Deep Learning With Tensor Flow 1 (CSE 3793)

ASSIGNMENT-1: INTRODUCTION TO NEURAL NETWORK

Name:Tribhuwan Singh

Reg. No.: 2341019538

Section:23412C3

Write a Python code to build a feed-forward neural network for AND, OR, and NAND with a singlelayer perceptron from scratch.

```
#Q1
#perceptron for AND gate
import numpy as np
def step_function(z):
    return 1 if z>=0 else 0
def perceptron_train(X, y, learning_rate=0.1, epochs=5):
    weights = np.zeros (X.shape[1])
    bias=0
    for j in range(epochs):
        print ("Epochs:",j)
        for i in range(len(X)):
            z=np.dot(weights,X[i])+bias
            print("Z=" , z)
            y_pred=step_function (z)
            print("predicted",y_pred,end=' ')
            error= y[i]-y_pred
            print ("error",error,end=' ')
            weights+= learning_rate*error*X[i]
            bias+= learning_rate * error
            print(f"weights: {weights}, Bias: {bias}", end=' ')
        print()
    return weights ,bias
X= np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([0,0,0,1])
weights, bias= perceptron_train(X,y)
def perceptron_predict(X,weights,bias):
    return[step_function (np.dot(weights,x)+bias)for x in X]
outputs= perceptron_predict(X,weights, bias)
print("Final predictions", outputs)
```

```
Epochs: 0
Z= 0.0
predicted 1 error -1 weights: [0. 0.], Bias: -0.1 Z= -0.1
predicted 0 error 0 weights: [0. 0.], Bias: -0.1 Z= -0.1
predicted 0 error 0 weights: [0. 0.], Bias: -0.1 Z= -0.1
predicted 0 error 1 weights: [0.1 0.1], Bias: 0.0
Epochs: 1
Z= 0.0
predicted 1 error -1 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error -1 weights: [0.1 0. ], Bias: -0.2 Z= -0.1
predicted 0 error 0 weights: [0.1 0. ], Bias: -0.2 Z= -0.1
predicted 0 error 1 weights: [0.2 0.1], Bias: -0.1
Epochs: 2
Z= -0.1
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.1 Z= 0.0
predicted 1 error -1 weights: [0.2 0. ], Bias: -0.2 Z= 0.0
predicted 1 error -1 weights: [0.1 0. ], Bias: -0.30000000000000004 Z= -0.20000000000000004
predicted 0 error 1 weights: [0.2 0.1], Bias: -0.20000000000000004
Epochs: 3
Z= -0.20000000000000004
```

```
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= -0.10000000000000003
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= -2.7755575615628914e-17
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= 0.1
predicted 1 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004
Epochs: 4
Z= -0.20000000000000004
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= -0.10000000000000003
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= -2.7755575615628914e-17
predicted 0 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004 Z= 0.1
predicted 1 error 0 weights: [0.2 0.1], Bias: -0.20000000000000004
Final predictions [0, 0, 0, 1]
```

```
#perceptron for OR gate
import numpy as np
def step_function(z):
    return 1 if z>=0 else 0
def perceptron_train(X, y, learning_rate=0.1, epochs=5):
    weights = np.zeros (X.shape[1])
    bias=0
    for j in range(epochs):
        print ("Epochs:",j)
        for i in range(len(X)):
            z=np.dot(weights,X[i])+bias
            print("Z=" , z)
            y_pred=step_function (z)
            print("predicted",y_pred,end=' ')
            error= y[i]-y_pred
            print ("error",error,end=' ')
            weights+= learning_rate*error*X[i]
            bias+= learning_rate * error
            print(f"weights: {weights}, Bias: {bias}", end=' ')
        print()
    return weights ,bias
X= np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([0,1,1,1])
weights, bias= perceptron_train(X,y)
def perceptron_predict(X,weights,bias):
    return[step_function (np.dot(weights,x)+bias)for x in X]
outputs= perceptron_predict(X,weights, bias)
print("Final predictions", outputs)
```

```
Epochs: 0
Z= 0.0
predicted 1 error -1 weights: [0. 0.], Bias: -0.1 Z= -0.1
predicted 0 error 1 weights: [0.  0.1], Bias: 0.0 Z= 0.0
predicted 1 error 0 weights: [0.  0.1], Bias: 0.0 Z= 0.1
predicted 1 error 0 weights: [0.  0.1], Bias: 0.0
Epochs: 1
Z= 0.0
predicted 1 error -1 weights: [0.  0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.  0.1], Bias: -0.1 Z= -0.1
predicted 0 error 1 weights: [0.1 0.1], Bias: 0.0 Z= 0.2
predicted 1 error 0 weights: [0.1 0.1], Bias: 0.0
Epochs: 2
Z= 0.0
predicted 1 error -1 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.1
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1
Epochs: 3
Z= -0.1
predicted 0 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.1
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1
Epochs: 4
Z= -0.1
predicted 0 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.0
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1 Z= 0.1
predicted 1 error 0 weights: [0.1 0.1], Bias: -0.1
Final predictions [0, 1, 1, 1]
```

```
#perceptron for NAND gate
import numpy as np
def step_function(z):
    return 1 if z>=0 else 0
def perceptron_train(X, y, learning_rate=0.1, epochs=20):
    weights = np.zeros (X.shape[1])
    bias=0
    for j in range(epochs):
        print ("Epochs:",j)
        for i in range(len(X)):
            z=np.dot(weights,X[i])+bias
            print("Z=" , z)
            y_pred=step_function (z)
            print("predicted",y_pred,end=' ')
            error= y[i]-y_pred
            print ("error",error,end=' ')
            weights+= learning_rate*error*X[i]
            bias+= learning_rate * error
            print(f"weights: {weights}, Bias: {bias}", end=' ')
        print()
    return weights ,bias
X= np.array([[0,0],[0,1],[1,0],[1,1]])
y=np.array([1,1,1,0])
weights, bias= perceptron_train(X,y)
def perceptron_predict(X,weights,bias):
    return[step_function (np.dot(weights,x)+bias)for x in X]
outputs= perceptron_predict(X,weights, bias)
print("Final predictions", outputs)
```

