60009220202

Ayushi singh

D025

Step 1: Import Libraries

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import (
    SGD, RMSprop, Adagrad, Adadelta, Adam
)

import matplotlib.pyplot as plt
```

Step 2: Load and Prepare the MNIST Dataset

```

(x_train, y_train), (x_test, y_test) = mnist.load_data()


x_train = x_train.astype('float32') / 255
x_test = x_test.astype('float32') / 255


y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

Step 3: Initialize a Simple Neural Network Model

```
def create_model():
    model = Sequential([
        Flatten(input_shape=(28, 28)),
        Dense(128, activation='relu'),
        Dense(10, activation='softmax')
    ])
    return model
```

Step 4: Compile and Train the Model with Different Optimizers

```
optimizers = {
    'SGD': SGD(),
    'SGD with Momentum': SGD(momentum=0.9),
    'Mini-Batch SGD': SGD(),
    'Adagrad': Adagrad(),
    'RMSProp': RMSprop(),
    'Adadelta': Adadelta(),
    'Adam': Adam()
}

results = {}

for opt_name, optimizer in optimizers.items():
    model = create_model()
    model.compile(optimizer=optimizer,
loss='categorical_crossentropy', metrics=['accuracy'])
    history = model.fit(x_train, y_train, epochs=10, batch_size=32,
validation_data=(x_test, y_test), verbose=0)
    results[opt_name] = history.history['val_accuracy']
    print(f"{opt_name} final validation accuracy:
{history.history['val_accuracy'][-1]:.4f}")
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/reshaping/
flatten.py:37: UserWarning: Do not pass an `input_shape`/`input_dim`
argument to a layer. When using Sequential models, prefer using an
`Input(shape)` object as the first layer in the model instead.
  super().__init__(**kwargs)
```

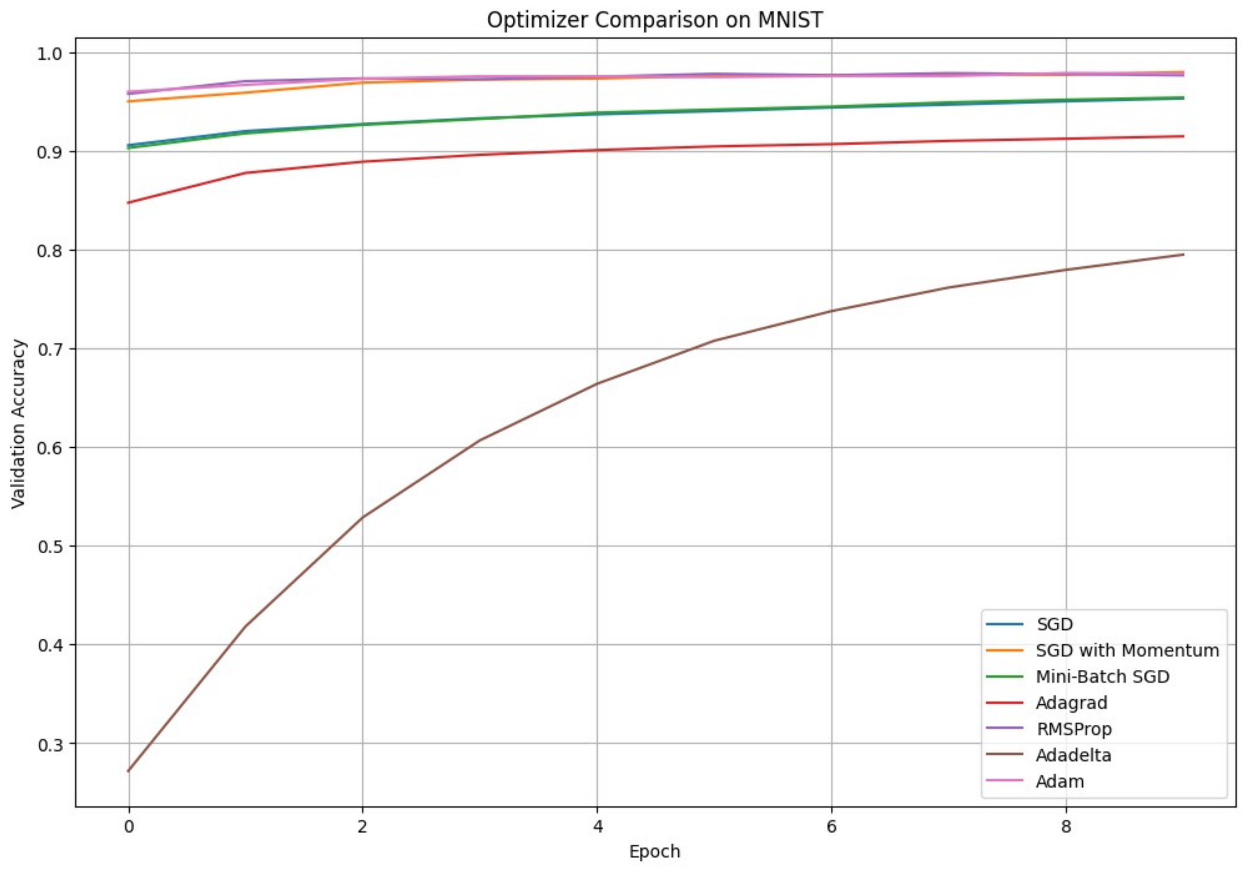
```
SGD final validation accuracy: 0.9529
SGD with Momentum final validation accuracy: 0.9796
Mini-Batch SGD final validation accuracy: 0.9538
Adagrad final validation accuracy: 0.9145
```

```
RMSProp final validation accuracy: 0.9763
Adadelta final validation accuracy: 0.7947
Adam final validation accuracy: 0.9782
```

