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.

Ans:-

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.optimizers import Adam

# Create the Sequential model

model = Sequential([

    Flatten(input\_shape=(28, 28)),              # Input layer

    Dense(128, activation='relu'),              # Hidden layer

    Dropout(0.3),                               # Dropout layer

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

])

# Compile the model

model.compile(optimizer=Adam(),

              loss='categorical\_crossentropy',

              metrics=['accuracy'])

# Display the model architecture

model.summary()

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.

Ans:-

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.optimizers import Adam

# Create the CNN model

model = Sequential([

Conv2D(32, (3,3), activation='relu', input\_shape=(28, 28, 1)), # 1st Conv layer

MaxPooling2D(pool\_size=(2,2)), # 1st Pooling layer

Conv2D(64, (3,3), activation='relu'), # 2nd Conv layer

Flatten(), # Flatten for dense layers

Dense(128, activation='relu'), # Hidden layer

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

])

# Compile the model

model.compile(optimizer=Adam(),

loss='categorical\_crossentropy',

metrics=['accuracy'])

# Show model summary

model.summary()

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

Ans:-

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Assume 'model' is your pre-trained Sequential model

# Let's modify it by adding a Dense layer before the output

# Get all layers

layers = model.layers

# Create a new Sequential model

new\_model = Sequential()

# Add all previous layers except the last (output) layer

for layer in layers[:-1]:

    new\_model.add(layer)

# Add the new Dense layer (128 neurons, ReLU) before the original output layer

new\_model.add(Dense(128, activation='relu'))

# Add the old output layer again

new\_model.add(layers[-1])

# Compile the new model

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

                  loss='categorical\_crossentropy',

                  metrics=['accuracy'])

# Check summary

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.

Ans:-

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

# ✅ Difference explained in comments:

# Sigmoid:

# - Used for binary classification (2 classes: e.g., cat vs. dog).

# - Outputs a probability between 0 and 1 for EACH neuron independently.

# - If you use multiple sigmoid neurons, each one acts as a separate binary classifier.

# Example: multi-label classification (an image can be both "cat" and "pet").

# Softmax:

# - Used for multi-class classification (e.g., 3 or more classes like cat, dog, horse).

# - Outputs a probability distribution across all classes that sums to 1.

# - The class with the highest probability is the model’s prediction.

# ✅ Example: 3-class classification model using Softmax

model = Sequential([

Dense(16, activation='relu', input\_shape=(10,)), # Input layer (10 features)

Dense(8, activation='relu'), # Hidden layer

Dense(3, activation='softmax') # Output: 3 classes → use Softmax

])

# Compile the model

model.compile(optimizer=Adam(),

loss='categorical\_crossentropy', # Used with softmax outputs

metrics=['accuracy'])

# Show the model summary

model.summary()

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.

Ans:-

from tensorflow.keras.models import Model

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

from tensorflow.keras.optimizers import Adam

# Define two input layers

input1 = Input(shape=(32,), name='Input1')

input2 = Input(shape=(32,), name='Input2')

# Concatenate the two inputs

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

# Hidden dense layer with 64 neurons and ReLU activation

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

# Output layer with 1 neuron and sigmoid activation

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

# Create the functional model

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

# Compile the model

model.compile(optimizer=Adam(),

loss='binary\_crossentropy',

metrics=['accuracy'])

# Display the model summary

model.summary()

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

Ans:-

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

# 1️⃣ Load CIFAR-10 dataset

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

# 2️⃣ Normalize image pixel values to range [0, 1]

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

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

# 3️⃣ One-hot encode the labels (10 classes)

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

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

# 4️⃣ Display shapes to verify

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

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

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

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

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

Ans:-

import numpy as np

# Suppose X has shape (1000, 28, 28)

# 1000 images, each 28x28 pixels (grayscale)

# Define a sample X for demonstration

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

# ✅ Reshape to include the channel dimension

X\_reshaped = X.reshape((1000, 28, 28, 1))

# Optionally, convert to float and normalize to [0,1]

X\_reshaped = X\_reshaped.astype('float32') / 255.0

# Check the new shape

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

print("Reshaped for CNN:", X\_reshaped.shape)

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

Ans:-

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

# Example dataset (X = features, y = labels)

# Assume X and y are already defined, e.g. X.shape = (1000, 28, 28), y.shape = (1000,)

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

X\_train, X\_temp, y\_train, y\_temp = train\_test\_split(X, y, test\_size=0.30, random\_state=42)

# Step 2: Split temp into validation (15%) and test (15%)

# Since X\_temp is 30%, we split it equally into 15% each => test\_size=0.5

X\_val, X\_test, y\_val, y\_test = train\_test\_split(X\_temp, y\_temp, test\_size=0.5, random\_state=42)