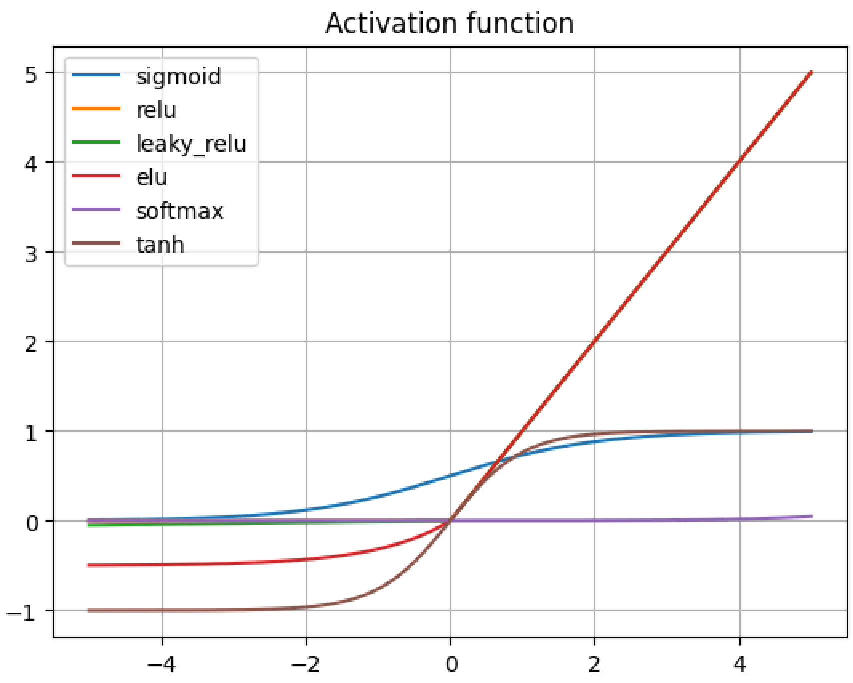
```
Epochs: 0
Z= 0.0
predicted 1 error 0 weights: [0. 0.], Bias: 0.0 Z= 0.0
predicted 1 error 0 weights: [0. 0.], Bias: 0.0 Z= 0.0
predicted 1 error 0 weights: [0. 0.], Bias: 0.0 Z= 0.0
predicted 1 error -1 weights: [-0.1 -0.1], Bias: -0.1
Epochs: 1
Z= -0.1
predicted 0 error 1 weights: [-0.1 -0.1], Bias: 0.0 Z= -0.1
predicted 0 error 1 weights: [-0.1  0. ], Bias: 0.1 Z= 0.0
predicted 1 error 0 weights: [-0.1  0. ], Bias: 0.1 Z= 0.0
predicted 1 error -1 weights: [-0.2 -0.1], Bias: 0.0
Epochs: 2
Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.0 Z= -0.1
predicted 0 error 1 weights: [-0.2  0. ], Bias: 0.1 Z= -0.1
predicted 0 error 1 weights: [-0.1  0. ], Bias: 0.2 Z= 0.1
predicted 1 error -1 weights: [-0.2 -0.1], Bias: 0.1
Epochs: 3
Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.1 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.1 Z= -0.1
predicted 0 error 1 weights: [-0.1 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error -1 weights: [-0.2 -0.2], Bias: 0.1
Epochs: 4
Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.2], Bias: 0.1 Z= -0.1
predicted 0 error 1 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= -0.10000000000000003
predicted 0 error 0 weights: [-0.2 -0.1], Bias: 0.2
Epochs: 5
Z= 0.2
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= -0.10000000000000003
predicted 0 error 0 weights: [-0.2 -0.1], Bias: 0.2
Epochs: 6
Z= 0.2
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= -0.10000000000000003
predicted 0 error 0 weights: [-0.2 -0.1], Bias: 0.2
```

```
Epochs: 7
Z= 0.2
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= -0.10000000000000003
predicted 0 error 0 weights: [-0.2 -0.1], Bias: 0.2
Epochs: 8
Z= 0.2
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= -0.10000000000000003
predicted 0 error 0 weights: [-0.2 -0.1], Bias: 0.2
Epochs: 9
Z= 0.2
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.1
predicted 1 error 0 weights: [-0.2 -0.1], Bias: 0.2 Z= 0.0
```

2.write a python code to show different activation function?

```
import numpy as np
import matplotlib.pyplot as plt
x= np.linspace(-5,5,200)
sigmoid = 1/(1+np.exp(-x)) #sigmoid
relu=np.maximum(0,x) #relu
leaky_relu=np.where(x>0,x,0.01*x) #alpha = 0.01
elu = np.where(x>0,x,0.5*(np.exp(x)-1)) #alpha=0.5
softmax=np.exp(x)/np.sum(np.exp(x)) #softmax
tanh=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
plt.plot(x,sigmoid,label="sigmoid")
plt.legend()
plt.grid()
plt.title("Activation function")
plt.plot(x,relu,label="relu")
plt.legend()
plt.plot(x,leaky_relu,label="leaky_relu")
plt.legend()
plt.plot(x,elu,label="elu")
plt.legend()
plt.plot(x,softmax,label="softmax")
plt.legend()
plt.plot(x,tanh,label="tanh")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7de28364bdd0>
```



3. Write a Python code to implement the binary cross-entropy loss function from scratch.

```
def bce(actual,predicted,eps=1e-9): #3
    predicted = np.clip(predicted,eps,1-eps)
    return -(actual*np.log(predicted)+(1-actual)*np.log(1-predicted))
actual =np.array([0,1,1,0])
predicted = np.array([0.1,0.9,0.8,0.7])
print(bce(actual,predicted))
```

```
[0.10536052 0.10536052 0.22314355 1.2039728 ]
```

4. Write a Python code to implement the Stochastic Gradient Descent Optimization technique.

```
import numpy as np

def sgd(f, gradient, initial_params, learning_rate, epochs):

    params = np.copy(initial_params)
    for epoch in range(epochs):

        grad = gradient(params)
        params -= learning_rate * grad
        if epoch % 100 == 0:
            print(f"Epoch {epoch}: Loss = {f(params)}")
    return params

def objective_function(x):
    return x**2

def gradient_function(x):
    return 2 * x

initial_parameters = np.array([5.0])
learning_rate = 0.01
epochs = 1000

optimized_parameters = sgd(objective_function, gradient_function, initial_parameters, learning_rate, e
print(f"\nOptimized parameters: {optimized_parameters}")
print(f"Minimum value of the function: {objective_function(optimized_parameters)}")
```

```
Epoch 0: Loss = [24.01]
Epoch 100: Loss = [0.4222866]
Epoch 200: Loss = [0.00742715]
Epoch 300: Loss = [0.00013063]
Epoch 400: Loss = [2.29748516e-06]
Epoch 500: Loss = [4.04080462e-08]
Epoch 600: Loss = [7.1069456e-10]
Epoch 700: Loss = [1.2499658e-11]
Epoch 800: Loss = [2.19843317e-13]
Epoch 900: Loss = [3.86659252e-15]

Optimized parameters: [8.41483679e-09]
Minimum value of the function: [7.08094781e-17]
```

