```
import tensorflow as tf
from tensorflow import keras
import numpy as np
import matplotlib.pyplot as plt

print("TensorFlow version:", tf.__version__)
```

```
TensorFlow version: 2.19.0
```

```
# Load the dataset
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# --- Data Preprocessing ---

# Normalize the images to be between 0 and 1
x_train = x_train.astype("float32") / 255.0
x_test = x_test.astype("float32") / 255.0

# Flatten the images from 28x28 to a 784-element vector
x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)

# One-hot encode the labels
# There are 10 classes (digits 0-9)
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)

print("Training data shape:", x_train.shape)
print("Test data shape:", x_test.shape)
print("Sample one-hot encoded label:", y_train[0])
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Training data shape: (60000, 784)
Test data shape: (10000, 784)
Sample one-hot encoded label: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

```
# Define the model using the Sequential API
model = keras.Sequential([
    # Input layer: specify the input shape for the first layer
    keras.layers.Dense(128, activation='relu', input_shape=(784,)),

    # First hidden layer
    keras.layers.Dense(128, activation='relu'),

    # Output layer: 10 neurons for 10 classes, softmax for probabilities
    keras.layers.Dense(10, activation='softmax')
])

# Print a summary of the model's architecture
model.summary()
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/core/dense.py:93: UserWarning: Do not pass an `input_shape`
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Model: "sequential"
```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 128)	100,480
dense_1 (Dense)	(None, 128)	16,512
dense_2 (Dense)	(None, 10)	1,290

```
Total params: 118,282 (462.04 KB)
Trainable params: 118,282 (462.04 KB)
Non-trainable params: 0 (0.00 B)
```

```
# Compile the model
model.compile(optimizer='sgd',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Train the model
# We'll save the training history to plot it later
history = model.fit(x_train, y_train,
                    batch_size=128,
                    epochs=40,
                    verbose=1,
                    validation_data=(x_test, y_test))
```

```
Epoch 1/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.4921 - loss: 1.7419 - val_accuracy: 0.8492 - val_loss: 0.61
Epoch 2/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8566 - loss: 0.5630 - val_accuracy: 0.8909 - val_loss: 0.40
Epoch 3/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8878 - loss: 0.4100 - val_accuracy: 0.9025 - val_loss: 0.34
Epoch 4/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.8995 - loss: 0.3562 - val_accuracy: 0.9085 - val_loss: 0.31
Epoch 5/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9072 - loss: 0.3219 - val_accuracy: 0.9141 - val_loss: 0.29
Epoch 6/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9116 - loss: 0.3091 - val_accuracy: 0.9194 - val_loss: 0.28
Epoch 7/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9198 - loss: 0.2831 - val_accuracy: 0.9234 - val_loss: 0.26
Epoch 8/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9225 - loss: 0.2728 - val_accuracy: 0.9261 - val_loss: 0.25
Epoch 9/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9256 - loss: 0.2630 - val_accuracy: 0.9297 - val_loss: 0.24
Epoch 10/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 9ms/step - accuracy: 0.9285 - loss: 0.2482 - val_accuracy: 0.9323 - val_loss: 0.24
Epoch 11/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9298 - loss: 0.2470 - val_accuracy: 0.9347 - val_loss: 0.23
Epoch 12/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9329 - loss: 0.2392 - val_accuracy: 0.9362 - val_loss: 0.22
Epoch 13/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9381 - loss: 0.2217 - val_accuracy: 0.9387 - val_loss: 0.21
Epoch 14/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9392 - loss: 0.2179 - val_accuracy: 0.9406 - val_loss: 0.21
Epoch 15/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9408 - loss: 0.2101 - val_accuracy: 0.9418 - val_loss: 0.20
Epoch 16/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.9431 - loss: 0.2011 - val_accuracy: 0.9435 - val_loss: 0.19
Epoch 17/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9442 - loss: 0.1988 - val_accuracy: 0.9448 - val_loss: 0.19
Epoch 18/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9456 - loss: 0.1941 - val_accuracy: 0.9461 - val_loss: 0.18
Epoch 19/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9480 - loss: 0.1833 - val_accuracy: 0.9480 - val_loss: 0.18
Epoch 20/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9485 - loss: 0.1814 - val_accuracy: 0.9488 - val_loss: 0.17
Epoch 21/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9506 - loss: 0.1757 - val_accuracy: 0.9506 - val_loss: 0.17
Epoch 22/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9536 - loss: 0.1661 - val_accuracy: 0.9512 - val_loss: 0.17
Epoch 23/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9531 - loss: 0.1657 - val_accuracy: 0.9516 - val_loss: 0.16
Epoch 24/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9538 - loss: 0.1612 - val_accuracy: 0.9531 - val_loss: 0.16
Epoch 25/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9567 - loss: 0.1517 - val_accuracy: 0.9535 - val_loss: 0.15
Epoch 26/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9569 - loss: 0.1522 - val_accuracy: 0.9541 - val_loss: 0.15
Epoch 27/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9568 - loss: 0.1466 - val_accuracy: 0.9567 - val_loss: 0.15
Epoch 28/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.9605 - loss: 0.1415 - val_accuracy: 0.9569 - val_loss: 0.14
Epoch 29/40
469/469 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9603 - loss: 0.1389 - val_accuracy: 0.9573 - val_loss: 0.14
```

