```
# ---------------- Cell 1: Imports ----------------
import numpy as np                                   # numerical arrays & operations (foundation for ML code)
import tensorflow as tf                              # TensorFlow library (includes Keras high-level API)
from tensorflow.keras.datasets import mnist          # built-in MNIST dataset loader (handwritten digits)
from tensorflow.keras.models import Sequential       # Sequential model: stack layers linearly
from tensorflow.keras.layers import Dense            # Dense = fully-connected layer
import matplotlib.pyplot as plt                      # plotting library for visualizing results
```

```
# ---------------- Cell 2: Load and preprocess MNIST ----------------
# Informational print so user knows data loading has started — helpful in long scripts or remote runs.
print("[INFO] accessing MNIST...")                    # prints a small status message to the console

# Load MNIST dataset into training and test sets.
# (x_train, y_train) holds training images and labels, (x_test, y_test) holds testing images and labels.
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Reshape images from (N, 28, 28) to (N, 784) because we are using a feedforward network (not CNN).
# - x_train.shape[0] is the number of training samples (60000).
# - '-1' in reshape flattens the 28x28 image into a vector of length 784.
# Convert to float32 to ensure compatibility with TF ops and divide by 255 to normalize pixel values to [0,1].
x_train = x_train.reshape((x_train.shape[0], -1)).astype('float32') / 255
x_test  = x_test.reshape((x_test.shape[0], -1)).astype('float32') / 255

# Note on normalization:
# Normalizing to [0,1] stabilizes and accelerates training because inputs are in a small numeric range.
# Converting to float32 avoids integer division and ensures consistent numeric precision for TF.
```

```
[INFO] accessing MNIST...
```

```
# ---------------- Cell 3: Preserve original integer labels and one-hot encode ----------------
# Re-load the dataset labels separately to ensure we have the original integer labels.
# (This step avoids accidental double one-hot encoding if code is re-run multiple times in the same session.)
(_, y_train_original), (_, y_test_original) = mnist.load_data()

# Convert integer class labels (0..9) into one-hot vectors of length 10.
# Example: label 3 -> [0,0,0,1,0,0,0,0,0,0]
# We use categorical labels here because the model's final activation will be softmax and
# loss will be categorical_crossentropy which expects one-hot targets.
y_train = tf.keras.utils.to_categorical(y_train_original, num_classes=10)
y_test  = tf.keras.utils.to_categorical(y_test_original,  num_classes=10)
```

```
# ---------------- Cell 4: Define the feedforward neural network architecture ----------------
# We create a simple fully-connected (MLP) model using Keras Sequential API.
# The input will be 784-dimensional (flattened 28x28 image).
model = Sequential([
    tf.keras.Input(shape=(784,)),                    # explicit input layer with shape (784,)
    Dense(64, activation='relu'),                    # first hidden layer: 64 units, ReLU activation
    Dense(64, activation='relu'),                    # second hidden layer: 64 units, ReLU activation
    Dense(64, activation='relu'),                    # third hidden layer: 64 units, ReLU activation
    Dense(10, activation='softmax')                  # output layer: 10 units (one per class), softmax -> probabilities
])

# Explanation of architecture choices:
# - Dense layers: each neuron is connected to all inputs from previous layer (typical for MLPs).
# - 64 units is a modest hidden layer size suitable for MNIST and educational purposes.
# - ReLU activation prevents vanishing gradients and is computationally efficient.
# - Softmax at the end gives a probability distribution across the 10 digit classes.
```

```
# ---------------- Cell 5: Compile the model ----------------
# Before training, we configure the learning process:
# - optimizer: algorithm to update weights (SGD here; alternatives include 'adam', 'rmsprop')
# - loss: 'categorical_crossentropy' is appropriate for multi-class one-hot targets
# - metrics: list of metrics to track; 'accuracy' is common for classification
model.compile(optimizer='sgd',                         # Stochastic Gradient Descent optimizer
              loss='categorical_crossentropy',        # training objective (cross-entropy for classification)
              metrics=['accuracy'])                   # measure accuracy during training & evaluation

# Note:
# - SGD can work fine on MNIST but may be slower than Adam; it's used often in teaching to show basics.
# - You can tune optimizer hyperparameters (learning rate, momentum) via optimizer arguments if needed.
```

```
# ---------------- Cell 6: Train the model ----------------
# model.fit performs training:
# - x_train, y_train: training data & labels
# - epochs: how many times to iterate over the full training set
# - batch_size: number of samples per gradient update
# - validation_data: tuple used to evaluate validation metrics after each epoch
H = model.fit(x_train, y_train,
              epochs=15,                            # run 15 epochs (passes through dataset)
              batch_size=32,                        # update weights after each batch of 32 samples
              validation_data=(x_test, y_test))     # evaluate on test set after each epoch (as validation)

