Implementing a Neural Network from Scratch in Python

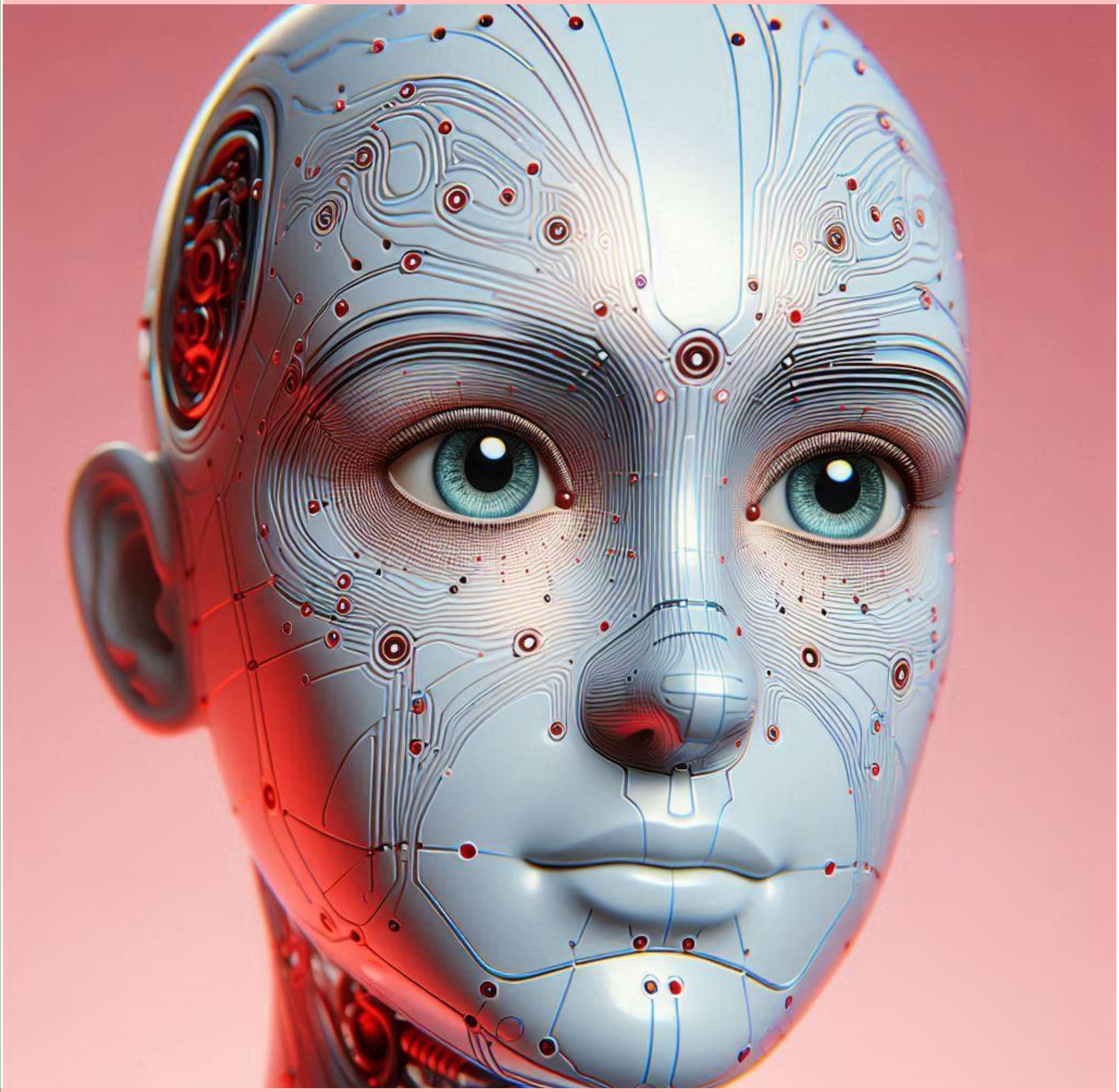


Table of Contents

Implementing a Neural Network from Scratch in Python

1. Introduction to Neural Networks

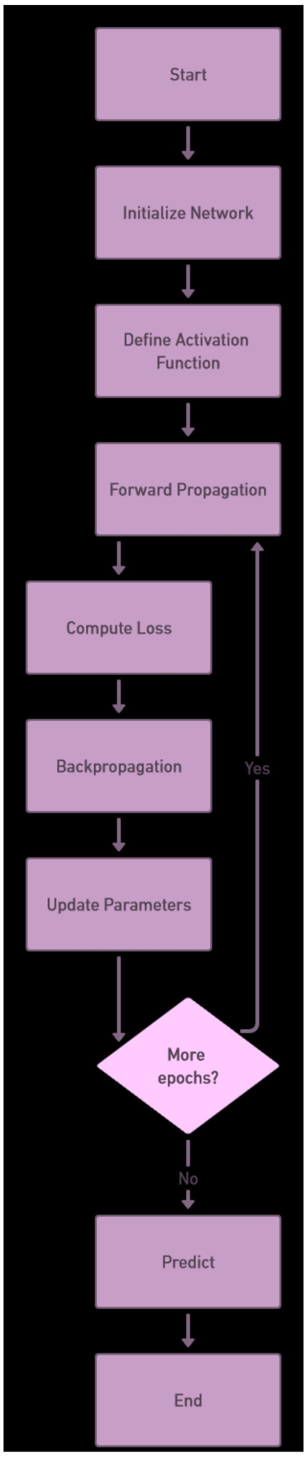
A neural network is a series of algorithms that attempt to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. The network consists of layers of interconnected nodes, or neurons, each performing a specific function.

2. The Structure of a Neural Network

A typical neural network consists of three types of layers:

- **Input Layer**: Receives the input data.
- **Hidden Layers**: Perform computations and feature extraction.
- **Output Layer**: Produces the final output.

Here is a simple diagram to illustrate a basic neural network:



3. Implementation in Python

Let's implement a simple neural network in Python to classify data. We'll use NumPy for numerical computations.

Step 1: Import Libraries

```
import numpy as np
```

Step 2: Initialize the Network

We initialize the network with random weights and biases

```
   - Input: Input size, hidden size, output size
   - Process: Initialize weights (W1, W2) and biases (b1, b2) with random values
```

```
def initialize_network(input_size, hidden_size, output_size):
    np.random.seed(42)  # For reproducibility
    network = {
        'W1': np.random.randn(input_size, hidden_size),
        'b1': np.zeros((1, hidden_size)),
        'W2': np.random.randn(hidden_size, output_size),
        'b2': np.zeros((1, output_size))
    }
    return network
```

Step 3: Activation Functions

We'll use the sigmoid function for activation. The sigmoid function and its derivative are used for activation and gradient computation.

