Subject: CSC701 – Deep Learning

Name:	JIDNYASA PATIL
Roll No:	43
Class:	BE/CSE(DS)
Assignment No:	03
Date of Performance:	
Date of Submission:	
Marks:	
Sign of Faculty:	

Experiment No. 03

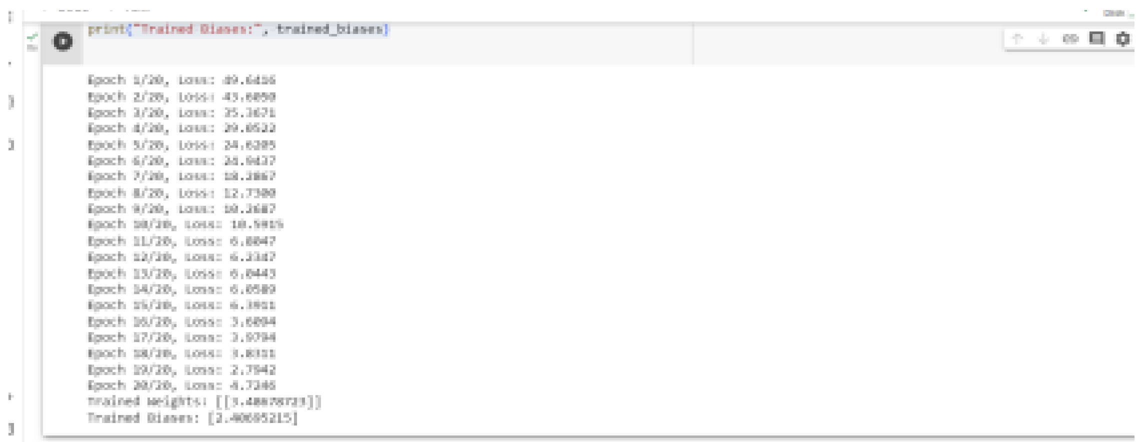
1)Stochastic Gradient Descent

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:



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:

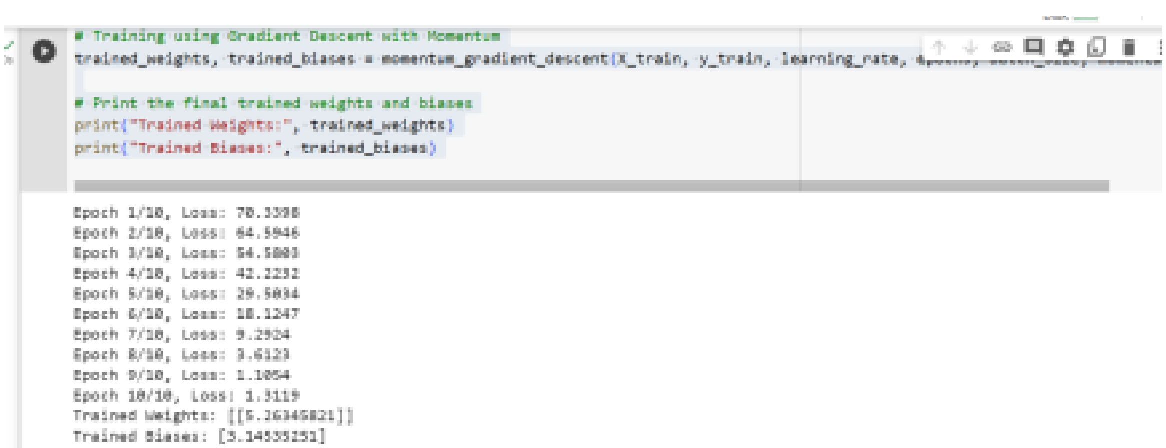
3) Momentum GD

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,

# 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: 54.5083
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) 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:

```
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: 7.1980
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]
```

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:

```
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: 16.5530
Epoch 10/11, Loss: 15.4883
Epoch 11/11, Loss: 14.8409
Trained Weights: [[2.33682509]]
Trained Biases: [0.67278367]
```