**PYTHON CODES**

1. Basic Architecture of a Neural Network

import numpy as np

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):

# Initialize weights and biases

self.weights1 = np.random.randn(input\_size, hidden\_size) \* 0.01

self.bias1 = np.zeros((1, hidden\_size))

self.weights2 = np.random.randn(hidden\_size, output\_size) \* 0.01

self.bias2 = np.zeros((1, output\_size))

def display\_architecture(self):

print("Input layer size:", self.weights1.shape[0])

print("Hidden layer size:", self.weights1.shape[1])

print("Output layer size:", self.weights2.shape[1])

2. Activation Functions

import matplotlib.pyplot as plt

def sigmoid(x):

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

def relu(x):

return np.maximum(0, x)

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

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

# Plot activation functions

x = np.linspace(-10, 10, 400)

plt.figure(figsize=(10, 6))

plt.plot(x, sigmoid(x), label='Sigmoid')

plt.plot(x, relu(x), label='ReLU')

plt.plot(x, leaky\_relu(x), label='Leaky ReLU')

plt.legend()

plt.title("Activation Functions")

plt.show()

3. Forward Propagation

def forward\_propagation(X, weights1, bias1, weights2, bias2):

# Layer 1

Z1 = np.dot(X, weights1) + bias1

A1 = relu(Z1)

# Layer 2

Z2 = np.dot(A1, weights2) + bias2

A2 = sigmoid(Z2)

return A1, A2

4. Overfitting and Regularization

def l2\_regularization(weights1, weights2, reg\_lambda):

reg\_loss = reg\_lambda \* (np.sum(np.square(weights1)) + np.sum(np.square(weights2)))

return reg\_loss

5. Training a Neural Network

def gradient\_descent(X, Y, weights1, bias1, weights2, bias2, learning\_rate, iterations):

for i in range(iterations):

# Forward propagation

A1, A2 = forward\_propagation(X, weights1, bias1, weights2, bias2)

# Backpropagation

m = X.shape[0]

dZ2 = A2 - Y

dW2 = (1/m) \* np.dot(A1.T, dZ2)

db2 = (1/m) \* np.sum(dZ2, axis=0, keepdims=True)

dZ1 = np.dot(dZ2, weights2.T) \* (A1 > 0) # Derivative of ReLU

dW1 = (1/m) \* np.dot(X.T, dZ1)

db1 = (1/m) \* np.sum(dZ1, axis=0, keepdims=True)

# Update weights and biases

weights1 -= learning\_rate \* dW1

bias1 -= learning\_rate \* db1

weights2 -= learning\_rate \* dW2

bias2 -= learning\_rate \* db2

# Print loss

if i % 100 == 0:

loss = np.mean(-Y \* np.log(A2) - (1 - Y) \* np.log(1 - A2))

print(f"Iteration {i}, Loss: {loss}")

6. Adaptive Learning Rates (Adam Optimizer)

def adam\_optimizer(X, Y, weights1, bias1, weights2, bias2, learning\_rate, beta1, beta2, epsilon, iterations):

m\_w1, v\_w1 = np.zeros\_like(weights1), np.zeros\_like(weights1)

m\_b1, v\_b1 = np.zeros\_like(bias1), np.zeros\_like(bias1)

m\_w2, v\_w2 = np.zeros\_like(weights2), np.zeros\_like(weights2)

m\_b2, v\_b2 = np.zeros\_like(bias2), np.zeros\_like(bias2)

for t in range(1, iterations + 1):

# Forward propagation

A1, A2 = forward\_propagation(X, weights1, bias1, weights2, bias2)

# Backpropagation

m = X.shape[0]

dZ2 = A2 - Y

dW2 = (1/m) \* np.dot(A1.T, dZ2)

db2 = (1/m) \* np.sum(dZ2, axis=0, keepdims=True)

dZ1 = np.dot(dZ2, weights2.T) \* (A1 > 0)

dW1 = (1/m) \* np.dot(X.T, dZ1)

db1 = (1/m) \* np.sum(dZ1, axis=0, keepdims=True)

# Update moving averages of gradients

m\_w1 = beta1 \* m\_w1 + (1 - beta1) \* dW1

v\_w1 = beta2 \* v\_w1 + (1 - beta2) \* (dW1 \*\* 2)

m\_b1 = beta1 \* m\_b1 + (1 - beta1) \* db1

v\_b1 = beta2 \* v\_b1 + (1 - beta2) \* (db1 \*\* 2)

m\_w2 = beta1 \* m\_w2 + (1 - beta1) \* dW2

v\_w2 = beta2 \* v\_w2 + (1 - beta2) \* (dW2 \*\* 2)

m\_b2 = beta1 \* m\_b2 + (1 - beta1) \* db2

v\_b2 = beta2 \* v\_b2 + (1 - beta2) \* (db2 \*\* 2)

# Bias correction

m\_w1\_corr = m\_w1 / (1 - beta1 \*\* t)

v\_w1\_corr = v\_w1 / (1 - beta2 \*\* t)

m\_b1\_corr = m\_b1 / (1 - beta1 \*\* t)

v\_b1\_corr = v\_b1 / (1 - beta2 \*\* t)

m\_w2\_corr = m\_w2 / (1 - beta1 \*\* t)

v\_w2\_corr = v\_w2 / (1 - beta2 \*\* t)

m\_b2\_corr = m\_b2 / (1 - beta1 \*\* t)

v\_b2\_corr = v\_b2 / (1 - beta2 \*\* t)

# Update weights and biases

weights1 -= learning\_rate \* m\_w1\_corr / (np.sqrt(v\_w1\_corr) + epsilon)

bias1 -= learning\_rate \* m\_b1\_corr / (np.sqrt(v\_b1\_corr) + epsilon)

weights2 -= learning\_rate \* m\_w2\_corr / (np.sqrt(v\_w2\_corr) + epsilon)

bias2 -= learning\_rate \* m\_b2\_corr / (np.sqrt(v\_b2\_corr) + epsilon)