

**Name: Urvashi Patel**

**CSE(DS)**

**Roll No: 41**

## **Deep Learning**

### **Experiment No. 03**

#### **1)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
```

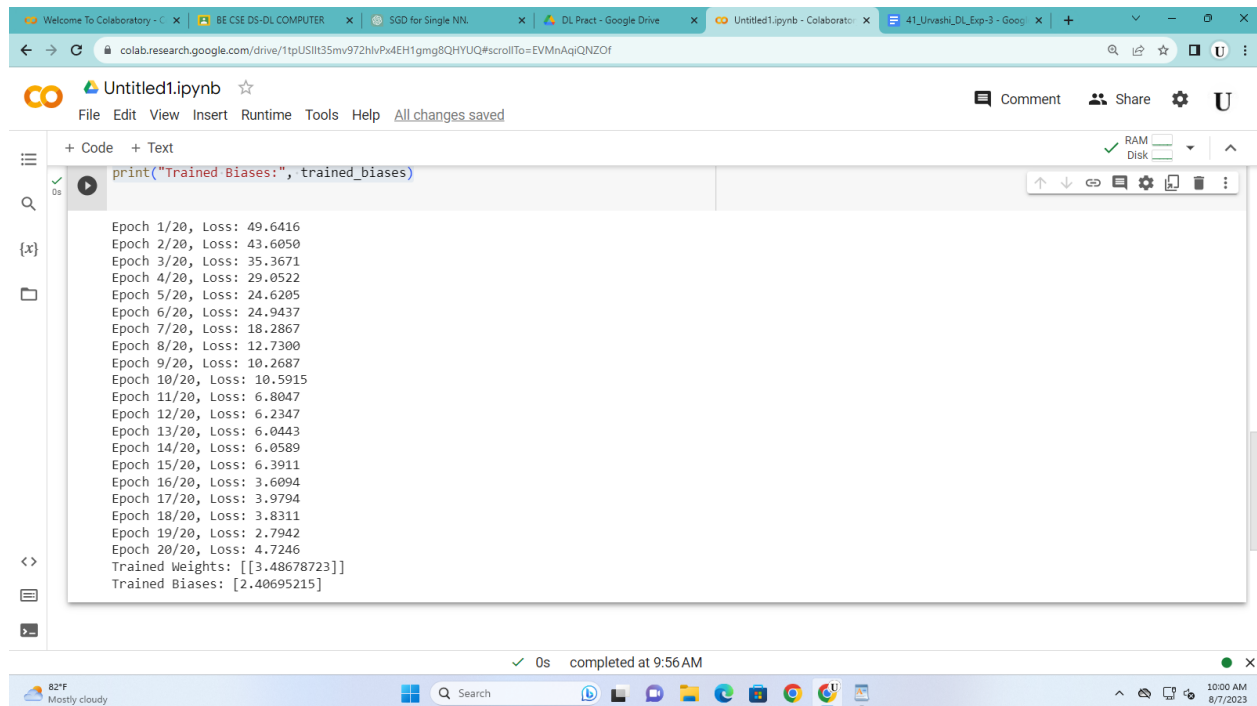
```
trained_weights, trained_biases = stochastic_gradient_descent(X_train, y_train, learning_rate,  
epochs, batch_size)
```

```
# Print the final trained weights and biases
```

```
print("Trained Weights:", trained_weights)
```

```
print("Trained Biases:", trained_biases)
```

## Output:



```
print("Trained Biases:", trained_biases)
```

```
Epoch 1/20, Loss: 49.6416
Epoch 2/20, Loss: 43.6050
Epoch 3/20, Loss: 35.3671
Epoch 4/20, Loss: 29.0522
Epoch 5/20, Loss: 24.6205
Epoch 6/20, Loss: 24.9437
Epoch 7/20, Loss: 18.2867
Epoch 8/20, Loss: 12.7300
Epoch 9/20, Loss: 10.2687
Epoch 10/20, Loss: 10.5915
Epoch 11/20, Loss: 6.8047
Epoch 12/20, Loss: 6.2347
Epoch 13/20, Loss: 6.0443
Epoch 14/20, Loss: 6.0589
Epoch 15/20, Loss: 6.3911
Epoch 16/20, Loss: 3.6094
Epoch 17/20, Loss: 3.9794
Epoch 18/20, Loss: 3.8311
Epoch 19/20, Loss: 2.7942
Epoch 20/20, Loss: 4.7246
Trained Weights: [[3.48678723]]
Trained Biases: [2.40695215]
```

## 2) 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
```

