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}')
```

2. **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}')
```

3. **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()
```

4. **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}")
```

5. **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()
```

6. 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)
```

7. 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)
```

8. 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)
```

9. 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)
```

10. 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()
```

11. 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}")
```

12. 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')
```

13. 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()
```

14. 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}")
```

15. 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
```

```python
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()
```

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

```python
# 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()
```

17. **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)
```

18. 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}")
```

19. 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}")
```

20. 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()))
```