# Check dataset sizes

print(f"Training set: {len(X\_train)} samples")

print(f"Validation set: {len(X\_val)} samples")

print(f"Testing set: {len(X\_test)} samples")

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

Ans:-

from tensorflow.keras.preprocessing.image import ImageDataGenerator

import numpy as np # Import numpy

# Create an ImageDataGenerator with the specified augmentations

datagen = ImageDataGenerator(

    rotation\_range=15,          # Rotate images up to 15 degrees

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

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

    horizontal\_flip=True,       # Randomly flip images horizontally

    fill\_mode='nearest'         # Fill empty pixels after rotation/shift

)

# Example: assuming X\_train contains your training images

# Reshape X\_train to include a channel dimension (assuming grayscale)

X\_train\_reshaped = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], X\_train.shape[2], 1)

# Fit the generator on your data (optional but good for some datasets)

datagen.fit(X\_train\_reshaped)

# Example of using the generator in training

# model.fit(datagen.flow(X\_train\_reshaped, y\_train, batch\_size=32), epochs=20)

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

Ans:-

import matplotlib.pyplot as plt

import numpy as np

# Assume X contains images and y contains labels

# For example, X.shape = (1000, 28, 28, 1) or (1000, 28, 28, 3)

# y can be numeric labels or one-hot encoded (if one-hot, convert to class index)

if len(y.shape) > 1: # one-hot encoded

y\_labels = np.argmax(y, axis=1)

else:

y\_labels = y

# Randomly select 5 indices

random\_indices = np.random.choice(len(X), size=5, replace=False)

# Plot the images with labels

plt.figure(figsize=(12, 3))

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

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

# Squeeze in case of single channel (grayscale)

if X.shape[-1] == 1:

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

else:

plt.imshow(X[idx])

plt.title(f"Label: {y\_labels[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.

Ans:-

from tensorflow.keras.models import Sequential

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

from tensorflow.keras.callbacks import EarlyStopping

import numpy as np # Import numpy

# Define sample data for demonstration

X\_train = np.random.rand(700, 28, 28, 1) # Sample training images (700 samples, 28x28 grayscale)

y\_train = np.random.randint(0, 10, 700) # Sample training labels (700 samples, numeric)

X\_val = np.random.rand(150, 28, 28, 1)   # Sample validation images (150 samples, 28x28 grayscale)

y\_val = np.random.randint(0, 10, 150)   # Sample validation labels (150 samples, numeric)

# One-hot encode the labels (if not already done)

from tensorflow.keras.utils import to\_categorical

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

y\_val = to\_categorical(y\_val, num\_classes=10)

# Assuming y\_train is one-hot encoded, determine the number of classes

num\_classes = y\_train.shape[1]

# Reshape X\_train to include a channel dimension if it's missing (assuming grayscale)

if len(X\_train.shape) == 3:

    X\_train = X\_train.reshape(X\_train.shape[0], X\_train.shape[1], X\_train.shape[2], 1)

if len(X\_val.shape) == 3:

    X\_val = X\_val.reshape(X\_val.shape[0], X\_val.shape[1], X\_val.shape[2], 1)

# Example Sequential model

model = Sequential([

    Conv2D(32, (3,3), activation='relu', input\_shape=X\_train.shape[1:]),

    MaxPooling2D((2,2)),

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

    MaxPooling2D((2,2)),

    Flatten(),

    Dense(128, activation='relu'),

    Dense(num\_classes, activation='softmax')  # num\_classes = number of classes in y

])

# Compile the model

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

# EarlyStopping callback: stop if validation loss doesn't improve for 3 epochs

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]

)

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

Ans:-

from tensorflow.keras.callbacks import ModelCheckpoint

# Create a ModelCheckpoint callback

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 val\_accuracy is better

verbose=1

)

# Train the model with checkpoint (assuming 'model' is already defined)

history = model.fit(

X\_train, y\_train,

epochs=10,

batch\_size=32,

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

callbacks=[checkpoint] # Include ModelCheckpoint callback

)

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

Ans:-

import matplotlib.pyplot as plt

# Assume 'history' is the History object returned by model.fit()

loss = history.history['loss']

val\_loss = history.history['val\_loss']

epochs = range(1, len(loss) + 1)

# Plot training and validation loss

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

plt.plot(epochs, loss, 'b-', label='Training Loss')

plt.plot(epochs, val\_loss, 'r-', label='Validation Loss')

plt.title('Training and Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

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

Ans:-

from tensorflow.keras.utils import to\_categorical

import numpy as np # Import numpy

# Define sample data for demonstration

X\_test = np.random.rand(150, 28, 28, 1) # Sample test images (150 samples, 28x28 grayscale)

y\_test = np.random.randint(0, 10, 150)   # Sample test labels (150 samples, numeric)

# One-hot encode y\_test if it's not already

if len(y\_test.shape) == 1:

    y\_test = to\_categorical(y\_test, num\_classes=10) # Assuming 10 classes

# Evaluate the model on test data

test\_loss, test\_accuracy = model.evaluate(X\_test, y\_test, batch\_size=32)

# Print the results

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

print(f"Test Accuracy: {test\_accuracy:.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.

