1. Create a Sequential model for classifying MNIST digits with the following architecture: Flatten input layer for shape (28,28), Dense layer with 128 neurons and ReLU activation, Dropout layer with rate 0.3, Dense output layer with 10 neurons and softmax activation. Compile the model using Adam optimizer and categorical crossentropy loss.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Flatten, Dense, Dropout

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load and preprocess the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

x\_train = x\_train / 255.0

x\_test = x\_test / 255.0

# One-hot encode labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Create Sequential model

model = Sequential([

Flatten(input\_shape=(28, 28)),

Dense(128, activation='relu'),

Dropout(0.3),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'\nTest Accuracy: {test\_acc:.4f}')

1. Create a CNN model for grayscale images of shape (28,28,1) with: Conv2D layer with 32 filters, 3x3 kernel, ReLU; MaxPooling2D with pool size 2x2; Conv2D layer with 64 filters, 3x3 kernel, ReLU; Flatten and Dense layer with 128 neurons, ReLU; Output Dense layer with 10 neurons, softmax.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

from tensorflow.keras.datasets import mnist

from tensorflow.keras.utils import to\_categorical

# Load and preprocess the MNIST dataset

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

# Reshape to include channel dimension (grayscale = 1)

x\_train = x\_train.reshape(-1, 28, 28, 1).astype('float32') / 255.0

x\_test = x\_test.reshape(-1, 28, 28, 1).astype('float32') / 255.0

# One-hot encode labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Create CNN model

model = Sequential([

Conv2D(32, (3, 3), activation='relu', input\_shape=(28, 28, 1)),

MaxPooling2D(pool\_size=(2, 2)),

Conv2D(64, (3, 3), activation='relu'),

Flatten(),

Dense(128, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)

# Evaluate the model

test\_loss, test\_acc = model.evaluate(x\_test, y\_test, verbose=2)

print(f'\nTest Accuracy: {test\_acc:.4f}')

1. Given a pre-trained Sequential model, add a new Dense layer of 64 neurons before the output and compile the model.

# Import required libraries

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Example: Assume you already have a pre-trained Sequential model

pretrained\_model = Sequential([

Dense(128, activation='relu', input\_shape=(100,)),

Dense(10, activation='softmax')

])

# Get the output of all layers except the last one

layers = pretrained\_model.layers[:-1]

# Create a new model and add the previous layers

new\_model = Sequential()

for layer in layers:

new\_model.add(layer)

# Add a new Dense layer with 64 neurons (ReLU)

new\_model.add(Dense(64, activation='relu'))

# Add the original output layer again

new\_model.add(Dense(10, activation='softmax'))

# Compile the modified model

new\_model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Model summary (optional)

new\_model.summary()

1. Explain in code comments the difference between using sigmoid vs softmax for the output layer. Then implement an example with 3-class classification.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

# -----------------------------------------

# 🔍 Difference between Sigmoid and Softmax:

# -----------------------------------------

# - Sigmoid: Used for \*\*binary classification\*\* (2 classes).

# It outputs a single probability between 0 and 1 for one class.

# Example: Output layer -> Dense(1, activation='sigmoid')

#

# - Softmax: Used for \*\*multi-class classification\*\* (more than 2 classes).

# It outputs probabilities that sum up to 1 across all classes.

# Example: Output layer -> Dense(num\_classes, activation='softmax')

# -----------------------------------------

# Example: 3-class classification problem

# Create dummy input data (100 samples, 5 features each)

X = np.random.random((100, 5))

# Dummy target labels (values: 0, 1, or 2)

y = np.random.randint(3, size=(100,))

# Convert labels to one-hot encoded format for 3 classes

y\_onehot = tf.keras.utils.to\_categorical(y, num\_classes=3)

# Create the model

model = Sequential([

Dense(16, activation='relu', input\_shape=(5,)),

Dense(8, activation='relu'),

Dense(3, activation='softmax') # Softmax for multi-class output

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(X, y\_onehot, epochs=10, batch\_size=8, verbose=1)

# Evaluate the model

loss, acc = model.evaluate(X, y\_onehot, verbose=0)

print(f"\nModel Accuracy: {acc:.4f}")

1. Create a functional API model that takes two inputs: Input1 of shape (32,) and Input2 of shape (32,). Concatenate them, pass through a Dense layer of 64 neurons, then output a single neuron with sigmoid activation.