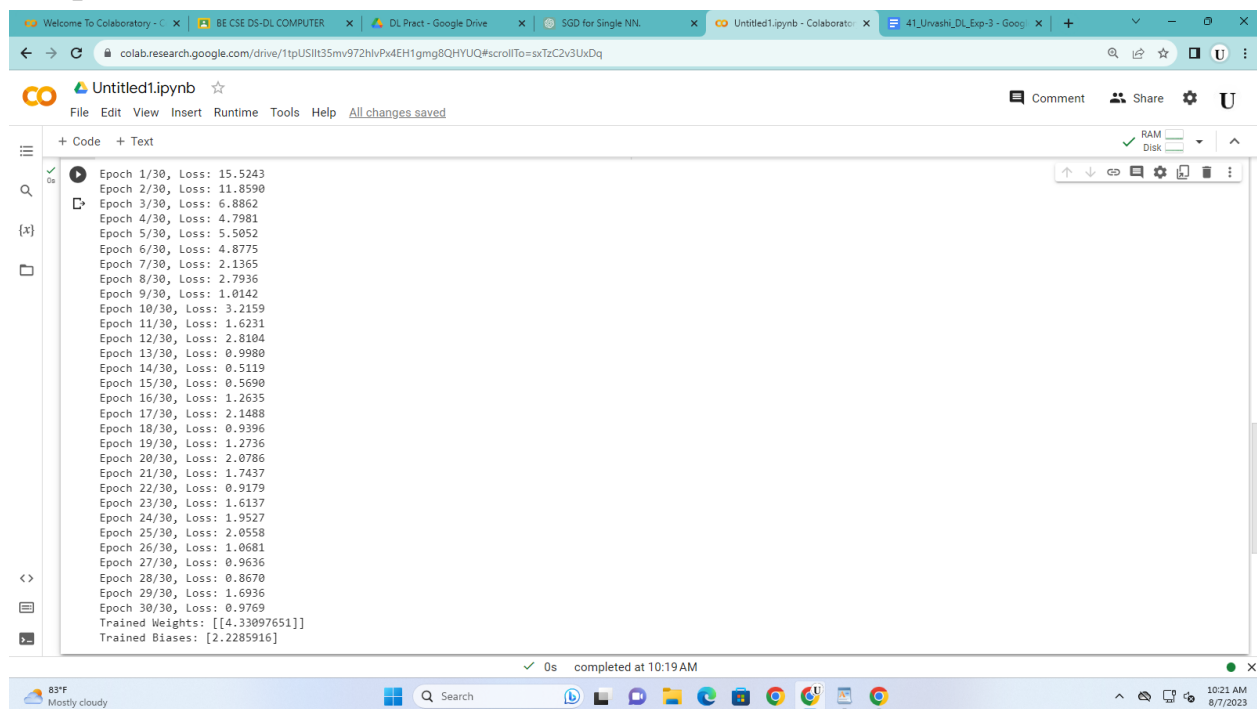
```
trained_weights, trained_biases = mini_batch_gradient_descent(X_train, y_train, learning_rate,  
epochs, batch_size)
```

```
# Print the final trained weights and biases
```

```
print("Trained Weights:", trained_weights)
```

```
print("Trained Biases:", trained_biases)
```

## Output:



```
Epoch 1/30, Loss: 15.5243  
Epoch 2/30, Loss: 11.8590  
Epoch 3/30, Loss: 6.8862  
Epoch 4/30, Loss: 4.7981  
Epoch 5/30, Loss: 5.5052  
Epoch 6/30, Loss: 4.8775  
Epoch 7/30, Loss: 2.1365  
Epoch 8/30, Loss: 2.7936  
Epoch 9/30, Loss: 1.0142  
Epoch 10/30, Loss: 3.2159  
Epoch 11/30, Loss: 1.6231  
Epoch 12/30, Loss: 2.8104  
Epoch 13/30, Loss: 0.9980  
Epoch 14/30, Loss: 0.5119  
Epoch 15/30, Loss: 0.5690  
Epoch 16/30, Loss: 1.2635  
Epoch 17/30, Loss: 2.1488  
Epoch 18/30, Loss: 0.9396  
Epoch 19/30, Loss: 1.2736  
Epoch 20/30, Loss: 2.0786  
Epoch 21/30, Loss: 1.7437  
Epoch 22/30, Loss: 0.9179  
Epoch 23/30, Loss: 1.6137  
Epoch 24/30, Loss: 1.9527  
Epoch 25/30, Loss: 2.0558  
Epoch 26/30, Loss: 1.0681  
Epoch 27/30, Loss: 0.9636  
Epoch 28/30, Loss: 0.8670  
Epoch 29/30, Loss: 1.6936  
Epoch 30/30, Loss: 0.9769  
Trained Weights: [[4.33097651]]  
Trained Biases: [2.2285916]
```