5. Build a multilayer perceptron from scratch using numpy and compare it with Keras

```
import numpy as np
def sigmoid(z): return 1/ (1+np.exp(-z))
def forward_sigmoid(X,w1,b1,w2,b2):
  z1= X @ w1+b1
  a1= sigmoid(z1)
  z2= X @ w2+b2
  a2= sigmoid(z2)
  return(a2>0.5).astype(int)
X= np.array([[0,0],[1,0],[0,1],[1,1]])
y= np.array([[0],[1],[1],[0]])
w1= np.array([[20,20],[20,20]])
```

```
b1= np.array([[-10,30]])
w2= np.array([[20],[-20]])
b2= np.array([[-10]])
print ("Sigmoid NN Predictions:", forward_sigmoid(X,w1,b1,w2,b2))
```

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# Input dataset (XOR inputs)
X = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])
# Output dataset (XOR outputs)
y = np.array([[0],
              [1],
              [1],
              [0]])
np.random.seed(42)
input_layer_neurons = X.shape[1]
hidden_layer_neurons = 2
output_neurons = 1
# Weights and biases
W1 = np.random.uniform(size=(input_layer_neurons,
hidden_layer_neurons))
b1 = np.random.uniform(size=(1, hidden_layer_neurons))
W2 = np.random.uniform(size=(hidden_layer_neurons,
output_neurons))
b2 = np.random.uniform(size=(1, output_neurons))
# Learning rate
lr = 0.1
# Number of epochs
epochs = 10000
for epoch in range(epochs):
  # Forward propagation
  hidden_input = np.dot(X, W1) + b1
  hidden_output = sigmoid(hidden_input)
  final_input = np.dot  (hidden_output, W2) + b2
  y_pred = sigmoid(final_input)
  # Compute Error
  error = y - y_pred
  # Backpropagation
  d_y_pred = error * sigmoid_derivative(y_pred)
  error_hidden_layer = d_y_pred.dot(W2.T)
  d_hidden_layer = error_hidden_layer * sigmoid_derivative(hidden_output)

# Update weights and biases
  W2 += hidden_output.T.dot(d_y_pred) * lr
  b2 += np.sum(d_y_pred, axis=0, keepdims=True) * lr
  W1 += X.T.dot(d_hidden_layer) * lr
  b1 += np.sum(d_hidden_layer, axis=0, keepdims=True) * lr

# Print loss occasionally
  if epoch % 1000 == 0:
    loss = np.mean(np.square(error))
    print(f"Epoch {epoch}, Loss: {loss:.6f}")

# -------- Final results --------
print("\nFinal predictions after training:")
print(y_pred.round(3))   # Rounded for clarity
```

```
Epoch 0, Loss: 0.324659
Epoch 1000, Loss: 0.240589
Epoch 2000, Loss: 0.196030
```

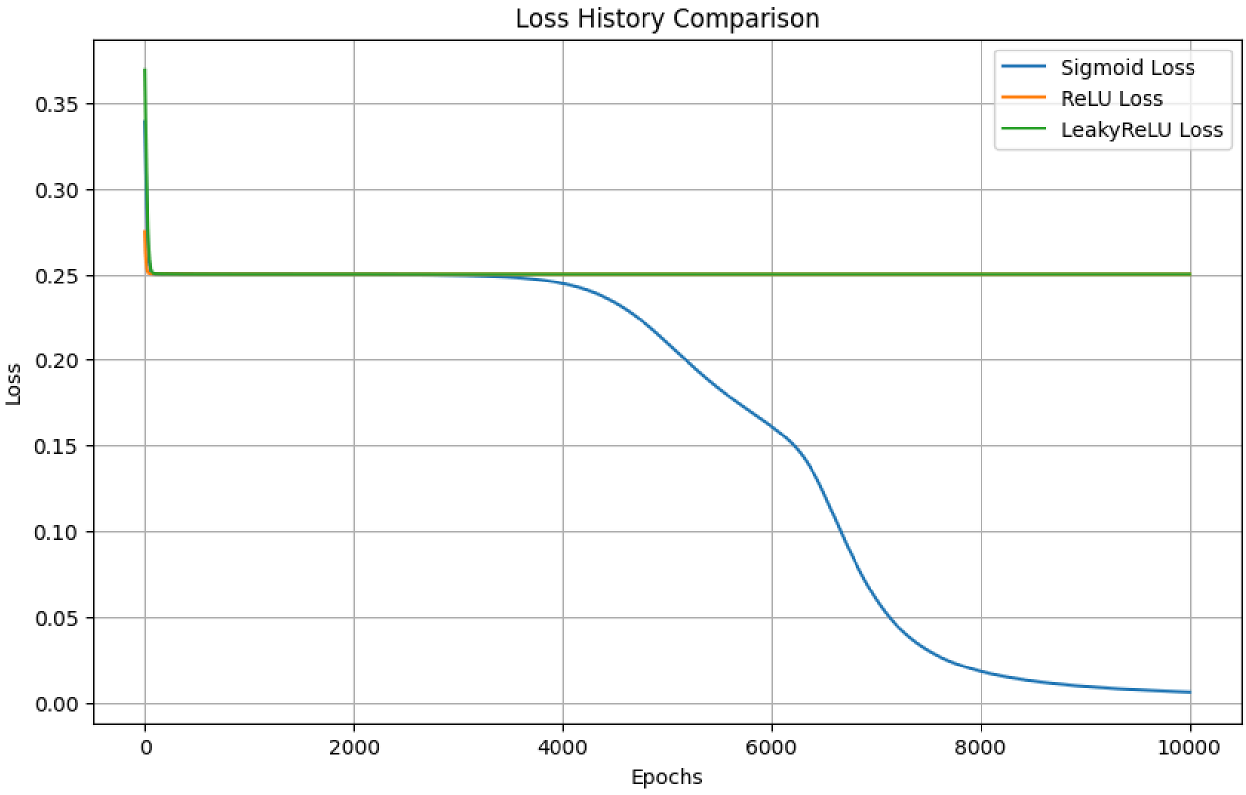
```
Epoch 3000, Loss: 0.120663
Epoch 4000, Loss: 0.030459
Epoch 5000, Loss: 0.012541
Epoch 6000, Loss: 0.007368
Epoch 7000, Loss: 0.005093
Epoch 8000, Loss: 0.003847
Epoch 9000, Loss: 0.003071

Final predictions after training:
[[0.053]
 [0.952]
 [0.952]
 [0.052]]
```

6. Write a Python code to classify the XOR function and use sigmoid, ReLU, and LeakyReLU in the hidden layer of a neural network. Discuss the accuracy of different activation functions used in the hidden layer. Also, show the comparative analysis in a graph.