# Import required libraries

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense, Concatenate

# Define two input layers

input1 = Input(shape=(32,))

input2 = Input(shape=(32,))

# Concatenate the inputs

merged = Concatenate()([input1, input2])

# Add a Dense layer with 64 neurons and ReLU activation

dense = Dense(64, activation='relu')(merged)

# Output layer with 1 neuron and sigmoid activation

output = Dense(1, activation='sigmoid')(dense)

# Create the Functional API model

model = Model(inputs=[input1, input2], outputs=output)

# Compile the model

model.compile(optimizer='adam',

loss='binary\_crossentropy',

metrics=['accuracy'])

# Display model summary (optional)

model.summary()

1. Load the CIFAR-10 dataset, normalize images to range 0–1, and one-hot encode the labels.

# Import required libraries

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

# Load the CIFAR-10 dataset

(x\_train, y\_train), (x\_test, y\_test) = cifar10.load\_data()

# Normalize images to range [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# One-hot encode the labels (10 classes)

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Check shapes (optional)

print("x\_train shape:", x\_train.shape)

print("y\_train shape:", y\_train.shape)

print("x\_test shape:", x\_test.shape)

print("y\_test shape:", y\_test.shape)

1. Given a dataset X of shape (1000, 28, 28), reshape it appropriately for a CNN input.

import numpy as np

# Example dataset

X = np.random.random((1000, 28, 28)) # shape: (samples, height, width)

# Reshape for CNN input -> (samples, height, width, channels)

# Since these are grayscale images, channels = 1

X\_reshaped = X.reshape(-1, 28, 28, 1)

# Verify the new shape

print("New shape:", X\_reshaped.shape)

1. Write code to split a dataset into 70% training, 15% validation, and 15% testing using sklearn.

# Import required library

from sklearn.model\_selection import train\_test\_split

import numpy as np

# Example dataset

X = np.random.rand(1000, 10) # 1000 samples, 10 features

y = np.random.randint(0, 2, 1000) # Binary labels

# Step 1: Split into 70% train and 30% temp (validation + test)

X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(

X, y, test\_size=0.30, random\_state=42, stratify=y

)

# Step 2: Split the 30% temp into 15% validation and 15% test

X\_val, X\_test, y\_val, y\_test = train\_test\_split(

X\_temp, y\_temp, test\_size=0.5, random\_state=42, stratify=y\_temp

)

# Verify shapes

print("Training set:", X\_train.shape)

print("Validation set:", X\_val.shape)

print("Testing set:", X\_test.shape)

1. Apply data augmentation on image data using ImageDataGenerator with horizontal flips, rotations of 15 degrees, and width/height shifts of 0.1.

# Import required library

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import numpy as np

# Example image dataset (100 samples, 28x28 grayscale images)

X = np.random.rand(100, 28, 28, 1)

y = np.random.randint(0, 10, 100)

# Create ImageDataGenerator with specified augmentations

datagen = ImageDataGenerator(

rotation\_range=15, # Rotate images by ±15 degrees

width\_shift\_range=0.1, # Shift horizontally by 10% of width

height\_shift\_range=0.1, # Shift vertically by 10% of height

horizontal\_flip=True # Randomly flip images horizontally

)

# Fit the generator (required if using featurewise\_center or std normalization)

datagen.fit(X)

# Example: Generate augmented images in batches

batch\_size = 16

augmented\_iterator = datagen.flow(X, y, batch\_size=batch\_size)

# Get one batch of augmented images

X\_augmented, y\_augmented = next(augmented\_iterator)

print("Original batch shape:", X.shape)

print("Augmented batch shape:", X\_augmented.shape)

1. Visualize 5 random images from your dataset with their corresponding labels using matplotlib.

# Import required libraries

import matplotlib.pyplot as plt

import numpy as np

# Example dataset (100 samples, 28x28 grayscale images)

X = np.random.rand(100, 28, 28) # shape: (samples, height, width)

y = np.random.randint(0, 10, 100) # labels

# Select 5 random indices

random\_indices = np.random.choice(X.shape[0], 5, replace=False)

# Plot the images

plt.figure(figsize=(10, 2))

for i, idx in enumerate(random\_indices):

plt.subplot(1, 5, i+1)

plt.imshow(X[idx], cmap='gray')

plt.title(f"Label: {y[idx]}")

plt.axis('off')

plt.show()

1. Train a Sequential model on training data for 10 epochs, using validation data for evaluation. Include EarlyStopping if validation loss doesn’t improve for 3 epochs.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.callbacks import EarlyStopping

import numpy as np

# Example dataset (1000 samples, 28x28 images, grayscale)

X = np.random.rand(1000, 28, 28, 1)

y = np.random.randint(0, 10, 1000)

# One-hot encode labels for multi-class classification

y = tf.keras.utils.to\_categorical(y, 10)

# Split into training and validation sets (80% train, 20% val)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a simple Sequential model

model = Sequential([

Flatten(input\_shape=(28,28,1)),

Dense(128, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# EarlyStopping callback

early\_stop = EarlyStopping(monitor='val\_loss', patience=3, restore\_best\_weights=True)

# Train the model

history = model.fit(

X\_train, y\_train,

epochs=10,

batch\_size=32,

validation\_data=(X\_val, y\_val),

callbacks=[early\_stop]