### 3) 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

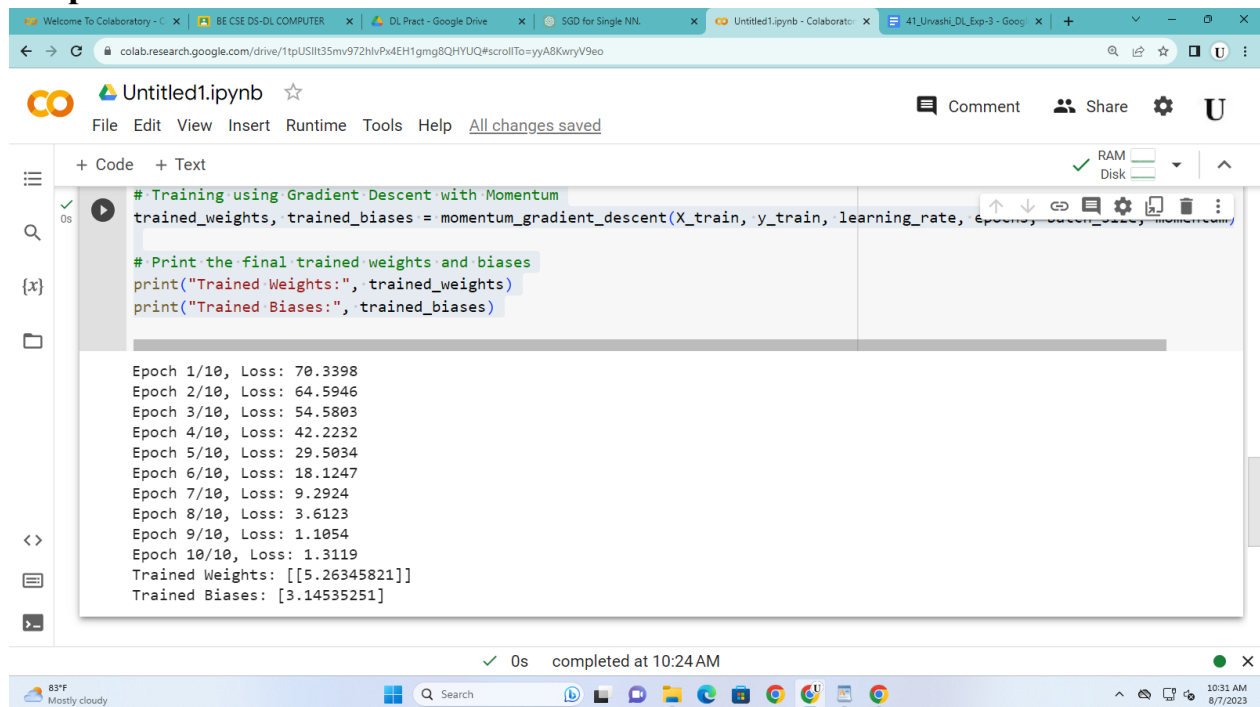
# Training using Gradient Descent with Momentum
trained_weights, trained_biases = momentum_gradient_descent(X_train, y_train, learning_rate,
epochs, batch_size, momentum)

# Print the final trained weights and biases
print("Trained Weights:", trained_weights)
print("Trained Biases:", trained_biases)

```



## Output:



```
# 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)
```

Epoch 1/10, Loss: 70.3398  
Epoch 2/10, Loss: 64.5946  
Epoch 3/10, Loss: 54.5803  
Epoch 4/10, Loss: 42.2232  
Epoch 5/10, Loss: 29.5034  
Epoch 6/10, Loss: 18.1247  
Epoch 7/10, Loss: 9.2924  
Epoch 8/10, Loss: 3.6123  
Epoch 9/10, Loss: 1.1054  
Epoch 10/10, Loss: 1.3119  
Trained Weights: [[5.26345821]]  
Trained Biases: [3.14535251]

## 4) Nesterov 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
```

