Deep Learning

Experiment 03

GRADIENT DESCENT

Gradient descent is an iterative optimization algorithm used to minimize a loss function and find the optimal parameters of a model. In the context of machine learning, it's commonly used to update the weights and biases of a model during the training process. The idea is to move in the direction of steepest decrease of the loss function to reach a minimum.

**1. Stochastic Gradient Descent (SGD) :** In stochastic gradient descent, we randomly select a single data point from the training set for each iteration to compute the gradient and update the parameters.

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:

Epoch 1/20, Loss: 57.3320 Epoch 1/20, Loss: 49.6416 Epoch 2/20, Loss: 45.7042 Epoch 2/20, Loss: 43.6050 Epoch 3/20, Loss: 39.2863 Epoch 3/20, Loss: 35.3671 Epoch 4/20, Loss: 33.4853 Epoch 4/20, Loss: 29.0522 Epoch 5/20, Loss: 27.8584 Epoch 5/20, Loss: 24.6205 Epoch 6/20, Loss: 19.1716 Epoch 6/20, Loss: 24.9437 Epoch 7/20, Loss: 18.8669 Epoch 7/20, Loss: 18.2867 Epoch 8/20, Loss: 18.6677 Epoch 8/20, Loss: 12.7300 Epoch 9/20, Loss: 16.3552 Epoch 9/20, Loss: 10.2687 Epoch 10/20, Loss: 12.0403 Epoch 10/20, Loss: 10.5915 Epoch 11/20, Loss: 12.5399 Epoch 11/20, Loss: 6.8047 Epoch 12/20, Loss: 10.3556 Epoch 12/20, Loss: 6.2347 Epoch 13/20, Loss: 8.2871 Epoch 13/20, Loss: 6.0443 Epoch 14/20, Loss: 6.3545 Epoch 14/20, Loss: 6.0589 Epoch 15/20, Loss: 4.4422 Epoch 15/20, Loss: 6.3911 Epoch 16/20, Loss: 5.9123 Epoch 16/20, Loss: 3.6094 Epoch 17/20, Loss: 4.4455 Epoch 17/20, Loss: 3.9794 Epoch 18/20, Loss: 3.7058 Epoch 18/20, Loss: 3.8311 Epoch 19/20, Loss: 3.9784 Epoch 19/20, Loss: 2.7942 Epoch 20/20, Loss: 1.4482 Epoch 20/20, Loss: 4.7246

Trained Weights: [[3.48678723]]

Trained Biases: [2.40695215]

**2. Mini Batch Gradient Descent:** Mini batch gradient descent is a compromise between SGD and full batch GD. It uses small batches of data to compute the gradient and update the parameters.

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 the 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}, Batch {batch\_num+1}/{num\_batches}, 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:

Epoch 1/30, Batch 1/3, Loss: 15.5782 Epoch 1/30, Batch 2/3, Loss: 11.8381 Epoch 1/30, Batch 3/3, Loss: 15.5243 Epoch 2/30, Batch 1/3, Loss: 13.6629 Epoch 2/30, Batch 2/3, Loss: 8.3417 Epoch 2/30, Batch 3/3, Loss: 11.8590 Epoch 3/30, Batch 1/3, Loss: 11.9813 Epoch 3/30, Batch 2/3, Loss: 7.9761 Epoch 3/30, Batch 3/3, Loss: 6.8862 Epoch 4/30, Batch 1/3, Loss: 9.8944 Epoch 4/30, Batch 2/3, Loss: 6.8423 Epoch 4/30, Batch 3/3, Loss: 4.7981 Epoch 5/30, Batch 1/3, Loss: 5.4716 Epoch 5/30, Batch 2/3, Loss: 6.4146 Epoch 5/30, Batch 3/3, Loss: 5.5052 Epoch 6/30, Batch 1/3, Loss: 3.7364 Epoch 6/30, Batch 2/3, Loss: 5.5804 Epoch 6/30, Batch 3/3, Loss: 4.8775 Epoch 7/30, Batch 1/3, Loss: 4.8392 Epoch 7/30, Batch 2/3, Loss: 4.7902 Epoch 7/30, Batch 3/3, Loss: 2.1365 Epoch 8/30, Batch 1/3, Loss: 4.7370 Epoch 8/30, Batch 2/3, Loss: 2.3736 Epoch 8/30, Batch 3/3, Loss: 2.7936 Epoch 9/30, Batch 1/3, Loss: 4.3771 Epoch 9/30, Batch 2/3, Loss: 3.0928 . . . Epoch 30/30, Batch 3/3, Loss: 0.9769

