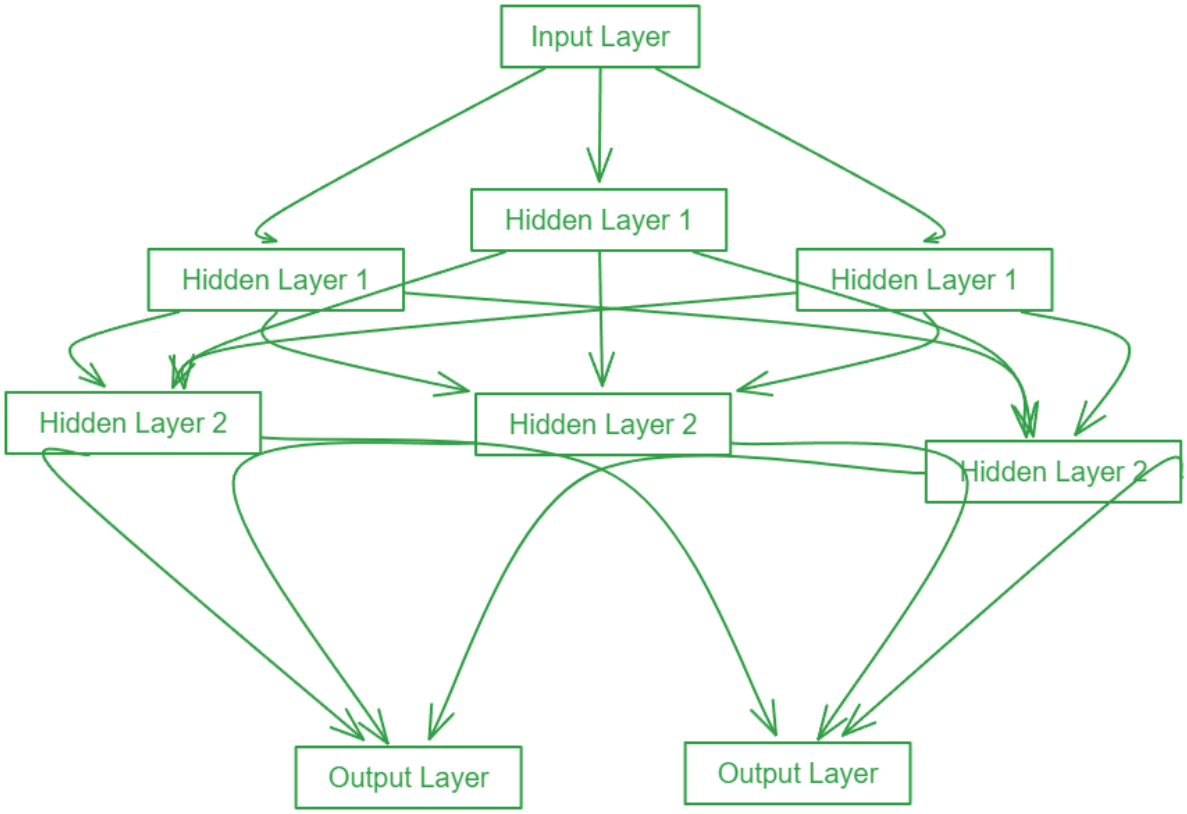
- **Sigmoid**: σ(x)=σ(x)=11+e−xσ(*x*)=1+*e*−*x*1
- **Tanh**: tanh(x)=ex−e−xex+e−xtanh(*x*)=*ex*+*e*−*xex*−*e*−*x*
- **ReLU**: ReLU(x)=max(0,x)ReLU(*x*)=max(0,*x*)

Training a Feedforward Neural Network

Training a Feedforward Neural Network involves adjusting the weights of the neurons to minimize the error between the predicted output and the actual output. This process is typically performed using backpropagation and gradient descent.

1. **Forward Propagation**: During forward propagation the input data passes through the network and the output is calculated.
2. **Loss Calculation**: The loss (or error) is calculated using a loss function such as Mean Squared Error (MSE) for regression tasks or Cross-Entropy Loss for classification tasks.

3. **Backpropagation**: In backpropagation the error is propagated back through the network to update the weights. The gradient of the loss function with respect to each weight is calculated and the weights are adjusted using gradient descent.



Forward Propagation

Gradient Descent

Gradient Descent is an optimization algorithm used to minimize the loss function by iteratively updating the weights in the direction of the negative gradient. Common variants of gradient descent include:

- **Batch Gradient Descent**: Updates weights after computing the gradient over the entire dataset.
- **Stochastic Gradient Descent (SGD)**: Updates weights for each training example individually.
- **Mini-batch Gradient Descent**: It Updates weights after computing the gradient over a small batch of training examples.

Evaluation of Feedforward neural network

Evaluating the performance of the trained model involves several metrics:

- **Accuracy**: The proportion of correctly classified instances out of the total instances.
- **Precision**: The ratio of true positive predictions to the total predicted positives.

- **Recall**: The ratio of true positive predictions to the actual positives.
- **F1 Score**: The harmonic mean of precision and recall, providing a balance between the two.
- **Confusion Matrix:** A table used to describe the performance of a classification model, showing the true positives, true negatives, false positives, and false negatives.

Code Implementation of Feedforward neural network

This code demonstrates the process of building, training and evaluating a neural network model using TensorFlow and Keras to classify handwritten digits from the MNIST dataset.

The model architecture is defined using the **Sequential API** consisting of:

- a Flatten layer to convert the 2D image input into a 1D array
- a Dense layer with 128 neurons and ReLU activation
- a final Dense layer with 10 neurons and softmax activation to output probabilities for each digit class.

