Experiment no: 3

Aim: Implement Mini-Batch Gradient Descent in TensorFlow.

Apparatus:

- Laptop/Desktop
- Python (3.x)
- TensorFlow
- NumPy
- Matplotlib
- Jupyter Notebook / Google Colab / Any Python IDE

Theory:

Gradient Descent is the cornerstone algorithm in training neural networks. It is an iterative optimization algorithm used to minimize a loss function by updating the model parameters (weights and biases) in the direction of the steepest descent (negative gradient).

Type

- Batch Gradient Descent
- Stochastic Gradient Descent (SGD)
- Mini-Batch Gradient Descent

Mini-Batch Gradient Descent

In practice, Mini-Batch Gradient Descent is preferred because:

- It speeds up training compared to full batch.
- It reduces the noise present in SGD.
- It makes use of vectorized operations for better performance on GPUs.
- It improves generalization due to exposure to diverse batches.

Mathematical Representation:

Let the loss function be $J(\theta)$, and θ be the parameters (weights) of the model.

For each mini-batch $B \subset D$, update rule:

$\theta := \theta - \eta \cdot \nabla JB(\theta)$

Where:

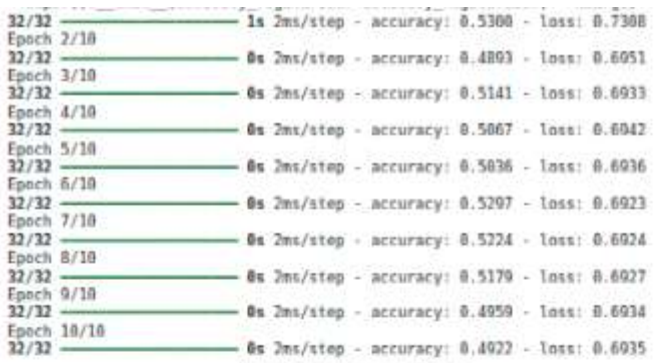
- η is the learning rate,

- ∇JB(θ) is the gradient computed on mini-batch B.

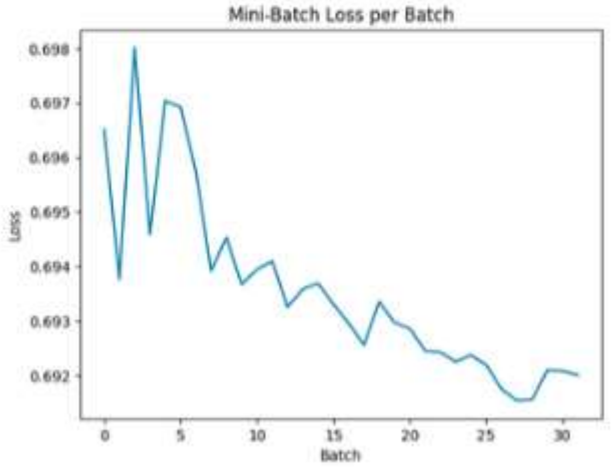
Code:

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np
import matplotlib.pyplot as plt # corrected 'pylot' to 'pyplot'

# Generate random data
X_train = np.random.rand(1000, 2)
y_train = np.random.randint(0, 2, size=(1000, 1))

# Define the model
model = models.Sequential([
layers.Dense(64, activation='relu', input_shape=(2,)), # changed input shape to (2,) because X_train has shape
(1000, 2)
layers.Dense(64, activation='sigmoid'),
layers.Dense(1, activation='sigmoid') # added an output layer for binary classification
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy']) # fixed typo in
'binary_crossentropy'

# Custom callback to store batch losses
class BatchLossHistory(tf.keras.callbacks.Callback): # fixed typo in 'callbacks'
def on_train_batch_end(self, batch, logs=None):
if not hasattr(self, 'losses'):
self.losses = []
self.losses.append(logs['loss'])

def on_epoch_begin(self, epoch, logs=None):
self.losses = [] # Reset losses at the beginning of each epoch

# Instantiate the custom callback
batch_loss_history = BatchLossHistory()

# Train the model and store losses
history = model.fit(X_train, y_train, epochs=10, batch_size=32, callbacks=[batch_loss_history])

# Plot the mini-batch loss per batch
plt.plot(batch_loss_history.losses)
plt.xlabel('Batch')
plt.ylabel('Loss')
plt.title('Mini-Batch Loss per Batch')
plt.show()
```

Output:



```
32/32 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.5300 - loss: 0.7308
Epoch 2/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4893 - loss: 0.6951
Epoch 3/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5141 - loss: 0.6933
Epoch 4/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5067 - loss: 0.6942
Epoch 5/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5036 - loss: 0.6936
Epoch 6/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5297 - loss: 0.6923
Epoch 7/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5224 - loss: 0.6924
Epoch 8/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5179 - loss: 0.6927
Epoch 9/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4959 - loss: 0.6934
Epoch 10/10
32/32 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.4922 - loss: 0.6935
```



Conclusion:

We successfully implemented and visualized Mini-Batch Gradient Descent using TensorFlow. This method provided an efficient and effective way to optimize our model, combining the advantages of batch and stochastic gradient descent.