```
   - Sigmoid function: \( \sigma(x) = \frac{1}{1 + e^{-x}} \)
   - Sigmoid derivative function: \( \sigma'(x) = \sigma(x) \cdot (1 - \sigma(x)) \)
```

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)
```

Step 4: Forward Propagation

Forward propagation involves passing input data through the layers of the network to obtain an output. Each neuron in a layer performs a weighted sum of its inputs, applies an activation function, and passes the result to the next layer. Computes the output of the network by passing inputs through each layer.

```
   - Input: Network parameters (W1, b1, W2, b2), input data (X)
   - Process:
     - Compute \( z1 = X \cdot W1 + b1 \)
     - Compute \( a1 = \sigma(z1) \)
     - Compute \( z2 = a1 \cdot W2 + b2 \)
     - Compute \( a2 = \sigma(z2) \)
   - Output: Predicted output (a2), cache (z1, a1, z2, a2)
```

```
def forward_propagation(network, X):
    W1, b1 = network['W1'], network['b1']
    W2, b2 = network['W2'], network['b2']

     # Input to hidden layer
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)

     # Hidden to output layer
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    cache = (z1, a1, z2, a2)
    return a2, cache
```

Step 5: Loss Function

The loss function measures how well the neural network's predictions match the actual data. Common loss functions include Mean Squared Error (MSE) for regression and Cross-Entropy Loss for classification.

Measures the error between predicted and actual outputs We'll use Mean Squared Error (MSE) for simplicity.
- Input: Actual output (y), predicted output (a2)
- Process: Compute mean squared error loss
- Output: Loss value