Model is compiled with the Adam optimizer, SparseCategoricalCrossentropy loss function and SparseCategoricalAccuracy metric and then trained for 5 epochs on the training data.

Backpropagation in Neural Network

Last Updated : 01 Jul, 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:

MSE=(Predicted Output−Actual Output)2MSE=(Predicted Output−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:

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

Where,

- aj*aj* is the weighted sum of all the inputs and weights at each node
- wi,j*wi*,*j* represents the weights between the ith*ith*input and the jth*jth* neuron
- xi*xi* represents the value of the ith*ith* input

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

ojoj = activation function(ajaj)

2. Sigmoid Function

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

$yj=11+e-ajyj=1+e-aj1$

To find the outputs of y3, y4 and y5

3. Computing Outputs

At h1 node

a1=(w1,1x1)+(w2,1x2)=(0.2∗0.35)+(0.2∗0.7)=0.21$a1=(w1,1x1)+(w2,1x2)=(0.2∗0.35)+(0.2∗0.7)=0.21$

Once we calculated the a1 value, we can now proceed to find the y3 value:

yj=F(aj)=11+e−a1$yj=F(aj)=1+e−a11$

y3=F(0.21)=11+e−0.21$y3=F(0.21)=1+e−0.211$

y3=0.56$y3=0.56$

Similarly find the values of y4 at **h2** and y5 at O3

a2=(w1,2∗x1)+(w2,2∗x2)=(0.3∗0.35)+(0.3∗0.7)=0.315$a2=(w1,2∗x1)+(w2,2∗x2)=(0.3∗0.35)+(0.3∗0.7)=0.315$

y4=F(0.315)=11+e−0.315$y4=F(0.315)=1+e−0.3151$

a3=(w1,3∗y3)+(w2,3∗y4)=(0.3∗0.57)+(0.9∗0.59)=0.702$a3=(w1,3∗y3)+(w2,3∗y4)=(0.3∗0.57)+(0.9∗0.59)=0.702$

y5=F(0.702)=11+e−0.702=0.67$y5=F(0.702)=1+e−0.7021=0.67$

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:

$Errorj=ytarget−y5Errorj=ytarget−y5$

=> $0.5−0.67=−0.17$=> $0.5−0.67=−0.17$

Using this error value we will be backpropagating.

Back Propagation

1. Calculating Gradients

The change in each weight is calculated as:

$Δwij=η×δj×OjΔwij=η×δj×Oj$

Where:

- $δj$$\delta j$ is the error term for each unit,
- $η$$\eta$ is the learning rate.

2. Output Unit Error

For O3:

δ5=y5(1−y5)(ytarget−y5)$\delta 5 = y5(1-y5)(ytarget-y5)$

=0.67(1−0.67)(−0.17)=−0.0376=0.67(1−0.67)(−0.17)=−0.0376

3. Hidden Unit Error

For h1:

δ3=y3(1−y3)(w1,3×δ5)$\delta 3 = y3(1-y3)(w1,3 \times \delta 5)$

=0.56(1−0.56)(0.3×−0.0376)=−0.0027=0.56(1−0.56)(0.3×−0.0376)=−0.0027

For h2:

δ4=y4(1−y4)(w2,3×δ5)$\delta 4 = y4(1-y4)(w2,3 \times \delta 5)$

=0.59(1−0.59)(0.9×−0.0376)=−0.0819=0.59(1−0.59)(0.9×−0.0376)=−0.0819

4. Weight Updates

For the weights from hidden to output layer:

Δw2,3=1×(−0.0376)×0.59=−0.022184$\Delta w2,3 = 1 \times (-0.0376) \times 0.59 = -0.022184$

New weight:

w2,3(new)=−0.022184+0.9=0.877816$w2,3(new) = -0.022184 + 0.9 = 0.877816$

For weights from input to hidden layer:

Δw1,1=1×(−0.0027)×0.35=0.000945$\Delta w1,1 = 1 \times (-0.0027) \times 0.35 = 0.000945$

New weight:

w1,1(new)=0.000945+0.2=0.200945$w1,1(new) = 0.000945 + 0.2 = 0.200945$

Similarly other weights are updated:

- w1,2(new)=0.273225$w1,2(new) = 0.273225$
- w1,3(new)=0.086615$w1,3(new) = 0.086615$
- w2,1(new)=0.269445$w2,1(new) = 0.269445$
- w2,2(new)=0.18534$w2,2(new) = 0.18534$

The updated weights are illustrated below

Through backward pass the weights are updated

After updating the weights the forward pass is repeated yielding:

- y3=0.57$y3$=0.57
- y4=0.56$y4$=0.56
- y5=0.61$y5$=0.61

Since y5=0.61$y5$=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.

$Error=ytarget−y5Error=ytarget−y5$

=0.5−0.61=−0.11=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

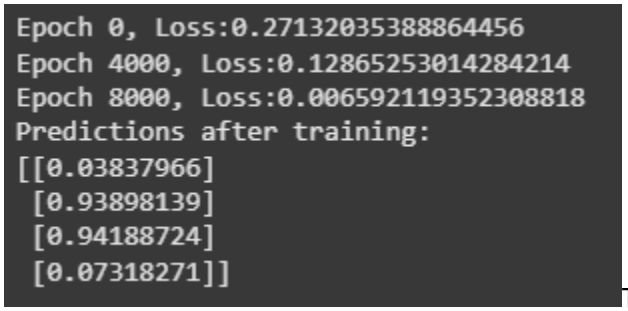
- **X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]]):** defines the input data
- **y = np.array([[0], [1], [1], [0]]):** defines the target values
- **nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1):** initializes the neural network
- **nn.train(X, y, epochs=10000, learning_rate=0.1):**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



Trained Model

- 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.