Trained Weights: [[4.33097651]]

Trained Biases: [2.2285916]

**3. Momentum Gradient Descent:** Momentum gradient descent adds a momentum term that accelerates convergence and helps escape local minima.

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}, Batch {batch\_num+1}/{num\_batches}, 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:

Epoch 1/10, Batch 1/1, Loss: 70.3398 Epoch 2/10, Batch 1/1, Loss: 64.5946 Epoch 3/10, Batch 1/1, Loss: 54.5803 Epoch 4/10, Batch 1/1, Loss: 42.2232 Epoch 5/10, Batch 1/1, Loss: 29.5034 Epoch 6/10, Batch 1/1, Loss: 18.1247 Epoch 7/10, Batch 1/1, Loss: 9.2924 Epoch 8/10, Batch 1/1, Loss: 3.6123 Epoch 9/10, Batch 1/1, Loss: 1.1054 Epoch 10/10, Batch 1/1, Loss: 1.3119

Trained Weights: [[5.26345821]]

Trained Biases: [3.14535251]

**4. Nesterov Accelerated Gradient Descent (NAG)**: Nesterov momentum improves on traditional momentum by considering the future estimate of the gradient.

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}, Batch {batch\_num+1}/{num\_batches}, 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:

Epoch 1/16, Batch 1/1, Loss: 46.4056 Epoch 2/16, Batch 1/1, Loss: 40.3194 Epoch 3/16, Batch 1/1, Loss: 29.5730 Epoch 4/16, Batch 1/1, Loss: 18.6853 Epoch 5/16, Batch 1/1, Loss: 12.1055 Epoch 6/16, Batch 1/1, Loss: 5.9639 Epoch 7/16, Batch 1/1, Loss: 4.5615 Epoch 8/16, Batch 1/1, Loss: 1.7660 Epoch 9/16, Batch 1/1, Loss: 4.6945 Epoch 10/16, Batch 1/1, Loss: 7.1960 Epoch 11/16, Batch 1/1, Loss: 9.2042 Epoch 12/16, Batch 1/1, Loss: 10.5230 Epoch 13/16, Batch 1/1, Loss: 10.4598 Epoch 14/16, Batch 1/1, Loss: 8.1961 Epoch 15/16, Batch 1/1, Loss: 6.6230 Epoch 16/16, Batch 1/1, Loss: 6.7928

Trained Weights: [[5.10282206]]

Trained Biases: [2.59443318]

**5. Adagrad Gradient Descent:** Adagrad adapts the learning rate for each parameter based on historical gradient information.

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 the 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}, Batch {batch\_num+1}/{num\_batches}, 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:

Epoch 1/11, Batch 1/1, Loss: 20.0208 Epoch 2/11, Batch 1/1, Loss: 18.3632 Epoch 3/11, Batch 1/1, Loss: 19.9461 Epoch 4/11, Batch 1/1, Loss: 16.3753 Epoch 5/11, Batch 1/1, Loss: 17.3140 Epoch 6/11, Batch 1/1, Loss: 18.4489 Epoch 7/11, Batch 1/1, Loss: 16.9884 Epoch 8/11, Batch 1/1, Loss: 16.2863 Epoch 9/11, Batch 1/1, Loss: 16.5538 Epoch 10/11, Batch 1/1, Loss: 15.4883 Epoch 11/11, Batch 1/1, Loss: 14.8409

Trained Weights: [[2.33682509]]

Trained Biases: [0.67278367]