Neural Networks from Scratch

Contents

[Introduction to Neural Networks for Beginners 2](#_Toc172821283)

[1. What is a Neural Network? 2](#_Toc172821284)

[2. Basic Structure of a Neural Network 2](#_Toc172821285)

[ Input Layer: 2](#_Toc172821286)

[ Hidden Layers: 2](#_Toc172821287)

[ Output Layer: 2](#_Toc172821288)

[3. Neurons and Activation Functions 2](#_Toc172821289)

[ Sigmoid: 2](#_Toc172821290)

[ ReLU (Rectified Linear Unit): 3](#_Toc172821291)

[ Tanh: 3](#_Toc172821292)

[4. Training a Neural Network 3](#_Toc172821293)

[ Forward Propagation: 3](#_Toc172821294)

[ Loss Function: 3](#_Toc172821295)

[ Backpropagation: 3](#_Toc172821296)

[5. Types of Neural Networks 3](#_Toc172821297)

[ Feedforward Neural Networks (FNN): 3](#_Toc172821298)

[ Convolutional Neural Networks (CNN): 3](#_Toc172821299)

[ Recurrent Neural Networks (RNN): 3](#_Toc172821300)

[6. Applications of Neural Networks 3](#_Toc172821301)

[ Computer Vision 3](#_Toc172821302)

[ Natural Language Processing (NLP): 3](#_Toc172821303)

[ Healthcare: 3](#_Toc172821304)

[ Finance: 3](#_Toc172821305)

[ Autonomous Systems: 3](#_Toc172821306)

[Conclusion 4](#_Toc172821307)

# Introduction to Neural Networks for Beginners

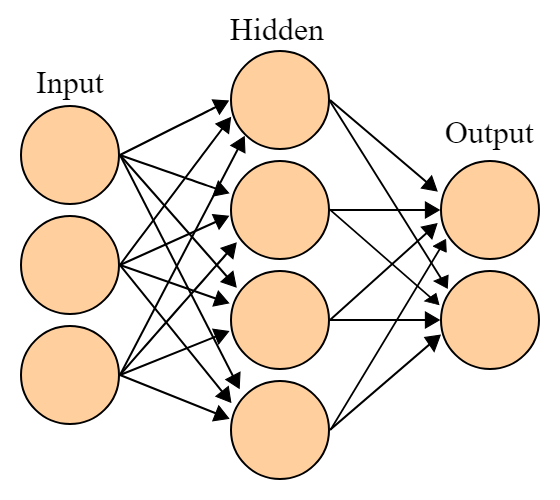
Neural networks are a foundational concept in the field of artificial intelligence and machine learning. They are inspired by the structure and function of the human brain, aiming to enable machines to learn from data and make intelligent decisions.

## 1. What is a Neural Network?

A neural network is a computational model that consists of layers of interconnected nodes, or "neurons." Each neuron processes input data and passes the result to the next layer. Neural networks can recognize patterns, classify data, and make predictions.

## 2. Basic Structure of a Neural Network

* Input Layer: The input layer receives the initial data. Each neuron in this layer represents a feature or attribute of the data.
* Hidden Layers: Between the input and output layers are one or more hidden layers. These layers perform complex computations on the inputs received from the previous layer.
* Output Layer: The output layer produces the final result, such as a classification label or a predicted value.



## 3. Neurons and Activation Functions

Each neuron performs a weighted sum of its inputs, adds a bias term, and then applies an activation function. Activation functions introduce non-linearity into the network, enabling it to learn complex patterns. Common activation functions include:

* Sigmoid: σ(x)=11+e−x\sigma(x) = \frac{1}{1 + e^{-x}}σ(x)=1+e−x1​
* ReLU (Rectified Linear Unit): ReLU(x)=max⁡(0,x)\text{ReLU}(x) = \max(0, x)ReLU(x)=max(0,x)
* Tanh: tanh(x)=ex−e−xex+e−x\text{tanh}(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}tanh(x)=ex+e−xex−e−x​

## 4. Training a Neural Network

Training a neural network involves adjusting the weights and biases to minimize the difference between the predicted output and the actual output. This process is called optimization and typically uses a method called **backpropagation** in conjunction with an optimization algorithm like **gradient descent**.

* Forward Propagation: Pass the input data through the network to get the output.
* Loss Function: Calculate the error (loss) between the predicted output and the actual output. Common loss functions include Mean Squared Error (MSE) for regression tasks and Cross-Entropy Loss for classification tasks.
* Backpropagation: Compute the gradient of the loss function with respect to each weight using the chain rule and adjust the weights to reduce the loss.

## 5. Types of Neural Networks

* Feedforward Neural Networks (FNN): The simplest type of neural network, where connections do not form cycles. Data moves in one direction, from input to output.
* Convolutional Neural Networks (CNN): Primarily used for image recognition tasks. They use convolutional layers to automatically and adaptively learn spatial hierarchies of features from input images.
* Recurrent Neural Networks (RNN): Designed for sequential data, such as time series or natural language. They have connections that form cycles, allowing them to maintain memory of previous inputs.