)

# Evaluate on validation data

val\_loss, val\_acc = model.evaluate(X\_val, y\_val, verbose=0)

print(f"\nValidation Accuracy: {val\_acc:.4f}")

1. Implement ModelCheckpoint to save the best model during training based on validation accuracy.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.callbacks import ModelCheckpoint

import numpy as np

# Example dataset (1000 samples, 28x28 grayscale images)

X = np.random.rand(1000, 28, 28, 1)

y = np.random.randint(0, 10, 1000)

# One-hot encode labels

y = tf.keras.utils.to\_categorical(y, 10)

# Split into training and validation sets

from sklearn.model\_selection import train\_test\_split

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a simple Sequential model

model = Sequential([

Flatten(input\_shape=(28,28,1)),

Dense(128, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# ModelCheckpoint callback: save best model based on validation accuracy

checkpoint = ModelCheckpoint(

'best\_model.h5', # Filepath to save the model

monitor='val\_accuracy', # Metric to monitor

save\_best\_only=True, # Save only the best model

mode='max', # 'max' because higher accuracy is better

verbose=1

)

# Train the model

history = model.fit(

X\_train, y\_train,

epochs=10,

batch\_size=32,

validation\_data=(X\_val, y\_val),

callbacks=[checkpoint]

)

# Load the best model later (optional)

best\_model = tf.keras.models.load\_model('best\_model.h5')

1. Plot the training and validation loss curves for a trained model and explain how to identify overfitting from the graph.

# Import required libraries

import matplotlib.pyplot as plt

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

import numpy as np

from sklearn.model\_selection import train\_test\_split

# Example dataset (1000 samples, 28x28 grayscale images)

X = np.random.rand(1000, 28, 28, 1)

y = np.random.randint(0, 10, 1000)

# One-hot encode labels

y = tf.keras.utils.to\_categorical(y, 10)

# Split into training and validation sets

X\_train, X\_val, y\_train, y\_val = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a simple Sequential model

model = Sequential([

Flatten(input\_shape=(28,28,1)),

Dense(128, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

history = model.fit(

X\_train, y\_train,

epochs=20,

batch\_size=32,

validation\_data=(X\_val, y\_val)

)

# Plot training and validation loss curves

plt.figure(figsize=(8,5))

plt.plot(history.history['loss'], label='Training Loss', marker='o')

plt.plot(history.history['val\_loss'], label='Validation Loss', marker='o')

plt.title('Training vs Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.grid(True)

plt.show()

1. Evaluate a trained model on test data and print both loss and accuracy.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

import numpy as np

from sklearn.model\_selection import train\_test\_split

# Example dataset (1000 samples, 28x28 grayscale images)

X = np.random.rand(1000, 28, 28, 1)

y = np.random.randint(0, 10, 1000)

# One-hot encode labels

y = tf.keras.utils.to\_categorical(y, 10)

# Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create a simple Sequential model

model = Sequential([

Flatten(input\_shape=(28,28,1)),

Dense(128, activation='relu'),

Dense(10, activation='softmax')

])

# Compile the model

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Train the model

model.fit(X\_train, y\_train, epochs=5, batch\_size=32, verbose=1)

# Evaluate the model on test data

test\_loss, test\_acc = model.evaluate(X\_test, y\_test, verbose=0)

print(f"Test Loss: {test\_loss:.4f}")

print(f"Test Accuracy: {test\_acc:.4f}")

1. Given training and validation loss arrays: train\_loss = [0.8, 0.5, 0.3, 0.2] and val\_loss = [0.9, 0.6, 0.4, 0.5], write code to detect overfitting and explain why it occurs.

# Training and validation loss arrays

train\_loss = [0.8, 0.5, 0.3, 0.2]

val\_loss = [0.9, 0.6, 0.4, 0.5]

# Detect overfitting

overfitting\_detected = False

for i in range(1, len(train\_loss)):

# Check if training loss decreases but validation loss increases

if train\_loss[i] < train\_loss[i-1] and val\_loss[i] > val\_loss[i-1]:

overfitting\_detected = True

epoch\_overfit = i+1

break

if overfitting\_detected:

print(f"Overfitting detected at epoch {epoch\_overfit}")

else:

print("No overfitting detected")

# Optional: visualize the loss curves

import matplotlib.pyplot as plt

plt.plot(range(1, len(train\_loss)+1), train\_loss, label='Training Loss', marker='o')

plt.plot(range(1, len(val\_loss)+1), val\_loss, label='Validation Loss', marker='o')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.title('Training vs Validation Loss')

plt.legend()

plt.grid(True)

plt.show()

1. Given a pre-trained CNN, write code to freeze all layers except the last Dense layer, then compile it for fine-tuning.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense

# Example pre-trained CNN model

pretrained\_model = Sequential([

Conv2D(32, (3,3), activation='relu', input\_shape=(28,28,1)),

MaxPooling2D((2,2)),

Conv2D(64, (3,3), activation='relu'),

Flatten(),

Dense(128, activation='relu'),

Dense(10, activation='softmax') # Output layer

])

# Freeze all layers except the last Dense layer

for layer in pretrained\_model.layers[:-1]:

layer.trainable = False

# Compile the model for fine-tuning

pretrained\_model.compile(

optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy']

)

# Model summary to check which layers are trainable

pretrained\_model.summary()

1. Extract the output of an intermediate layer (second Dense layer) for a single input sample using Keras functional API.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

import numpy as np

# Example Functional API model

inputs = Input(shape=(10,))

x = Dense(16, activation='relu')(inputs)

x = Dense(8, activation='relu', name='second\_dense')(x) # Second Dense layer

outputs = Dense(3, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs)

# Compile model (optional for extraction)

model.compile(optimizer='adam', loss='categorical\_crossentropy', metrics=['accuracy'])

# Example single input sample

sample\_input = np.random.rand(1, 10) # shape (1, 10)

# Create a new model to extract output of the second Dense layer

intermediate\_layer\_model = Model(inputs=model.input,

outputs=model.get\_layer('second\_dense').output)

# Get the output of the intermediate layer

intermediate\_output = intermediate\_layer\_model.predict(sample\_input)

print("Output of second Dense layer:", intermediate\_output)

1. Implement a custom loss function for mean squared error and use it in compiling a model.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Flatten

import numpy as np

# Define custom MSE loss function

def custom\_mse(y\_true, y\_pred):

return tf.reduce\_mean(tf.square(y\_true - y\_pred))

# Example dataset (regression problem)

X = np.random.rand(100, 5) # 100 samples, 5 features

y = np.random.rand(100, 1) # Regression targets

# Create a simple Sequential model

model = Sequential([

Dense(32, activation='relu', input\_shape=(5,)),

Dense(16, activation='relu'),

Dense(1) # Output for regression

])

# Compile the model with the custom loss function

model.compile(optimizer='adam',

loss=custom\_mse,

metrics=['mae'])

# Train the model

model.fit(X, y, epochs=10, batch\_size=8, verbose=1)

# Evaluate the model

loss, mae = model.evaluate(X, y, verbose=0)

print(f"Custom MSE Loss: {loss:.4f}, MAE: {mae:.4f}")

1. Save a trained model to disk in both HDF5 and SavedModel formats. Then write code to load it back.

# Import required libraries

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

# Example dataset

X = np.random.rand(100, 5)

y = np.random.rand(100, 1)

# Create a simple model

model = Sequential([

Dense(32, activation='relu', input\_shape=(5,)),

Dense(16, activation='relu'),

Dense(1)

])

# Compile the model

model.compile(optimizer='adam', loss='mse', metrics=['mae'])

# Train the model

model.fit(X, y, epochs=5, batch\_size=8, verbose=1)

# -------------------------------

# Save the model

# -------------------------------

# 1. HDF5 format

model.save('model\_hdf5.h5')

# 2. SavedModel format

model.save('model\_savedmodel') # Folder format

# -------------------------------

# Load the model back

# -------------------------------

# Load from HDF5

loaded\_hdf5\_model = tf.keras.models.load\_model('model\_hdf5.h5')

# Load from SavedModel

loaded\_savedmodel\_model = tf.keras.models.load\_model('model\_savedmodel')

# Test loaded model

loss, mae = loaded\_hdf5\_model.evaluate(X, y, verbose=0)

print(f"HDF5 Model - Loss: {loss:.4f}, MAE: {mae:.4f}")

loss, mae = loaded\_savedmodel\_model.evaluate(X, y, verbose=0)

print(f"SavedModel - Loss: {loss:.4f}, MAE: {mae:.4f}")

1. Apply softmax manually on the tensor logits = [2.0, 1.0, 0.1] using numpy and compare it with TensorFlow’s softmax output.

# Import required libraries

import numpy as np

import tensorflow as tf

# Example logits

logits = np.array([2.0, 1.0, 0.1])

# -----------------------------

# 1. Manual softmax using NumPy

# -----------------------------

exp\_logits = np.exp(logits - np.max(logits)) # Subtract max for numerical stability

softmax\_manual = exp\_logits / np.sum(exp\_logits)

print("Softmax (manual):", softmax\_manual)

# -----------------------------

# 2. Softmax using TensorFlow

# -----------------------------

logits\_tf = tf.constant(logits, dtype=tf.float32)

softmax\_tf = tf.nn.softmax(logits\_tf)

print("Softmax (TensorFlow):", softmax\_tf.numpy())

# -----------------------------

# 3. Compare results

# -----------------------------

print("Difference:", np.abs(softmax\_manual - softmax\_tf.numpy()))