Step 5: Plot the Results

```
plt.figure(figsize=(12, 8))
for opt_name, val_acc in results.items():
    plt.plot(val_acc, label=opt_name)

plt.title('Optimizer Comparison on MNIST')
plt.xlabel('Epoch')
plt.ylabel('Validation Accuracy')
plt.legend()
plt.grid(True)
plt.show()
```



This code trains a simple neural network on the MNIST dataset using different optimizers while keeping the batch size and number of epochs the same. The final validation accuracy for each optimizer is printed, and a comparison of validation accuracy over epochs is plotted.

Task (e): Advantages and Disadvantages of Each Optimizer

1. Gradient Descent

Advantages: Simple and easy to implement. Suitable for convex problems where it can converge to the global minimum. Disadvantages: Slow convergence for large datasets. Can get stuck in local minima in non-convex problems. Requires fine-tuning of the learning rate.

1. Stochastic Gradient Descent (SGD)

Advantages: Faster convergence as it updates weights more frequently. Can escape local minima due to its stochastic nature. Disadvantages: High variance in updates can lead to fluctuations in the optimization path. May require a smaller learning rate to stabilize, slowing down training.

1. Stochastic Gradient Descent with Momentum

Advantages: Accelerates convergence and reduces fluctuations by adding momentum to updates. Helps in navigating ravines in the error surface, leading to faster convergence. Disadvantages: Requires tuning of an additional hyperparameter (momentum). Can overshoot the minimum if momentum is too high.

1. Mini-Batch Gradient Descent

Advantages: Combines the advantages of both SGD and batch gradient descent. Reduces variance in updates compared to SGD, leading to more stable convergence. Efficient use of computational resources. Disadvantages: Requires selecting an optimal batch size, which can be challenging. Still sensitive to the learning rate and can oscillate around the minimum.

1. Adagrad

Advantages: Adaptive learning rates that improve performance on sparse data. No need to manually tune the learning rate during training. Disadvantages: Learning rate can become very small over time, leading to premature convergence. Accumulation of squared gradients causes slow convergence in later stages.

1. RMSProp

Advantages: Adaptive learning rate that mitigates the diminishing learning rate issue of Adagrad. Effective for non-stationary and online learning problems. Disadvantages: Sensitive to the choice of hyperparameters, especially the decay rate. Requires careful tuning of learning rate and decay factor.

1. Adadelta

Advantages: Overcomes the learning rate decay issue of Adagrad by limiting the window of accumulated gradients. Suitable for training deep networks with complex architectures. Disadvantages: More complex implementation compared to simpler optimizers. Requires careful tuning of hyperparameters for optimal performance.

1. Adam

Advantages: Combines the benefits of both RMSProp and momentum, leading to faster and more stable convergence. Well-suited for a wide range of problems and often works well out-of-the-box. Adaptive learning rate and momentum make it robust to noisy gradients.

Disadvantages: Requires tuning of multiple hyperparameters (learning rate, beta1, beta2). May sometimes converge to suboptimal minima compared to simpler optimizers. This completes the theoretical analysis and practical evaluation of the performance of different optimizers on a neural network using the MNIST dataset.