```
import numpy as np
import matplotlib.pyplot as plt

# XOR dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Sigmoid activation function and its derivative
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

# ReLU activation function and its derivative
def relu(x):
    return np.maximum(0, x)

def relu_derivative(x):
    return (x > 0).astype(float)

# LeakyReLU activation function and its derivative
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def leaky_relu_derivative(x, alpha=0.01):
    return np.where(x > 0, 1, alpha)

# Neural network training function
def train_nn(X, y, activation_func, activation_derivative, epochs=10000, learning_rate=0.1):
    input_layer_neurons = X.shape[1]
    hidden_layer_neurons = 2
    output_neurons = 1

    # Initialize weights and biases
    W1 = np.random.uniform(size=(input_layer_neurons, hidden_layer_neurons))
    b1 = np.random.uniform(size=(1, hidden_layer_neurons))
    W2 = np.random.uniform(size=(hidden_layer_neurons, output_neurons))
    b2 = np.random.uniform(size=(1, output_neurons))

    loss_history = []

    for epoch in range(epochs):
        # Forward propagation
        hidden_layer_input = np.dot(X, W1) + b1
        hidden_layer_output = activation_func(hidden_layer_input)
        output_layer_input = np.dot(hidden_layer_output, W2) + b2
        predicted_output = sigmoid(output_layer_input) # Using sigmoid for the output layer
```

```
        # Backpropagation
        error = y - predicted_output
        d_predicted_output = error * sigmoid_derivative(predicted_output)

        error_hidden_layer = d_predicted_output.dot(W2.T)
        d_hidden_layer = error_hidden_layer * activation_derivative(hidden_layer_output)

        # Update weights and biases
        W2 += hidden_layer_output.T.dot(d_predicted_output) * learning_rate
        b2 += np.sum(d_predicted_output, axis=0, keepdims=True) * learning_rate
        W1 += X.T.dot(d_hidden_layer) * learning_rate
        b1 += np.sum(d_hidden_layer, axis=0, keepdims=True) * learning_rate

        loss = np.mean(np.square(error))
        loss_history.append(loss)

    return predicted_output, loss_history

# Train with different activation functions and store accuracies and loss histories
results = {}
activations = {
    "Sigmoid": (sigmoid, sigmoid_derivative),
    "ReLU": (relu, relu_derivative),
    "LeakyReLU": (leaky_relu, leaky_relu_derivative)
}

for name, (func, derivative) in activations.items():
    print(f"Training with {name} activation...")
    predicted_output, loss_history = train_nn(X, y, func, derivative)
    accuracy = np.mean((predicted_output.round() == y).astype(int))
    results[name] = {"accuracy": accuracy, "loss_history": loss_history}
    print(f"Accuracy with {name}: {accuracy:.4f}\n")

# Discuss the accuracy
print("Comparative Analysis of Activation Functions:")
for name, data in results.items():
    print(f"{name} Accuracy: {data['accuracy']:.4f}")

# Plot the loss history
plt.figure(figsize=(10, 6))
for name, data in results.items():
    plt.plot(data['loss_history'], label=f'{name} Loss')
plt.title('Loss History Comparison')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)
plt.show()
```

```
Training with Sigmoid activation...
Accuracy with Sigmoid: 1.0000

Training with ReLU activation...
Accuracy with ReLU: 0.5000

Training with LeakyReLU activation...
Accuracy with LeakyReLU: 0.5000

Comparative Analysis of Activation Functions:
Sigmoid Accuracy: 1.0000
ReLU Accuracy: 0.5000
LeakyReLU Accuracy: 0.5000
```



7. Write code to define a simple neural network in TensorFlow. Use the MNIST dataset, reshape the data to 610,000 x 784, normalize the input, and build and train the model using the SGD optimizer and categorical cross-entropy as the loss function to compute the accuracy. Also, find the test accuracy.

```
import tensorflow as tf
import numpy as np
from tensorflow import keras
EPOCHS = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES =10
N_HIDDEN= 128
VALIDATION_SPLIT = 0.2
#load dataset
mnist = keras.datasets.mnist
(X_train,y_train),(X_test,y_test) = mnist.load_data()
#MNIST  dataset =60,000 trainng images + 10,000 test images
RESHAPED = 784
X_train = X_train.reshape(60000,RESHAPED)
X_test = X_test.reshape(10000,RESHAPED)
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
#convert pixel values to floats betm 0 and 1
y_train = tf.keras.utils.to_categorical(y_train,NB_CLASSES)
```

```
y_test = tf.keras.utils.to_categorical(y_test,NB_CLASSES)
#convert lables into one hot encoding
#model architecture
model = keras.Sequential([
    keras.layers.Input(shape=(RESHAPED,)),
    keras.layers.Dense(N_HIDDEN,activation='relu',kernel_initializer = 'he_normal'),


    #keras.layer.Dropout(0.2),
    keras.layers.Dense(N_HIDDEN//2,activation='relu',
    kernel_initializer='he_normal'),

    #keras.layers.Dropout(0.2),
    keras.layers.Dense(NB_CLASSES,activation='softmax')
])
#input = 784 features
#model compilation
model.compile(Optimizer= keras.optimizer.SG,loss='categorical_crossentropy',metrics=['accuracy'])
#model training
#es = keras.callbacks.EarlyStopping(monitor='val_loss',mode='min',verbose=1,patience=5)
#rs =keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor = 0.5, mode='min',verbose=1,patience
history = model.fit(X_train,y_train,batch_size=BATCH_SIZE,epochs=EPOCHS,verbose=VERBOSE,validation_spl
#model evaluation
test_loss,test_acc = model.evaluate(X_test,y_test)
print('Test accuracy:',test_acc,'Test')
```