```
def compute_loss(y_true, y_pred):
    return np.mean((y_true - y_pred) ** 2)
```

Step 6: Backpropagation

Backpropagation is the process of updating the network's weights to minimize the loss. It involves computing the gradient of the loss function with respect to each weight and adjusting the weights in the direction that reduces the loss.

Computes the gradient of the loss function with respect to each weight.

- Input: Network parameters (W1, b1, W2, b2), cache (z1, a1, z2, a2), input data (X), actual output (y)
- Process:
  - Compute output error and delta
  - Compute hidden layer error and delta
  - Compute gradients for W2, b2, W1, b1
- Output: Gradients (dW1, db1, dW2, db2)

```
def backward_propagation(network, cache, X, y):
    W1, b1 = network['W1'], network['b1']
    W2, b2 = network['W2'], network['b2']
    z1, a1, z2, a2 = cache

     # Output layer error
    output_error = y - a2
    output_delta = output_error * sigmoid_derivative(a2)

     # Hidden layer error
    hidden_error = np.dot(output_delta, W2.T)
    hidden_delta = hidden_error * sigmoid_derivative(a1)

     # Gradient for W2 and b2
    dW2 = np.dot(a1.T, output_delta)
    db2 = np.sum(output_delta, axis=0, keepdims=True)

     # Gradient for W1 and b1
    dW1 = np.dot(X.T, hidden_delta)
    db1 = np.sum(hidden_delta, axis=0, keepdims=True)

    gradients = {'dW1': dW1, 'db1': db1, 'dW2': dW2, 'db2': db2}
    return gradients
```

Step 7: Update Weights

Adjusts the weights using the computed gradients

- Input: Network parameters (W1, b1, W2, b2), gradients (dW1, db1, dW2, db2), learning rate
- Process: Update weights and biases using gradients

```
def update_parameters(network, gradients, learning_rate):
    network['W1'] += learning_rate * gradients['dW1']
    network['b1'] += learning_rate * gradients['db1']
    network['W2'] += learning_rate * gradients['dW2']
    network['b2'] += learning_rate * gradients['db2']
    return network
```

Step 8: Training Function

Training a neural network involves iteratively performing forward propagation, computing the loss, performing backpropagation, and updating the weights.
Trains the network by performing forward and backward propagation and updating weights iteratively.

```
   - Input: Training data (X, y), network parameters, epochs, learning rate
   - Process: Repeat forward propagation, compute loss, backpropagation, and update
       parameters for each epoch
   - Output: Trained network parameters
```

```
def train_network(X, y, input_size, hidden_size, output_size, epochs, learning_rate):
    network = initialize_network(input_size, hidden_size, output_size)
    for epoch in range(epochs):
        # Forward propagation
        y_pred, cache = forward_propagation(network, X)

        # Compute loss
        loss = compute_loss(y, y_pred)
        if epoch % 100 == 0:
            print(f'Epoch {epoch}, Loss: {loss}')

        # Backward propagation
        gradients = backward_propagation(network, cache, X, y)

        # Update parameters
        network = update_parameters(network, gradients, learning_rate)

    return network
```

Step 9: Test the Network

Tests the trained network on sample input data

```
- Input: Trained network parameters, input data (X)
- Process: Forward propagation using trained parameters
- Output: Predicted output
```

```python
def predict(network, X):
    y_pred, _ = forward_propagation(network, X)
    return y_pred

# Example data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])  # XOR problem

# Training the network
input_size = 2
hidden_size = 2
output_size = 1
epochs = 10000
learning_rate = 0.1

trained_network = train_network(X, y, input_size, hidden_size, output_size, epochs, learning_rate)

# Making predictions
predictions = predict(trained_network, X)
print(f'Predictions: \n{predictions}')
```

4. Conclusion

By implementing a neural network from scratch, we've explored the core concepts and mechanisms underlying neural networks. This hands-on approach helps solidify understanding and prepares us for more advanced topics in neural networks and deep learning. The example provided can be extended and modified for more complex tasks and architectures.

Feel free to experiment with **different activation functions, loss functions, network architectures, and optimization techniques** to deepen your understanding further.

Constructive comments and feedback are welcomed