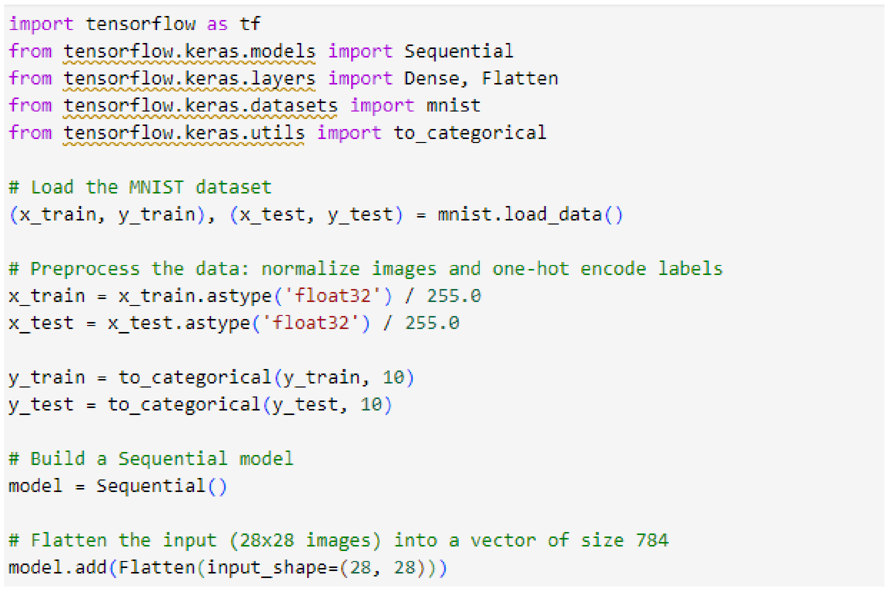
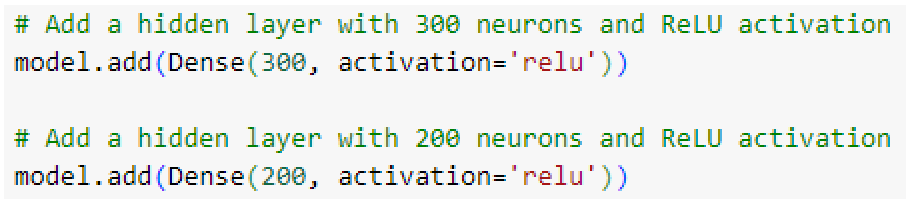


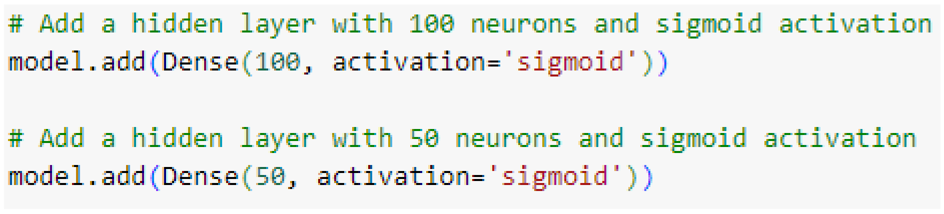
```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data: normalize images and one-hot encode labels
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build a Sequential model
model = Sequential()

# Flatten the input (28x28 images) into a vector of size 784
model.add(Flatten(input_shape=(28, 28)))
```

As per the assignment direction, this code is primarily duplicated from the in-class example provided. From this, I take the Sequential API setup and use it without any alterations / variations.



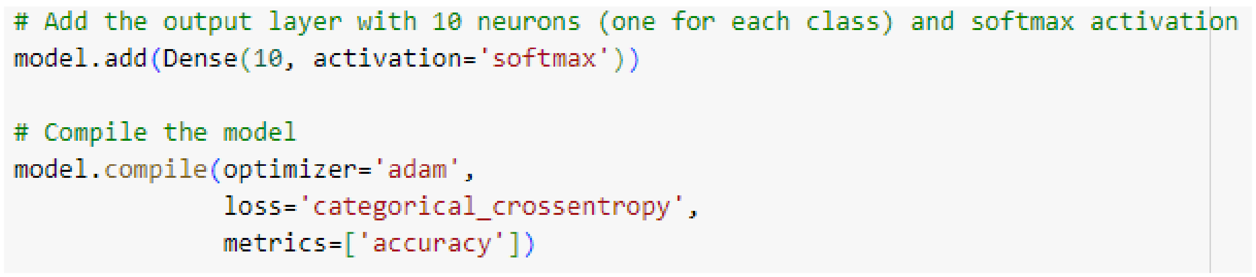
```
# Add a hidden layer with 300 neurons and ReLU activation
model.add(Dense(300, activation='relu'))

# Add a hidden layer with 200 neurons and ReLU activation
model.add(Dense(200, activation='relu'))
```