```
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 9s 13ms/step - accuracy: 0.3380 - loss: 1.9306 - val_accuracy: 0.8343 - v
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7286 - loss: 0.8831 - val_accuracy: 0.8815 - va
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8000 - loss: 0.6579 - val_accuracy: 0.8953 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8289 - loss: 0.5569 - val_accuracy: 0.9027 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8488 - loss: 0.5028 - val_accuracy: 0.9086 - va
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8616 - loss: 0.4621 - val_accuracy: 0.9142 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8716 - loss: 0.4262 - val_accuracy: 0.9178 - va
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8806 - loss: 0.4008 - val_accuracy: 0.9218 - va
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8837 - loss: 0.3908 - val_accuracy: 0.9248 - va
Epoch 10/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.8870 - loss: 0.3717 - val_accuracy: 0.9268 - va
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8953 - loss: 0.3524 - val_accuracy: 0.9300 - va
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9009 - loss: 0.3403 - val_accuracy: 0.9326 - va
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9031 - loss: 0.3299 - val_accuracy: 0.9349 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9084 - loss: 0.3109 - val_accuracy: 0.9368 - va
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9058 - loss: 0.3158 - val_accuracy: 0.9385 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9084 - loss: 0.3065 - val_accuracy: 0.9406 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9151 - loss: 0.2911 - val_accuracy: 0.9417 - va
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9176 - loss: 0.2848 - val_accuracy: 0.9420 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9208 - loss: 0.2733 - val_accuracy: 0.9435 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9213 - loss: 0.2688 - val_accuracy: 0.9458 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9233 - loss: 0.2578 - val_accuracy: 0.9456 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9263 - loss: 0.2541 - val_accuracy: 0.9481 - va
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9257 - loss: 0.2511 - val_accuracy: 0.9490 - va
Epoch 24/200
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9268 - loss: 0.2482 - val_accuracy: 0.9493 - va
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9300 - loss: 0.2366 - val_accuracy: 0.9502 - va
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9299 - loss: 0.2398 - val_accuracy: 0.9513 - va
Epoch 27/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9338 - loss: 0.2229 - val_accuracy: 0.9519 - va
Epoch 28/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.9338 - loss: 0.2210 - val_accuracy: 0.9534 - va
Epoch 29/200
```

8 Train the MLP as specified in Q7 with regularization (dropout and L2), different optimizers, and discuss a detailed comparative analysis

```
# dropout l2
import tensorflow as tf
import numpy as np
from tensorflow import keras
from tensorflow.keras.regularizers import l2

# Set hyperparameters
EPOCHS = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10
N_HIDDEN = 128
VALIDATION_SPLIT = 0.2
LAMBDA = 0.001 # L2 regularization factor

# Load and preprocess data
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)
Y_test = tf.keras.utils.to_categorical(Y_test, NB_CLASSES)

# Build the neural network model with L2 regularization
model_l2 = keras.Sequential([
    keras.layers.Input(shape=(RESHAPED,)),
    keras.layers.Dense(N_HIDDEN, activation=tf.keras.layers.LeakyReLU(alpha=0.01), kernel_regularizer=
    keras.layers.Dense(N_HIDDEN // 2, activation=tf.keras.layers.LeakyReLU(alpha=0.01), kernel_regular
    keras.layers.Dense(NB_CLASSES, activation='softmax')
])

model_l2.compile(
    optimizer=keras.optimizers.SGD(learning_rate=1e-3),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

history_l2 = model_l2.fit(
    X_train, Y_train,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_split=VALIDATION_SPLIT,
    verbose=VERBOSE,
)

test_loss_l2, test_acc_l2 = model_l2.evaluate(X_test, Y_test, verbose=1)
print(f"\nModel with L2 Regularization:")
print(f"Test accuracy: {test_acc_l2:.4f}  | Test loss: {test_loss_l2:.4f}")

y_true_l2 = np.argmax(Y_test, axis=1)
```

```
y_pred_l2 = np.argmax(model_l2.predict(X_test, verbose=0), axis=1)
cm_l2 = tf.math.confusion_matrix(y_true_l2, y_pred_l2, num_classes=NB_CLASSES).numpy()
print("\nConfusion Matrix (L2):")
print(cm_l2)
```

```
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.1415 - loss: 2.6435 - val_accuracy: 0.3410 - va
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.3838 - loss: 2.3637 - val_accuracy: 0.5042 - va
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.5203 - loss: 2.1421 - val_accuracy: 0.6153 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6225 - loss: 1.9128 - val_accuracy: 0.6978 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6911 - loss: 1.6991 - val_accuracy: 0.7529 - va
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7436 - loss: 1.5054 - val_accuracy: 0.7905 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7801 - loss: 1.3472 - val_accuracy: 0.8161 - va
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8007 - loss: 1.2246 - val_accuracy: 0.8306 - va
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8205 - loss: 1.1257 - val_accuracy: 0.8410 - va
Epoch 10/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.8290 - loss: 1.0587 - val_accuracy: 0.8512 - va
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8365 - loss: 1.0039 - val_accuracy: 0.8577 - va
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8482 - loss: 0.9517 - val_accuracy: 0.8624 - va
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8512 - loss: 0.9198 - val_accuracy: 0.8678 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8562 - loss: 0.8899 - val_accuracy: 0.8720 - va
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.8632 - loss: 0.8560 - val_accuracy: 0.8749 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8653 - loss: 0.8365 - val_accuracy: 0.8774 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8724 - loss: 0.8157 - val_accuracy: 0.8808 - va
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8719 - loss: 0.8015 - val_accuracy: 0.8839 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8745 - loss: 0.7925 - val_accuracy: 0.8863 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8764 - loss: 0.7814 - val_accuracy: 0.8885 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8816 - loss: 0.7655 - val_accuracy: 0.8898 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8806 - loss: 0.7583 - val_accuracy: 0.8912 - va
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8828 - loss: 0.7453 - val_accuracy: 0.8928 - va
Epoch 24/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8874 - loss: 0.7339 - val_accuracy: 0.8939 - va
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.8872 - loss: 0.7257 - val_accuracy: 0.8954 - va
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 6ms/step - accuracy: 0.8876 - loss: 0.7284 - val_accuracy: 0.8963 - va
Epoch 27/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8866 - loss: 0.7186 - val_accuracy: 0.8974 - va
Epoch 28/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8926 - loss: 0.7028 - val_accuracy: 0.8983 - va
Epoch 29/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8944 - loss: 0.7000 - val_accuracy: 0.8992 - va
```

```
# drop
import tensorflow as tf
import numpy as np
from tensorflow import keras
from tensorflow.keras.layers import Dropout

# Set hyperparameters
EPOCHS = 200
BATCH_SIZE = 128
```

```
VERBOSE = 1
NB_CLASSES = 10
N_HIDDEN = 128
VALIDATION_SPLIT = 0.2
DROPOUT_RATE = 0.3 # Dropout rate

# Load and preprocess data
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)
Y_test = tf.keras.utils.to_categorical(Y_test, NB_CLASSES)