## 6. Applications of Neural Networks

Neural networks have a wide range of applications across various fields, including:

* Computer Vision: Image and video recognition, object detection, facial recognition.
* Natural Language Processing (NLP): Language translation, sentiment analysis, speech recognition.
* Healthcare: Disease prediction, medical image analysis.
* Finance: Stock price prediction, fraud detection.
* Autonomous Systems: Self-driving cars, robotics.

**7**. Getting Started with Neural Networks

To get started with neural networks, you can use popular frameworks like TensorFlow or PyTorch. Here's a simple example of building a neural network using TensorFlow:

python

Copy code

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Create a simple neural network model

model = Sequential([

Dense(32, input\_shape=(784,), activation='relu'), # Input layer with 784 features (e.g., pixels of an image)

Dense(64, activation='relu'), # Hidden layer with 64 neurons

Dense(10, activation='softmax') # Output layer with 10 neurons (e.g., for 10 classes)

])

# Compile the model

model.compile(optimizer='adam', loss='sparse\_categorical\_crossentropy', metrics=['accuracy'])

# Train the model (example with dummy data)

# model.fit(x\_train, y\_train, epochs=10)

# Evaluate the model (example with dummy data)

# loss, accuracy = model.evaluate(x\_test, y\_test)

This example demonstrates how to define, compile, and train a simple neural network using TensorFlow. The model consists of an input layer, one hidden layer, and an output layer.

## Conclusion

Neural networks are powerful tools for solving complex problems across various domains. Understanding their basic structure, how they are trained, and their applications can help you leverage their capabilities in your projects. Start experimenting with neural networks using frameworks like TensorFlow and PyTorch to gain practical experience.

## 1. Batches

**Batches** refer to groups of data points that are processed together in one forward and backward pass of a neural network.

* **Why use batches?** Instead of feeding the entire dataset into the neural network at once, we divide the data into smaller subsets called batches. This helps in:
  + Reducing memory usage
  + Making training faster by leveraging parallel computation
  + Providing more frequent updates to the model weights
* **Batch Size:** This is the number of samples in each batch. For example, if you have a dataset of 1000 samples and a batch size of 100, you will have 10 batches.

## 2. Layers

**Layers** are the building blocks of neural networks, where each layer consists of a set of neurons that transform the input data.

* **Input Layer:** The first layer of the neural network that receives the input data.
* **Hidden Layers:** Layers between the input and output layers. They apply various transformations to the input data. The term "hidden" just means they are not directly observed in the input/output.
* **Output Layer:** The final layer that produces the output of the network. The number of neurons here often corresponds to the number of classes in a classification problem or the number of regression outputs.
* **Activation Functions:** These functions introduce non-linearity into the network. Common examples include ReLU (Rectified Linear Unit), Sigmoid, and Tanh. They are applied to the output of each layer.

## 3. Objects in Neural Networks

In the context of coding, especially with libraries like PyTorch, neural network components are often organized as objects.

* **Model:** The neural network itself is usually an object. For example, in PyTorch, you define a class that inherits from torch.nn.Module.
* **Layers as Objects:** Each layer (like linear layers, convolutional layers) is an object. In PyTorch, you might use torch.nn.Linear for a fully connected layer or torch.nn.Conv2d for a convolutional layer.
* **Optimizer:** An object that updates the weights of the network based on the gradients. Examples include torch.optim.SGD and torch.optim.Adam.
* **Loss Function:** An object that measures how well the model's predictions match the target values. Examples include torch.nn.CrossEntropyLoss and torch.nn.MSELoss.

Building a neural network from scratch is a great way to understand how the various components work together, including hidden layers and activation functions. Below, I will explain hidden layers, their purpose, and common activation functions, along with a simple implementation in Python.

## 1. Hidden Layers

**Hidden Layers** are the layers in a neural network that are not directly exposed to the input or output. They perform transformations on the input data and help the model learn complex patterns. Each hidden layer consists of neurons, and the outputs of one layer serve as inputs to the next layer.

## 2. Purpose of Hidden Layers

* **Feature Extraction:** Hidden layers allow the network to learn hierarchical feature representations. Lower layers may learn basic features, while deeper layers can learn more complex features.
* **Non-linearity:** By using activation functions, hidden layers introduce non-linearity to the model, enabling it to learn complex relationships in the data.

## 3. Common Activation Functions

Here are some commonly used activation functions for hidden layers:

**3.1. Sigmoid**

* **Formula:** f(x)=11+e−xf(x) = \frac{1}{1 + e^{-x}}f(x)=1+e−x1​
* **Range:** (0, 1)
* **Pros:** Smooth gradient, works well for binary classification.
* **Cons:** Can cause vanishing gradients, especially in deep networks.

def sigmoid(x):

return 1 / (1 + np.exp(-x))

## 3.2. Tanh

* **Formula:** f(x)=tanh⁡(x)=ex−e−xex+e−xf(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}f(x)=tanh(x)=ex+e−xex−e−x​
* **Range:** (-1, 1)
* **Pros:** Zero-centered, outputs are more evenly distributed.
* **Cons:** Still suffers from the vanishing gradient problem.

def tanh(x):

return np.tanh(x)