Here is the first provided difference from the example code; a different number of neurons. They all fit in the same ReLU activation method. As well, this is adding more layers into the system then there were originally, hopefully adding some stimulation to provide a chance at higher accuracy.



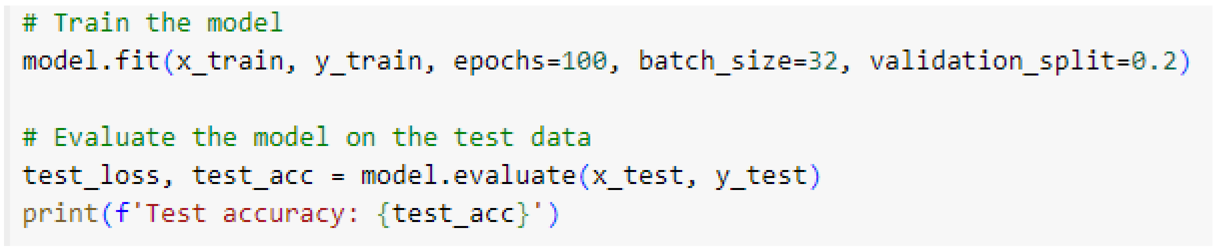
```
# Add a hidden layer with 100 neurons and sigmoid activation
model.add(Dense(100, activation='sigmoid'))

# Add a hidden layer with 50 neurons and sigmoid activation
model.add(Dense(50, activation='sigmoid'))
```

Here is the second provided difference from the example code; a different activation method. This variation in neuron activation methods should also help give the code a better, more ‘all-encompassing’ ability to verify data and classify it correctly. As well, this is adding even more layers into the system, which should help as previously mentioned.



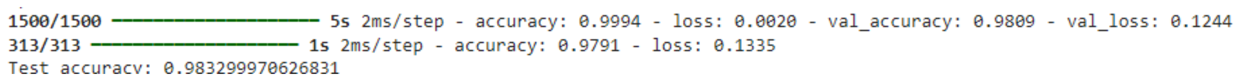
```python
# Add the output layer with 10 neurons (one for each class) and softmax activation
model.add(Dense(10, activation='softmax'))

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

This is all primarily the same as the provided code - keeping the density of the neurons the same is important. However, this acting as the output layer is important. Relatively unchanged, but still noteworthy.



```python
# Train the model
model.fit(x_train, y_train, epochs=100, batch_size=32, validation_split=0.2)