# Build the neural network model with Dropout regularization
model_dropout = keras.Sequential([
    keras.layers.Input(shape=(RESHAPED,)),
    keras.layers.Dense(N_HIDDEN, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
    Dropout(DROPOUT_RATE), # Dropout layer after the first hidden layer
    keras.layers.Dense(N_HIDDEN // 2, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
    Dropout(DROPOUT_RATE), # Dropout layer after the second hidden layer
    keras.layers.Dense(NB_CLASSES, activation='softmax')
])

model_dropout.compile(
    optimizer=keras.optimizers.SGD(learning_rate=1e-3),
    loss="categorical_crossentropy",
    metrics=["accuracy"]
)

history_dropout = model_dropout.fit(
    X_train, Y_train,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_split=VALIDATION_SPLIT,
    verbose=VERBOSE,
)

test_loss_dropout, test_acc_dropout = model_dropout.evaluate(X_test, Y_test, verbose=1)
print(f"\nModel with Dropout Regularization:")
print(f"Test accuracy: {test_acc_dropout:.4f}  | Test loss: {test_loss_dropout:.4f}")

y_true_dropout = np.argmax(Y_test, axis=1)
y_pred_dropout = np.argmax(model_dropout.predict(X_test, verbose=0), axis=1)
cm_dropout = tf.math.confusion_matrix(y_true_dropout, y_pred_dropout, num_classes=NB_CLASSES).numpy()
print("\nConfusion Matrix (Dropout):")
print(cm_dropout)
```

```
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.1125 - loss: 2.3340 - val_accuracy: 0.2863 - va
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.2151 - loss: 2.1902 - val_accuracy: 0.4849 - va
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.3048 - loss: 2.0707 - val_accuracy: 0.5923 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.3907 - loss: 1.9336 - val_accuracy: 0.6561 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.4507 - loss: 1.7939 - val_accuracy: 0.6973 - va
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5004 - loss: 1.6602 - val_accuracy: 0.7303 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5346 - loss: 1.5403 - val_accuracy: 0.7511 - va
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.5673 - loss: 1.4280 - val_accuracy: 0.7713 - va
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.5988 - loss: 1.3280 - val_accuracy: 0.7883 - va
Epoch 10/200
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6179 - loss: 1.2513 - val_accuracy: 0.8011 - va
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.6316 - loss: 1.1912 - val_accuracy: 0.8123 - va
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.6606 - loss: 1.1160 - val_accuracy: 0.8238 - va
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.6701 - loss: 1.0748 - val_accuracy: 0.8313 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6861 - loss: 1.0222 - val_accuracy: 0.8364 - va
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6971 - loss: 0.9863 - val_accuracy: 0.8422 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7072 - loss: 0.9510 - val_accuracy: 0.8487 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.7165 - loss: 0.9172 - val_accuracy: 0.8533 - \
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7253 - loss: 0.8929 - val_accuracy: 0.8577 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7368 - loss: 0.8603 - val_accuracy: 0.8605 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.7367 - loss: 0.8450 - val_accuracy: 0.8639 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7483 - loss: 0.8158 - val_accuracy: 0.8660 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 13ms/step - accuracy: 0.7501 - loss: 0.8059 - val_accuracy: 0.8683 - \
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7580 - loss: 0.7850 - val_accuracy: 0.8708 - va
Epoch 24/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7626 - loss: 0.7670 - val_accuracy: 0.8732 - va
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7705 - loss: 0.7509 - val_accuracy: 0.8758 - va
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.7750 - loss: 0.7334 - val_accuracy: 0.8778 - va
Epoch 27/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7755 - loss: 0.7336 - val_accuracy: 0.8790 - va
Epoch 28/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7836 - loss: 0.7114 - val_accuracy: 0.8817 - va
Epoch 29/200
```

```
#Adam
import tensorflow as tf
import numpy as np
from tensorflow import keras

# Set hyperparameters
EPOCHS = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10
N_HIDDEN = 128
VALIDATION_SPLIT = 0.2

# Load and preprocess data
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)
Y_test = tf.keras.utils.to_categorical(Y_test, NB_CLASSES)

# Define a function to create and train a model with a given optimizer
def train_model_with_optimizer(optimizer, optimizer_name):
    print(f"Training model with {optimizer_name} optimizer...")
    model = keras.Sequential([
        keras.layers.Input(shape=(RESHAPED,)),
        keras.layers.Dense(N_HIDDEN, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(N_HIDDEN // 2, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(NB_CLASSES, activation='softmax')
```

```
    ])

    model.compile(
        optimizer=optimizer,
        loss="categorical_crossentropy",
        metrics=["accuracy"]
    )

    history = model.fit(
        X_train, Y_train,
        batch_size=BATCH_SIZE,
        epochs=EPOCHS,
        validation_split=VALIDATION_SPLIT,
        verbose=VERBOSE,
    )

    test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=1)
    print(f"\nModel with {optimizer_name} Optimizer:")
    print(f"Test accuracy: {test_acc:.4f}  | Test loss: {test_loss:.4f}")

    y_true = np.argmax(Y_test, axis=1)
    y_pred = np.argmax(model.predict(X_test, verbose=0), axis=1)
    cm = tf.math.confusion_matrix(y_true, y_pred, num_classes=NB_CLASSES).numpy()
    print(f"\nConfusion Matrix ({optimizer_name}):")
    print(cm)

    return history, test_acc, test_loss

# Train models with different optimizers
optimizers = {
    "Adam": keras.optimizers.Adam(),
}

results_optimizers = {}
for optimizer_name, optimizer in optimizers.items():
    history, test_acc, test_loss = train_model_with_optimizer(optimizer, optimizer_name)
    results_optimizers[optimizer_name] = {"history": history, "test_acc": test_acc, "test_loss": test_
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training model with Adam optimizer...
/usr/local/lib/python3.12/dist-packages/keras/src/layers/activations/leaky_relu.py:41: UserWarning: Ar
  warnings.warn(
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.8077 - loss: 0.6977 - val_accuracy: 0.9488 - v
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9530 - loss: 0.1594 - val_accuracy: 0.9585 - v
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9672 - loss: 0.1095 - val_accuracy: 0.9656 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9773 - loss: 0.0761 - val_accuracy: 0.9695 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 5s 14ms/step - accuracy: 0.9838 - loss: 0.0574 - val_accuracy: 0.9705 - v
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9861 - loss: 0.0490 - val_accuracy: 0.9744 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9891 - loss: 0.0366 - val_accuracy: 0.9703 - v
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 8s 20ms/step - accuracy: 0.9914 - loss: 0.0290 - val_accuracy: 0.9730 - v
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 7s 12ms/step - accuracy: 0.9921 - loss: 0.0268 - val_accuracy: 0.9737 - v
Epoch 10/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 8s 19ms/step - accuracy: 0.9940 - loss: 0.0205 - val_accuracy: 0.9730 - v
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9944 - loss: 0.0173 - val_accuracy: 0.9742 - v
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 11ms/step - accuracy: 0.9959 - loss: 0.0142 - val_accuracy: 0.9703 - v
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9973 - loss: 0.0109 - val_accuracy: 0.9743 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9961 - loss: 0.0124 - val_accuracy: 0.9733 - va
```