```
# Evaluate the model on the test set
score = model.evaluate(x_test, y_test, verbose=0)

print(f"Test Loss: {score[0]:.4f}")
print(f"Test Accuracy: {score[1]:.4f}")
```

```
Test Loss: 0.1219
Test Accuracy: 0.9637
```

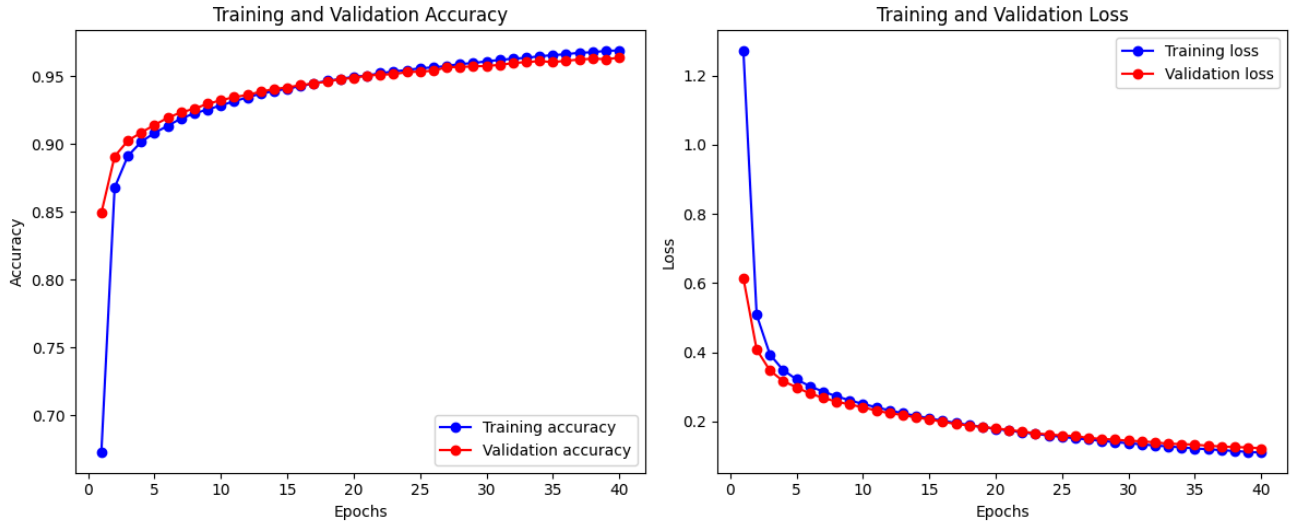
```
# Get training and validation accuracy and loss from history object
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
loss = history.history['loss']
val_loss = history.history['val_loss']

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

# Plot training and validation accuracy
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(epochs, acc, 'bo-', label='Training accuracy')
plt.plot(epochs, val_acc, 'ro-', label='Validation accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
```

```
# Plot training and validation loss
plt.subplot(1, 2, 2)
plt.plot(epochs, loss, 'bo-', label='Training loss')
plt.plot(epochs, val_loss, 'ro-', label='Validation loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.show()
```




```
# Load the dataset again to access the original unflattened data
(x_train_original, y_train_original), (x_test_original, y_test_original) = keras.datasets.mnist.load_data()

# Select the first image from the original training data
image_to_plot = x_train_original[0]

# Plot the image
plt.matshow(image_to_plot, cmap=plt.get_cmap('gray'))
plt.title("Sample Image from Training Data")
plt.show()
```