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')
```

This is the final change made to the code relatively speaking, which is the number of epochs increasing. From 5 to 100, this should provide the code more chances and more potential to catch the data in a more accurate method.

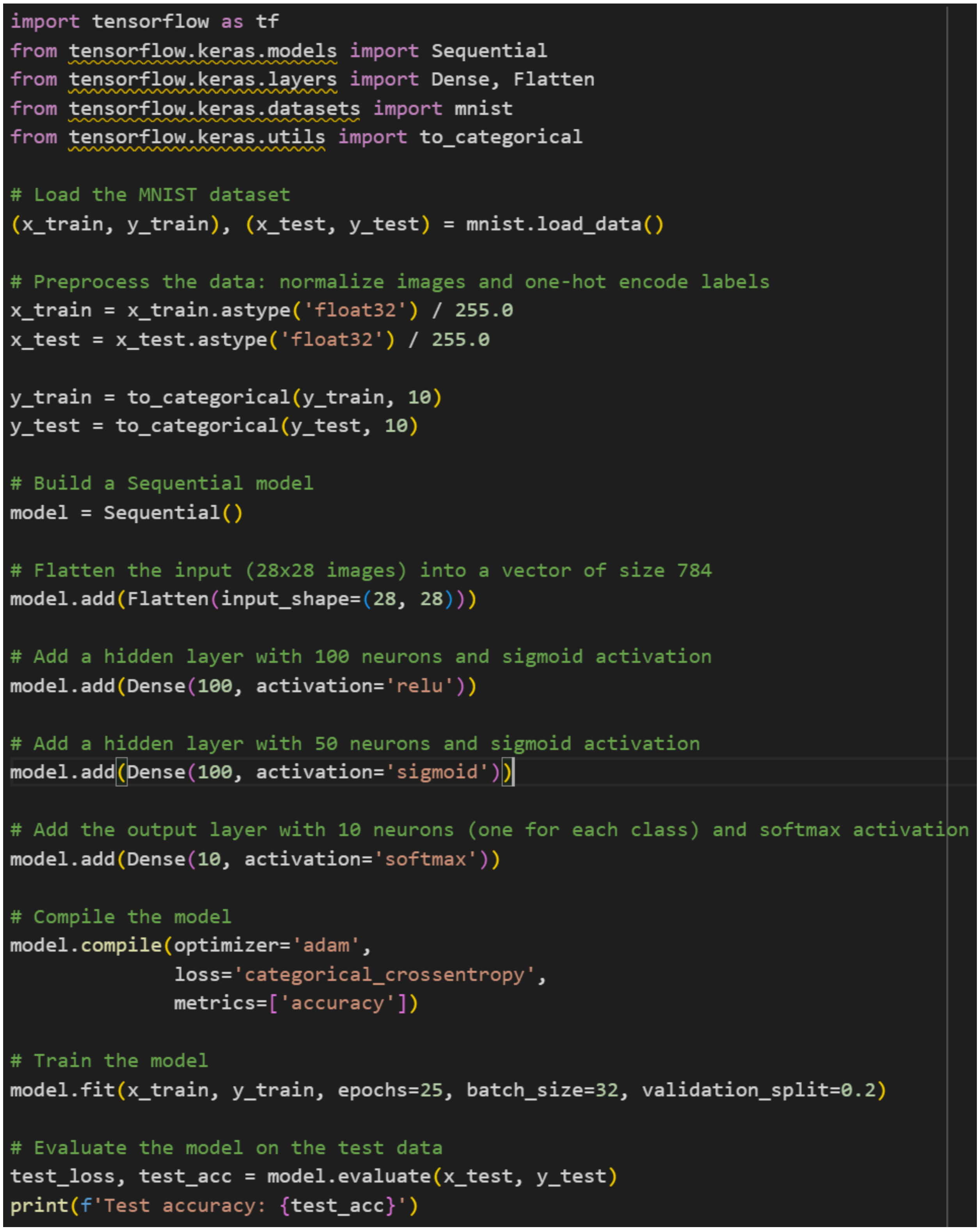


```
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9994 - loss: 0.0020 - val_accuracy: 0.9809 - val_loss: 0.1244
313/313 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9791 - loss: 0.1335
Test accuracy: 0.983299970626831
```

Without any other changes, the test accuracy seems to have hit a type of plateau. Simply adding more epochs likely won't change the result, or at least, will deliver diminishing returns. Going back and modifying how the code actually analyzes the data is most likely a more efficient method of raising this accuracy.

Through testing, it seems that adding an excessive amount of layers doesn't seem to help, rather, it seems to hinder the accuracy scores that the model can achieve. It also seems that having an excess of epochs can allow for "accuracy fluctuations", where no real improvement is being made. Having as many as 100 epochs allows for quite the large amount of this accuracy fluctuation.

Through extensive testing, my model presented seems to have produced the best results. The potential to change the optimizer model is there, but I'm unsure of the list of potential solutions to remedy this problem. A high 0.98 is the closest I've been able to achieve.



```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Load the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocess the data: normalize images and one-hot encode labels
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Build a Sequential model
model = Sequential()

# Flatten the input (28x28 images) into a vector of size 784
model.add(Flatten(input_shape=(28, 28)))

# Add a hidden layer with 100 neurons and sigmoid activation
model.add(Dense(100, activation='relu'))

# Add a hidden layer with 50 neurons and sigmoid activation
model.add(Dense(100, activation='sigmoid'))

# Add the output layer with 10 neurons (one for each class) and softmax activation
model.add(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=25, batch_size=32, validation_split=0.2)

# Evaluate the model on the test data
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'Test accuracy: {test_acc}')
```