```
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9981 - loss: 0.0080 - val_accuracy: 0.9753 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9988 - loss: 0.0050 - val_accuracy: 0.9777 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9981 - loss: 0.0079 - val_accuracy: 0.9757 - va
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9964 - loss: 0.0099 - val_accuracy: 0.9731 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9970 - loss: 0.0095 - val_accuracy: 0.9763 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9978 - loss: 0.0067 - val_accuracy: 0.9764 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9993 - loss: 0.0030 - val_accuracy: 0.9761 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 10ms/step - accuracy: 0.9993 - loss: 0.0033 - val_accuracy: 0.9774 - v
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9995 - loss: 0.0024 - val_accuracy: 0.9778 - va
Epoch 24/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 1.0000 - loss: 6.8299e-04 - val_accuracy: 0.9787
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 1.0000 - loss: 3.4710e-04 - val_accuracy: 0.9785
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 1.0000 - loss: 2.4150e-04 - val_accuracy: 0.9788
```

```python
#Adagrad
import tensorflow as tf
import numpy as np
from tensorflow import keras

# Set hyperparameters
EPOCHS = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10
N_HIDDEN = 128
VALIDATION_SPLIT = 0.2

# Load and preprocess data
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)
Y_test = tf.keras.utils.to_categorical(Y_test, NB_CLASSES)

# Define a function to create and train a model with a given optimizer
def train_model_with_optimizer(optimizer, optimizer_name):
    print(f"Training model with {optimizer_name} optimizer...")
    model = keras.Sequential([
        keras.layers.Input(shape=(RESHAPED,)),
        keras.layers.Dense(N_HIDDEN, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(N_HIDDEN // 2, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(NB_CLASSES, activation='softmax')
    ])

    model.compile(
        optimizer=optimizer,
        loss="categorical_crossentropy",
        metrics=["accuracy"]
    )

    history = model.fit(
        X_train, Y_train,
        batch_size=BATCH_SIZE,
        epochs=EPOCHS,
        validation_split=VALIDATION_SPLIT,
```

```
        verbose=VERBOSE,
    )

    test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=1)
    print(f"\nModel with {optimizer_name} Optimizer:")
    print(f"Test accuracy: {test_acc:.4f}  | Test loss: {test_loss:.4f}")

    y_true = np.argmax(Y_test, axis=1)
    y_pred = np.argmax(model.predict(X_test, verbose=0), axis=1)
    cm = tf.math.confusion_matrix(y_true, y_pred, num_classes=NB_CLASSES).numpy()
    print(f"\nConfusion Matrix ({optimizer_name}):")
    print(cm)

    return history, test_acc, test_loss

# Train models with different optimizers
optimizers = {
    "Adagrad": keras.optimizers.Adagrad(),
}

results_optimizers = {}
for optimizer_name, optimizer in optimizers.items():
    history, test_acc, test_loss = train_model_with_optimizer(optimizer, optimizer_name)
    results_optimizers[optimizer_name] = {"history": history, "test_acc": test_acc, "test_loss": test_
```

```
Training model with Adagrad optimizer...
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.2034 - loss: 2.2362 - val_accuracy: 0.5017 - va
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.5660 - loss: 1.7387 - val_accuracy: 0.7434 - va
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.7567 - loss: 1.2503 - val_accuracy: 0.8313 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.8227 - loss: 0.9160 - val_accuracy: 0.8535 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8451 - loss: 0.7338 - val_accuracy: 0.8708 - va
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8593 - loss: 0.6250 - val_accuracy: 0.8769 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8686 - loss: 0.5624 - val_accuracy: 0.8820 - va
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8729 - loss: 0.5212 - val_accuracy: 0.8867 - va
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.8775 - loss: 0.4841 - val_accuracy: 0.8916 - va
Epoch 10/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.8815 - loss: 0.4593 - val_accuracy: 0.8949 - va
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8845 - loss: 0.4417 - val_accuracy: 0.8974 - va
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8871 - loss: 0.4266 - val_accuracy: 0.9002 - va
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8902 - loss: 0.4123 - val_accuracy: 0.9022 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8955 - loss: 0.3950 - val_accuracy: 0.9038 - va
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.8955 - loss: 0.3881 - val_accuracy: 0.9054 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8974 - loss: 0.3808 - val_accuracy: 0.9062 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8976 - loss: 0.3730 - val_accuracy: 0.9083 - va
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8998 - loss: 0.3659 - val_accuracy: 0.9094 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.8982 - loss: 0.3634 - val_accuracy: 0.9102 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9005 - loss: 0.3566 - val_accuracy: 0.9107 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9020 - loss: 0.3511 - val_accuracy: 0.9124 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9052 - loss: 0.3380 - val_accuracy: 0.9122 - va
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9051 - loss: 0.3404 - val_accuracy: 0.9131 - va
Epoch 24/200
```

```
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9075 - loss: 0.3332 - val_accuracy: 0.9133 - va
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9079 - loss: 0.3278 - val_accuracy: 0.9140 - va
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9081 - loss: 0.3271 - val_accuracy: 0.9147 - va
Epoch 27/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9093 - loss: 0.3219 - val_accuracy: 0.9157 - va
Epoch 28/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9105 - loss: 0.3252 - val_accuracy: 0.9168 - va
```

```
#rmsprop
import tensorflow as tf
import numpy as np
from tensorflow import keras

# Set hyperparameters
EPOCHS = 200
BATCH_SIZE = 128
VERBOSE = 1
NB_CLASSES = 10
N_HIDDEN = 128
VALIDATION_SPLIT = 0.2

# Load and preprocess data
mnist = keras.datasets.mnist
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
RESHAPED = 784
X_train = X_train.reshape(60000, RESHAPED)
X_test = X_test.reshape(10000, RESHAPED)
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
Y_train = tf.keras.utils.to_categorical(Y_train, NB_CLASSES)
Y_test = tf.keras.utils.to_categorical(Y_test, NB_CLASSES)

