**Name: Manas Parab**

**Roll No: 40**

Deep Learning

**Experiment No. 03**

# Stochastic Gradient Descent Code:

import numpy as np

# Define the SGD function for training

def stochastic\_gradient\_descent(X, y, learning\_rate, epochs, batch\_size): input\_size = X.shape[1]

output\_size = 1 # For regression task, we have one output neuron

# Initialize weights and biases

weights = np.random.randn(input\_size, output\_size) biases = np.random.randn(output\_size)

for epoch in range(epochs):

# Shuffle the data for each epoch

random\_indices = np.random.permutation(len(X)) X\_shuffled = X[random\_indices]

y\_shuffled = y[random\_indices]

for batch\_start in range(0, len(X), batch\_size): # Get a batch of data

X\_batch = X\_shuffled[batch\_start:batch\_start + batch\_size] y\_batch = y\_shuffled[batch\_start:batch\_start + batch\_size]

# Forward pass

y\_pred = X\_batch.dot(weights) + biases

# Compute the loss (Mean Squared Error) loss = ((y\_batch - y\_pred) \*\* 2).mean()

# Backpropagation to compute gradients

gradient\_w = -2 \* X\_batch.T.dot(y\_batch - y\_pred) / batch\_size gradient\_b = -2 \* np.sum(y\_batch - y\_pred) / batch\_size

# Update weights and biases

weights -= learning\_rate \* gradient\_w biases -= learning\_rate \* gradient\_b

# Print the loss after each epoch

print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}") return weights, biases

# Sample data np.random.seed(10)

X\_train = 2 \* np.random.rand(20, 1)

y\_train = 4 + 3 \* X\_train + np.random.randn(20, 1)

# Hyperparameters learning\_rate = 0.01

epochs = 20

batch\_size = 10

# Training using SGD

trained\_weights, trained\_biases = stochastic\_gradient\_descent(X\_train, y\_train, learning\_rate, epochs, batch\_size)

# Print the final trained weights and biases print("Trained Weights:", trained\_weights) print("Trained Biases:", trained\_biases)

# Output:

1. **Mini Batch Gradient Descent Code:**

import numpy as np

# Define the Mini-Batch Gradient Descent function for training

def mini\_batch\_gradient\_descent(X, y, learning\_rate, epochs, batch\_size): input\_size = X.shape[1]

output\_size = 1 # For regression task, we have one output neuron

# Initialize weights and biases

weights = np.random.randn(input\_size, output\_size) biases = np.random.randn(output\_size) num\_batches = len(X) // batch\_size

for epoch in range(epochs):

# Shuffle the data for each epoch

random\_indices = np.random.permutation(len(X)) X\_shuffled = X[random\_indices]

y\_shuffled = y[random\_indices]

for batch\_num in range(num\_batches):

# Get a batch of data

X\_batch = X\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size] y\_batch = y\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size]

# Forward pass

y\_pred = X\_batch.dot(weights) + biases

# Compute the loss (Mean Squared Error) loss = ((y\_batch - y\_pred) \*\* 2).mean()

# Backpropagation to compute gradients

gradient\_w = -2 \* X\_batch.T.dot(y\_batch - y\_pred) / batch\_size gradient\_b = -2 \* np.sum(y\_batch - y\_pred) / batch\_size

# Update weights and biases

weights -= learning\_rate \* gradient\_w biases -= learning\_rate \* gradient\_b

# Print the loss after each epoch

print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}") return weights, biases

# Sample data np.random.seed(14)

X\_train = 2 \* np.random.rand(30, 1)

y\_train = 4 + 3 \* X\_train + np.random.randn(30, 1)

# Hyperparameters learning\_rate = 0.01

epochs = 30

batch\_size = 10

# Training using Mini-Batch Gradient Descent

trained\_weights, trained\_biases = mini\_batch\_gradient\_descent(X\_train, y\_train, learning\_rate, epochs, batch\_size)

# Print the final trained weights and biases print("Trained Weights:", trained\_weights) print("Trained Biases:", trained\_biases)

# Output:

1. **Momentum GD Code:**

import numpy as np

# Define the Gradient Descent with Momentum function for training

def momentum\_gradient\_descent(X, y, learning\_rate, epochs, batch\_size, momentum): input\_size = X.shape[1]

output\_size = 1 # For regression task, we have one output neuron

# Initialize weights, biases, and momentum terms weights = np.random.randn(input\_size, output\_size) biases = np.random.randn(output\_size)

velocity\_w = np.zeros\_like(weights) velocity\_b = np.zeros\_like(biases) num\_batches = len(X) // batch\_size

for epoch in range(epochs):

# Shuffle the data for each epoch

random\_indices = np.random.permutation(len(X)) X\_shuffled = X[random\_indices]

y\_shuffled = y[random\_indices]

for batch\_num in range(num\_batches):

# Get a batch of data

X\_batch = X\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size] y\_batch = y\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size]

# Forward pass

y\_pred = X\_batch.dot(weights) + biases

# Compute the loss (Mean Squared Error) loss = ((y\_batch - y\_pred) \*\* 2).mean()

# Backpropagation to compute gradients

gradient\_w = -2 \* X\_batch.T.dot(y\_batch - y\_pred) / batch\_size gradient\_b = -2 \* np.sum(y\_batch - y\_pred) / batch\_size

# Update momentum terms

velocity\_w = momentum \* velocity\_w - learning\_rate \* gradient\_w velocity\_b = momentum \* velocity\_b - learning\_rate \* gradient\_b

# Update weights and biases with momentum weights += velocity\_w

biases += velocity\_b

# Print the loss after each epoch

print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}") return weights, biases