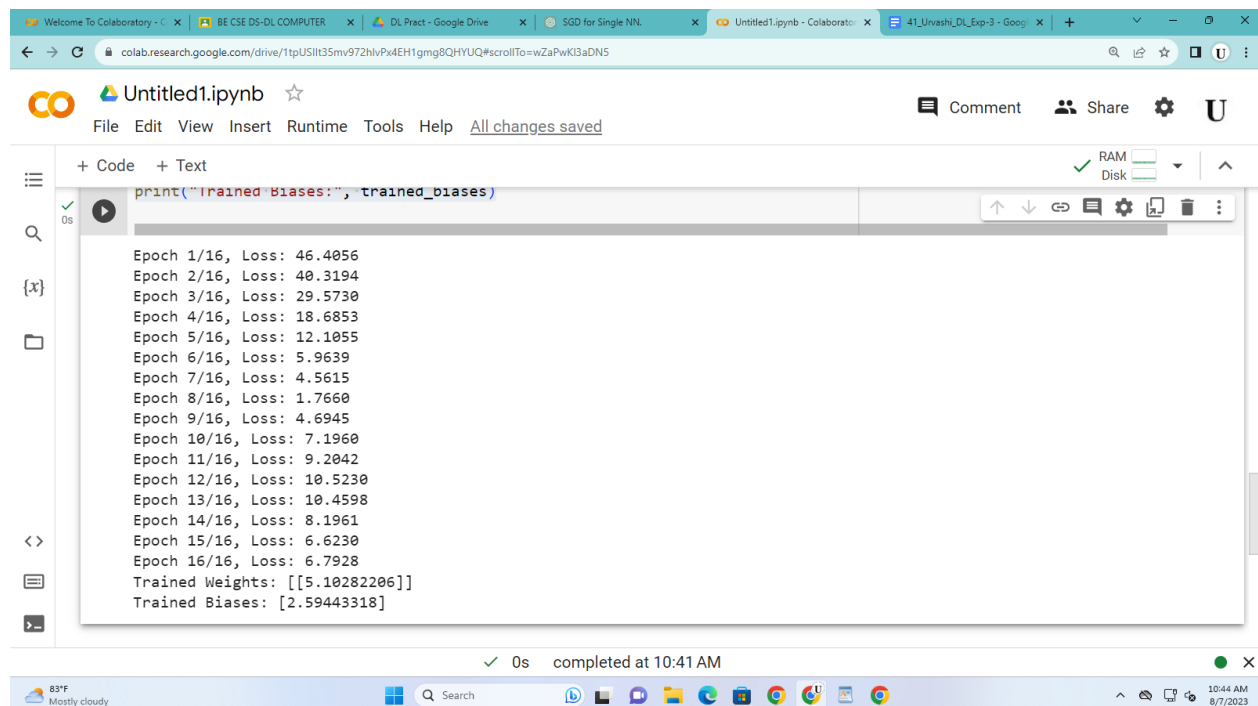
```
trained_weights, trained_biases = nesterov_gradient_descent(X_train, y_train, learning_rate,  
epochs, batch_size, momentum)
```

```
# Print the final trained weights and biases
```

```
print("Trained Weights:", trained_weights)
```

```
print("Trained Biases:", trained_biases)
```

## Output:



```
print("Trained Biases:", trained_biases)
```

```
Epoch 1/16, Loss: 46.4056  
Epoch 2/16, Loss: 40.3194  
Epoch 3/16, Loss: 29.5730  
Epoch 4/16, Loss: 18.6853  
Epoch 5/16, Loss: 12.1055  
Epoch 6/16, Loss: 5.9639  
Epoch 7/16, Loss: 4.5615  
Epoch 8/16, Loss: 1.7660  
Epoch 9/16, Loss: 4.6945  
Epoch 10/16, Loss: 7.1960  
Epoch 11/16, Loss: 9.2042  
Epoch 12/16, Loss: 10.5230  
Epoch 13/16, Loss: 10.4598  
Epoch 14/16, Loss: 8.1961  
Epoch 15/16, Loss: 6.6230  
Epoch 16/16, Loss: 6.7928  
Trained Weights: [[5.10282206]]  
Trained Biases: [2.59443318]
```

0s completed at 10:41 AM

## 5) 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:**

Welcome To Colaboratory - CSE DS-DL COMPUTERDL Pract - Google DriveSGD for Single NNUntitled1.ipynb - Colaboratory41\_Uivashi\_DL\_Exp-3 - Google Drive

colab.research.google.com/drive/1tpUSlIt35mv972hLvPx4EH1gmG8QHYUQ#scrollTo=nSLhwAj1bfPo

Untitled1.ipynb

File Edit View Insert Runtime Tools Help All changes saved

CommentShareSettingsU

+ Code + Text

0s

Epoch 1/11, Loss: 20.0208  
Epoch 2/11, Loss: 18.3632  
Epoch 3/11, Loss: 19.9461  
Epoch 4/11, Loss: 16.3753  
Epoch 5/11, Loss: 17.3140  
Epoch 6/11, Loss: 18.4489  
Epoch 7/11, Loss: 16.9884  
Epoch 8/11, Loss: 16.2863  
Epoch 9/11, Loss: 16.5538  
Epoch 10/11, Loss: 15.4883  
Epoch 11/11, Loss: 14.8409  
Trained Weights: [[2.33682509]]  
Trained Biases: [0.67278367]

RAM  
Disk

completed at 10:48 AM

83°F High winds soon10:52 AM 8/7/2023