# Output during training:
# Keras prints per-epoch metrics like loss and accuracy; the returned History object contains them for plotting.
```

```
Epoch 1/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.6258 - loss: 1.2738 - val_accuracy: 0.9012 - val_loss: 0.3392
Epoch 2/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9113 - loss: 0.3048 - val_accuracy: 0.9292 - val_loss: 0.2424
Epoch 3/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9315 - loss: 0.2367 - val_accuracy: 0.9353 - val_loss: 0.2140
Epoch 4/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9432 - loss: 0.1961 - val_accuracy: 0.9487 - val_loss: 0.1678
Epoch 5/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9500 - loss: 0.1664 - val_accuracy: 0.9543 - val_loss: 0.1505
Epoch 6/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9580 - loss: 0.1481 - val_accuracy: 0.9573 - val_loss: 0.1403
Epoch 7/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9614 - loss: 0.1301 - val_accuracy: 0.9612 - val_loss: 0.1274
Epoch 8/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9664 - loss: 0.1157 - val_accuracy: 0.9627 - val_loss: 0.1251
Epoch 9/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9686 - loss: 0.1063 - val_accuracy: 0.9650 - val_loss: 0.1130
Epoch 10/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9724 - loss: 0.0965 - val_accuracy: 0.9661 - val_loss: 0.1127
Epoch 11/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9730 - loss: 0.0932 - val_accuracy: 0.9704 - val_loss: 0.1051
Epoch 12/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9759 - loss: 0.0866 - val_accuracy: 0.9681 - val_loss: 0.1047
Epoch 13/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9763 - loss: 0.0812 - val_accuracy: 0.9696 - val_loss: 0.1002
Epoch 14/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9781 - loss: 0.0737 - val_accuracy: 0.9707 - val_loss: 0.0961
Epoch 15/15
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9798 - loss: 0.0708 - val_accuracy: 0.9702 - val_loss: 0.0978
```

```
# ---------------- Cell 7: Evaluate the model ----------------
# Evaluate model on the test dataset (returns loss and metric values defined in compile())
test_loss, test_accuracy = model.evaluate(x_test, y_test)

# Print a nicely formatted test accuracy percentage
print(f'Test accuracy: {test_accuracy*100:.2f}%')      # multiply by 100 to convert to percentage
```

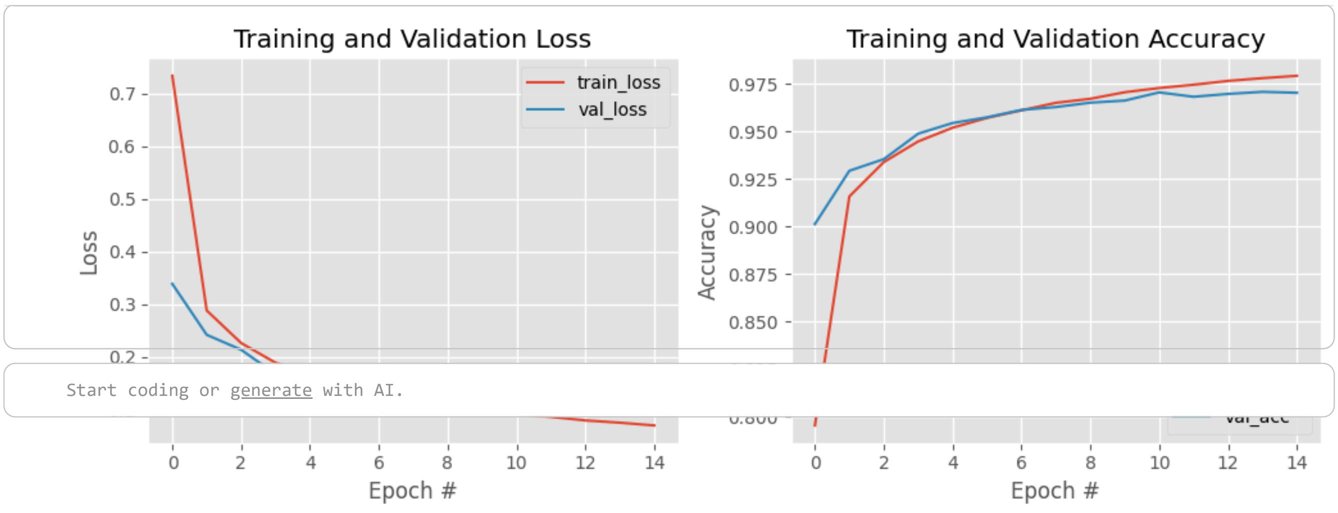
```
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9650 - loss: 0.1129
Test accuracy: 97.02%
```

```
# ---------------- Cell 8: Plot training loss and accuracy ----------------
# Visualize loss and accuracy progress over epochs using matplotlib.
plt.style.use("ggplot")                                # use a clean plotting style
plt.figure(figsize=(10, 4))                            # create a figure with a specified size (width, height)

# Plot training loss in the left subplot
plt.subplot(1, 2, 1)                                   # 1 row, 2 columns, first subplot
plt.plot(H.history['loss'], label="train_loss")        # plot training loss curve
plt.plot(H.history['val_loss'], label="val_loss")      # plot validation loss curve (if available)
plt.title("Training and Validation Loss")              # title for the subplot
plt.xlabel("Epoch #")                                  # x-axis label
plt.ylabel("Loss")                                     # y-axis label
plt.legend()                                           # show legend to distinguish curves

# Plot training accuracy in the right subplot
plt.subplot(1, 2, 2)                                   # second subplot (accuracy)
plt.plot(H.history['accuracy'], label="train_acc")     # plot training accuracy curve
plt.plot(H.history['val_accuracy'], label="val_acc")  # plot validation accuracy curve (if available)
plt.title("Training and Validation Accuracy")          # title for subplot
plt.xlabel("Epoch #")                                  # x-axis label
plt.ylabel("Accuracy")                                 # y-axis label
plt.legend()                                           # show legend

plt.tight_layout()                                     # adjust spacing so subplots don't overlap
plt.show()                                             # render the plots
```



```
Start coding or generate with AI.
```