# Sample data np.random.seed(7)

X\_train = 2 \* np.random.rand(10, 1)

y\_train = 4 + 3 \* X\_train + np.random.randn(10, 1)

# Hyperparameters learning\_rate = 0.01

epochs = 10

batch\_size = 10

momentum = 0.9

# Training using Gradient Descent with Momentum

trained\_weights, trained\_biases = momentum\_gradient\_descent(X\_train, y\_train, learning\_rate, epochs, batch\_size, momentum)

# Print the final trained weights and biases print("Trained Weights:", trained\_weights) print("Trained Biases:", trained\_biases)

# Output:

1. **Nestorev GD Code:**

import numpy as np

# Define the Nesterov Accelerated Gradient function for training

def nesterov\_gradient\_descent(X, y, learning\_rate, epochs, batch\_size, momentum): input\_size = X.shape[1]

output\_size = 1 # For regression task, we have one output neuron

# Initialize weights, biases, and momentum terms weights = np.random.randn(input\_size, output\_size) biases = np.random.randn(output\_size)

velocity\_w = np.zeros\_like(weights) velocity\_b = np.zeros\_like(biases) num\_batches = len(X) // batch\_size

for epoch in range(epochs):

# Shuffle the data for each epoch

random\_indices = np.random.permutation(len(X)) X\_shuffled = X[random\_indices]

y\_shuffled = y[random\_indices]

for batch\_num in range(num\_batches):

# Get a batch of data

X\_batch = X\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size] y\_batch = y\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size]

# Update weights and biases with Nesterov Accelerated Gradient weights\_ahead = weights + momentum \* velocity\_w biases\_ahead = biases + momentum \* velocity\_b

# Forward pass

y\_pred = X\_batch.dot(weights\_ahead) + biases\_ahead

# Compute the loss (Mean Squared Error) loss = ((y\_batch - y\_pred) \*\* 2).mean()

# Backpropagation to compute gradients

gradient\_w = -2 \* X\_batch.T.dot(y\_batch - y\_pred) / batch\_size gradient\_b = -2 \* np.sum(y\_batch - y\_pred) / batch\_size

# Update momentum terms

velocity\_w = momentum \* velocity\_w - learning\_rate \* gradient\_w velocity\_b = momentum \* velocity\_b - learning\_rate \* gradient\_b

# Update weights and biases weights += velocity\_w biases += velocity\_b

# Print the loss after each epoch

print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}") return weights, biases

# Sample data np.random.seed(4)

X\_train = 2 \* np.random.rand(16, 1)

y\_train = 4 + 3 \* X\_train + np.random.randn(16, 1)

# Hyperparameters learning\_rate = 0.01

epochs = 16

batch\_size = 10

momentum = 0.9

# Training using Nesterov Accelerated Gradient

trained\_weights, trained\_biases = nesterov\_gradient\_descent(X\_train, y\_train, learning\_rate, epochs, batch\_size, momentum)

# Print the final trained weights and biases print("Trained Weights:", trained\_weights) print("Trained Biases:", trained\_biases)

# Output:

1. **Adagrad GD**

# Code:

import numpy as np

# Define the Adagrad function for training

def adagrad\_gradient\_descent(X, y, learning\_rate, epochs, batch\_size): input\_size = X.shape[1]

output\_size = 1 # For regression task, we have one output neuron

# Initialize weights and biases

weights = np.random.randn(input\_size, output\_size) biases = np.random.randn(output\_size)

# Initialize the squared gradient accumulator grad\_squared\_w = np.zeros\_like(weights) grad\_squared\_b = np.zeros\_like(biases) num\_batches = len(X) // batch\_size

epsilon = 1e-8 # Small constant to avoid division by zero

for epoch in range(epochs):

# Shuffle the data for each epoch

random\_indices = np.random.permutation(len(X)) X\_shuffled = X[random\_indices]

y\_shuffled = y[random\_indices]

for batch\_num in range(num\_batches):

# Get a batch of data

X\_batch = X\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size] y\_batch = y\_shuffled[batch\_num \* batch\_size : (batch\_num + 1) \* batch\_size]

# Forward pass

y\_pred = X\_batch.dot(weights) + biases

# Compute the loss (Mean Squared Error) loss = ((y\_batch - y\_pred) \*\* 2).mean()

# Backpropagation to compute gradients

gradient\_w = -2 \* X\_batch.T.dot(y\_batch - y\_pred) / batch\_size gradient\_b = -2 \* np.sum(y\_batch - y\_pred) / batch\_size

# Accumulate squared gradients grad\_squared\_w += gradient\_w \*\* 2 grad\_squared\_b += gradient\_b \*\* 2

# Update weights and biases with Adagrad

weights -= learning\_rate \* gradient\_w / (np.sqrt(grad\_squared\_w) + epsilon) biases -= learning\_rate \* gradient\_b / (np.sqrt(grad\_squared\_b) + epsilon)

# Print the loss after each epoch

print(f"Epoch {epoch+1}/{epochs}, Loss: {loss:.4f}") return weights, biases

# Sample data np.random.seed(3)

X\_train = 2 \* np.random.rand(11, 1)

y\_train = 4 + 3 \* X\_train + np.random.randn(11, 1)

# Hyperparameters learning\_rate = 0.1

epochs = 11

batch\_size = 10

# Training using Adagrad

trained\_weights, trained\_biases = adagrad\_gradient\_descent(X\_train, y\_train, learning\_rate, epochs, batch\_size)

# Print the final trained weights and biases print("Trained Weights:", trained\_weights) print("Trained Biases:", trained\_biases)

# Output:

