import numpy as np

# Sample input data for the image

input\_data = np.array([0, 256, 0, 256, 0, 256, 0, 256, 0], dtype=float)

# Manually initialize weights and bias for the first hidden layer

weights = np.zeros\_like(input\_data)

bias = 0

# Assuming you have target values (labels) y

# For demonstration purposes, let's assume some target values

y\_true = np.array([1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0], dtype=float)

# Define learning rate and number of epochs

learning\_rate = 0.001

n\_epochs = 1000

# Stochastic Gradient Descent

for epoch in range(n\_epochs):

# Shuffle the data to introduce randomness

indices = np.arange(len(y\_true))

np.random.shuffle(indices)

for i in indices:

# Select a random training sample

x\_sample = input\_data[i]

y\_sample = y\_true[i]

# Forward pass

weighted\_sum = x\_sample \* weights + bias

prediction = 1 / (1 + np.exp(-weighted\_sum))

# Compute gradients

gradient\_weights = 2 \* (prediction - y\_sample) \* prediction \* (1 - prediction) \* x\_sample

gradient\_bias = 2 \* (prediction - y\_sample) \* prediction \* (1 - prediction)

# Update weights and bias

weights -= learning\_rate \* gradient\_weights

bias -= learning\_rate \* gradient\_bias

# Print the final weights and bias

print("Final Weights:", weights)

print("Final Bias:", bias)

import numpy as np

# Sample input data for the image

input\_data = np.array([0, 256, 0, 256, 0, 256, 0, 256, 0], dtype=float)

# Manually initialize weights and biases for the first hidden layer

weights = np.array([[-0.02, 0.03, 0.01, 0.04, 0.03, -0.02, 0.05, -0.02],

[-0.01, 0.02, -0.02, -0.03, -0.03, 0.04, -0.03, -0.01],

[0.03, -0.04, -0.02, -0.03, -0.02, 0.01, -0.02, -0.04],

[-0.02, 0.05, 0.03, -0.01, 0.02, -0.03, 0.02, 0.04],

[0.03, -0.02, -0.01, 0.02, -0.03, -0.01, -0.02, 0.03],

[-0.04, -0.03, -0.04, -0.01, 0.02, -0.02, -0.02, -0.03],

[0.04, 0.03, -0.04, -0.01, -0.01, -0.03, 0.03, 0.02],

[-0.03, 0.04, 0.01, -0.01, -0.01, -0.04, 0.01, 0.03],

[0.02, -0.04, 0.03, -0.04, 0.01, -0.01, -0.02, 0.03]], dtype=float)

biases = np.array([0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1], dtype=float)

# Assuming you have target values (labels) y

# For demonstration purposes, let's assume some target values

y\_true = np.array([1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0], dtype=float)

# Define a learning rate, momentum rate, and number of iterations

learning\_rate = 0.0001 # Adjust as needed

momentum\_rate = 0.9

n\_iterations = 1000

# Initialize momentum terms for weights and biases

momentum\_weights = np.zeros\_like(weights)

momentum\_biases = np.zeros\_like(biases)

print(momentum\_weights)

print(momentum\_biases)

# Momentum-Based Gradient Descent

for iteration in range(n\_iterations):

# Forward pass

weighted\_sum = np.dot(input\_data, weights) + biases

first\_hidden\_activations = np.maximum(0, weighted\_sum) # Applying ReLU activation function

# Compute mean squared error

mse = np.mean((first\_hidden\_activations - y\_true) \*\* 2)

# Backward pass (compute gradients)

gradients = 2 / len(y\_true) \* np.dot((first\_hidden\_activations - y\_true), weights.T)

# Update momentum terms

momentum\_weights = momentum\_rate \* momentum\_weights + learning\_rate \* gradients.T.dot(input\_data.reshape(-1, 1))

momentum\_biases = momentum\_rate \* momentum\_biases + learning\_rate \* np.sum(gradients)

# Update weights and biases using the momentum terms

weights -= momentum\_weights

biases -= momentum\_biases

# Print the current iteration's mean squared error

#if iteration % 100 == 0:

##print(f"Iteration {iteration}, Mean Squared Error: {mse}")

# Print the final weights and biases

#print("\nFinal Weights:")

#print(weights)

#print("\nFinal Biases:")

#print(biases)