## 3.3. ReLU (Rectified Linear Unit)

* **Formula:** f(x)=max⁡(0,x)f(x) = \max(0, x)f(x)=max(0,x)
* **Range:** [0, ∞)
* **Pros:** Simpler, computationally efficient, helps mitigate the vanishing gradient problem.
* **Cons:** Can suffer from dying ReLU problem where neurons can become inactive.

python

Copy code

def relu(x):

return np.maximum(0, x)

## 3.4. Leaky ReLU

* **Formula:**
  + f(x)=xf(x) = xf(x)=x if x>0x > 0x>0
  + f(x)=αxf(x) = \alpha xf(x)=αx if x≤0x \leq 0x≤0 (where α\alphaα is a small constant)
* **Range:** (-∞, ∞)
* **Pros:** Allows a small gradient when the unit is inactive.
* **Cons:** Still has some drawbacks of ReLU.

python

Copy code

def leaky\_relu(x, alpha=0.01):

return np.where(x > 0, x, alpha \* x)

## 4. Implementing a Simple Neural Network from Scratch

Let's create a simple neural network from scratch using NumPy, incorporating hidden layers and activation functions. This network will consist of one hidden layer and one output layer.

## Step-by-Step Implementation

Here’s how you can implement a simple neural network with one hidden layer using ReLU as the activation function:

import numpy as np

# Sigmoid activation function

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# ReLU activation function

def relu(x):

return np.maximum(0, x)

# Derivative of the ReLU function

def relu\_derivative(x):

return np.where(x > 0, 1, 0)

# Derivative of the sigmoid function

def sigmoid\_derivative(x):

return x \* (1 - x)

# Initialize parameters for a simple neural network

input\_size = 3 # Number of input features

hidden\_size = 4 # Number of neurons in the hidden layer

output\_size = 1 # Number of output neurons

# Initialize weights and biases

np.random.seed(42)

weights\_input\_hidden = np.random.rand(input\_size, hidden\_size)

bias\_hidden = np.random.rand(hidden\_size)

weights\_hidden\_output = np.random.rand(hidden\_size, output\_size)

bias\_output = np.random.rand(output\_size)

# Forward pass function

def forward\_pass(X):

# Calculate hidden layer activation

hidden\_input = np.dot(X, weights\_input\_hidden) + bias\_hidden

hidden\_output = relu(hidden\_input) # Apply ReLU activation

# Calculate output layer activation

output\_input = np.dot(hidden\_output, weights\_hidden\_output) + bias\_output

output = sigmoid(output\_input) # Apply sigmoid activation

return hidden\_output, output

# Backward pass function

def backward\_pass(X, y, hidden\_output, output, learning\_rate=0.01):

global weights\_input\_hidden, bias\_hidden, weights\_hidden\_output, bias\_output

# Calculate the error

output\_error = y - output # Error in output

output\_delta = output\_error \* sigmoid\_derivative(output) # Delta for output layer

# Calculate hidden layer error

hidden\_error = output\_delta.dot(weights\_hidden\_output.T) # Backpropagate error

hidden\_delta = hidden\_error \* relu\_derivative(hidden\_output) # Delta for hidden layer

# Update weights and biases

weights\_hidden\_output += hidden\_output.T.dot(output\_delta) \* learning\_rate

bias\_output += np.sum(output\_delta, axis=0) \* learning\_rate

weights\_input\_hidden += X.T.dot(hidden\_delta) \* learning\_rate

bias\_hidden += np.sum(hidden\_delta, axis=0) \* learning\_rate

# Training the neural network

def train(X, y, epochs):

for epoch in range(epochs):

hidden\_output, output = forward\_pass(X)

backward\_pass(X, y, hidden\_output, output)

if epoch % 100 == 0:

loss = np.mean(np.square(y - output)) # Mean squared error

print(f'Epoch {epoch}, Loss: {loss:.4f}')

# Sample input and output (XOR problem)

X = np.array([[0, 0, 0],

[0, 0, 1],

[0, 1, 0],

[0, 1, 1],

[1, 0, 0],

[1, 0, 1],

[1, 1, 0],

[1, 1, 1]])

y = np.array([[0], [0], [0], [1], [0], [1], [1], [1]]) # Expected output

# Train the neural network

train(X, y, epochs=1000)

## Explanation of the Code

1. **Initialization:**
   * We define the sizes of the input layer, hidden layer, and output layer.
   * We initialize the weights and biases randomly.
2. **Forward Pass:**
   * We compute the input to the hidden layer by multiplying the input data with the weights and adding the bias.
   * We apply the ReLU activation function to get the hidden layer output.
   * We compute the output layer input and apply the sigmoid activation function.
3. **Backward Pass:**
   * We calculate the output error and the delta (gradient) for the output layer.
   * We propagate this error back to the hidden layer to calculate the hidden error and delta.
   * We update the weights and biases using the calculated deltas.
4. **Training:**
   * We train the network for a specified number of epochs, printing the loss every 100 epochs.

### Conclusion

By following this guide, you have learned about hidden layers, activation functions, and how to implement a simple neural network from scratch using Python. This foundational understanding is crucial as you delve deeper into neural networks and machine learning.