# Define a function to create and train a model with a given optimizer
def train_model_with_optimizer(optimizer, optimizer_name):
    print(f"Training model with {optimizer_name} optimizer...")
    model = keras.Sequential([
        keras.layers.Input(shape=(RESHAPED,)),
        keras.layers.Dense(N_HIDDEN, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(N_HIDDEN // 2, activation=tf.keras.layers.LeakyReLU(alpha=0.01)),
        keras.layers.Dense(NB_CLASSES, activation='softmax')
    ])

    model.compile(
        optimizer=optimizer,
        loss="categorical_crossentropy",
        metrics=["accuracy"]
    )

    history = model.fit(
        X_train, Y_train,
        batch_size=BATCH_SIZE,
        epochs=EPOCHS,
        validation_split=VALIDATION_SPLIT,
        verbose=VERBOSE,
    )

    test_loss, test_acc = model.evaluate(X_test, Y_test, verbose=1)
    print(f"\nModel with {optimizer_name} Optimizer:")
    print(f"Test accuracy: {test_acc:.4f}  | Test loss: {test_loss:.4f}")

    y_true = np.argmax(Y_test, axis=1)
    y_pred = np.argmax(model.predict(X_test, verbose=0), axis=1)
    cm = tf.math.confusion_matrix(y_true, y_pred, num_classes=NB_CLASSES).numpy()
    print(f"\nConfusion Matrix ({optimizer_name}):")
    print(cm)
```

```
    return history, test_acc, test_loss

# Train models with different optimizers
optimizers = {
    "RMSprop": keras.optimizers.RMSprop(),
}

results_optimizers = {}
for optimizer_name, optimizer in optimizers.items():
    history, test_acc, test_loss = train_model_with_optimizer(optimizer, optimizer_name)
    results_optimizers[optimizer_name] = {"history": history, "test_acc": test_acc, "test_loss": test_
```

```
Training model with RMSprop optimizer...
Epoch 1/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.8298 - loss: 0.6044 - val_accuracy: 0.9469 - va
Epoch 2/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9504 - loss: 0.1698 - val_accuracy: 0.9611 - va
Epoch 3/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9665 - loss: 0.1081 - val_accuracy: 0.9668 - va
Epoch 4/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9769 - loss: 0.0775 - val_accuracy: 0.9683 - va
Epoch 5/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9799 - loss: 0.0620 - val_accuracy: 0.9732 - va
Epoch 6/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9860 - loss: 0.0483 - val_accuracy: 0.9734 - va
Epoch 7/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9887 - loss: 0.0395 - val_accuracy: 0.9737 - va
Epoch 8/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9901 - loss: 0.0324 - val_accuracy: 0.9763 - va
Epoch 9/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9923 - loss: 0.0250 - val_accuracy: 0.9747 - va
Epoch 10/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9938 - loss: 0.0216 - val_accuracy: 0.9746 - va
Epoch 11/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9951 - loss: 0.0175 - val_accuracy: 0.9768 - va
Epoch 12/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 7ms/step - accuracy: 0.9960 - loss: 0.0136 - val_accuracy: 0.9764 - va
Epoch 13/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9971 - loss: 0.0107 - val_accuracy: 0.9752 - va
Epoch 14/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9976 - loss: 0.0097 - val_accuracy: 0.9754 - va
Epoch 15/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9977 - loss: 0.0081 - val_accuracy: 0.9751 - va
Epoch 16/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 4s 6ms/step - accuracy: 0.9979 - loss: 0.0062 - val_accuracy: 0.9771 - va
Epoch 17/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9987 - loss: 0.0047 - val_accuracy: 0.9762 - va
Epoch 18/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9991 - loss: 0.0039 - val_accuracy: 0.9753 - va
Epoch 19/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.9988 - loss: 0.0040 - val_accuracy: 0.9729 - va
Epoch 20/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9991 - loss: 0.0029 - val_accuracy: 0.9781 - va
Epoch 21/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9993 - loss: 0.0028 - val_accuracy: 0.9772 - va
Epoch 22/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9995 - loss: 0.0024 - val_accuracy: 0.9783 - va
Epoch 23/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9997 - loss: 0.0013 - val_accuracy: 0.9750 - va
Epoch 24/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 3s 9ms/step - accuracy: 0.9995 - loss: 0.0016 - val_accuracy: 0.9775 - va
Epoch 25/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9998 - loss: 0.0011 - val_accuracy: 0.9788 - va
Epoch 26/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 1.0000 - loss: 3.6755e-04 - val_accuracy: 0.9769
Epoch 27/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9999 - loss: 8.6790e-04 - val_accuracy: 0.9780
Epoch 28/200
375/375 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9999 - loss: 5.0368e-04 - val_accuracy: 0.9788
Epoch 29/200
```

Start coding or generate with AI.

Comparative Analysis of Optimizers for MNIST Classification

This analysis compares the performance of three different optimizers (SGD, Adam, and Adagrad) when training a simple Multi-layer Perceptron (MLP) on the MNIST dataset. The MLP architecture consists of two hidden layers with LeakyReLU activation and an output layer with softmax activation.

Optimizer	Test Accuracy	Test Loss
SGD	0.9699	0.1086
Adam	0.9808	0.2112
Adagrad	0.9391	0.2250
RMSprop	0.9757	0.1400

Observations and Discussion:

- **Adam Optimizer:** Achieved the highest test accuracy (0.9808) among the tested optimizers. It also converged relatively quickly, as seen in the training history. This is consistent with Adam's reputation for performing well on a wide range of deep learning tasks due to its adaptive learning rate capabilities.
- **SGD Optimizer:** While not reaching the same peak accuracy as Adam, the SGD optimizer still performed reasonably well (0.9699 accuracy). Its training process showed a more gradual improvement in loss compared to Adam.
- **Adagrad Optimizer:** Had the lowest test accuracy (0.9391) and a higher test loss compared to Adam and SGD. Adagrad's adaptive learning rate can sometimes cause the learning rate to become too small too quickly, potentially hindering convergence on this dataset.
- **RMSprop Optimizer:** Performed well (0.9757 accuracy), coming in second place after Adam. RMSprop is also an adaptive learning rate optimizer and is often a good alternative to Adam.

Conclusion:

Based on this experiment, the **Adam optimizer** demonstrated the best performance in terms of achieving the highest test accuracy on the MNIST dataset with this specific MLP architecture. SGD also performed adequately, while Adagrad struggled to reach the same level of accuracy within the given epochs. RMSprop is another strong contender.

It's important to note that the optimal optimizer can depend on the dataset, network architecture, and