Ans:-

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

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

# Detect overfitting

overfitting = False

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

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

        overfitting = True

        print(f"Overfitting detected at epoch {i+1}")

        break

if not overfitting:

    print("No overfitting detected")

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

Ans:-

from tensorflow.keras.applications import VGG16

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense, Flatten

from tensorflow.keras.optimizers import Adam

# Load a pre-trained CNN (e.g., VGG16) without top layer

base\_model = VGG16(weights='imagenet', include\_top=False, input\_shape=(224, 224, 3))

# Freeze all layers in the base model

for layer in base\_model.layers:

layer.trainable = False

# Add a new Dense output layer

x = base\_model.output

x = Flatten()(x)

output = Dense(10, activation='softmax')(x) # Example: 10 classes

# Create the final model

model = Model(inputs=base\_model.input, outputs=output)

# Only the last Dense layer is trainable

for layer in model.layers:

print(layer.name, layer.trainable) # Optional: check which layers are trainable

# Compile the model for fine-tuning

model.compile(optimizer=Adam(learning\_rate=0.001),

loss='categorical\_crossentropy',

metrics=['accuracy'])

print("Model ready for fine-tuning!")

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

Ans:-

import numpy as np

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

# Example model

inputs = Input(shape=(20,))

x = Dense(32, activation='relu', name='dense1')(inputs)

x = Dense(16, activation='relu', name='dense2')(x) # Second Dense layer

output = Dense(10, activation='softmax', name='output')(x)

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

# Single input sample

sample\_input = np.random.rand(1, 20) # Shape: (1, 20)

# Create a new model to output the second Dense layer

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

outputs=model.get\_layer('dense2').output)

# Get the output of the second Dense layer

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

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

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

Ans:-

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# 1. Define the custom MSE loss function

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

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

# 2. Create a simple model

model = Sequential([

Dense(32, activation='relu', input\_shape=(10,)),

Dense(1)

])

# 3. Compile the model using the custom loss

model.compile(optimizer='adam', loss=custom\_mse, metrics=['mae'])

# 4. Summary

model.summary()

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

Ans-

import tensorflow as tf

from tensorflow.keras.models import Sequential, load\_model, Model

from tensorflow.keras.layers import Dense

import numpy as np

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

# 1️⃣ Create sample data

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

X = np.random.rand(100, 10)

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

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

# 2️⃣ Create and train a simple model

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

model = Sequential([

    Dense(32, activation='relu', input\_shape=(10,)),

    Dense(1)

])

model.compile(optimizer='adam', loss='mse')

model.fit(X, y, epochs=2, batch\_size=10, verbose=1)

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

# 3️⃣ Save the model

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

# Keras native format (recommended)

model.save('my\_model.keras')

# SavedModel format

model.export('my\_model\_saved')

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

# 4️⃣ Load the model back

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

# From Keras native format

model\_keras = load\_model('my\_model.keras')

print("Keras native format model loaded successfully!")

# From SavedModel

# Use TFSMLayer to load the SavedModel in Keras 3

loaded\_saved\_model\_layer = tf.keras.layers.TFSMLayer('my\_model\_saved', call\_endpoint='serving\_default')

# To use it for prediction, you can wrap it in a Model

input\_layer = tf.keras.Input(shape=(10,)) # Define input shape based on your model

model\_saved = Model(inputs=input\_layer, outputs=loaded\_saved\_model\_layer(input\_layer))

print("SavedModel format model loaded successfully using TFSMLayer!")

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

# 5️⃣ Check predictions are consistent

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

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

pred\_original = model.predict(sample\_input)

pred\_keras = model\_keras.predict(sample\_input)

pred\_saved = model\_saved.predict(sample\_input)

print("\nOriginal model prediction:", pred\_original)

print("Keras loaded model prediction:", pred\_keras)

print("SavedModel loaded model prediction:", pred\_saved)

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

Ans:-

import numpy as np

import tensorflow as tf

# logits

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

# --- Manual softmax ---

exp\_logits = np.exp(logits)

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

# --- TensorFlow softmax ---

logits\_tf = tf.constant([2.0, 1.0, 0.1])

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

# --- Print results ---

print("Manual softmax:", softmax\_manual)

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