Week 4 Programming Assignment

November 20, 2020

1 Programming Assignment

1.1 Saving and loading models, with application to the EuroSat dataset

1.1.1 Instructions

In this notebook, you will create a neural network that classifies land uses and land covers from satellite imagery. You will save your model using Tensorflow's callbacks and reload it later. You will also load in a pre-trained neural network classifier and compare performance with it.

Some code cells are provided for you in the notebook. You should avoid editing provided code, and make sure to execute the cells in order to avoid unexpected errors. Some cells begin with the line:

`#### GRADED CELL ####`

Don't move or edit this first line - this is what the automatic grader looks for to recognise graded cells. These cells require you to write your own code to complete them, and are automatically graded when you submit the notebook. Don't edit the function name or signature provided in these cells, otherwise the automatic grader might not function properly. Inside these graded cells, you can use any functions or classes that are imported below, but make sure you don't use any variables that are outside the scope of the function.

1.1.2 How to submit

Complete all the tasks you are asked for in the worksheet. When you have finished and are happy with your code, press the **Submit Assignment** button at the top of this notebook.

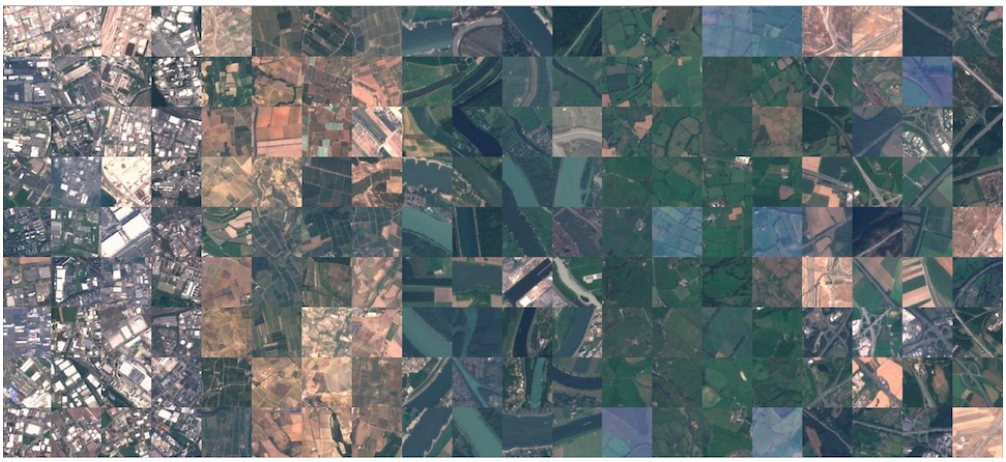
1.1.3 Let's get started!

We'll start running some imports, and loading the dataset. Do not edit the existing imports in the following cell. If you would like to make further Tensorflow imports, you should add them here.

In [1]:
```
#### PACKAGE IMPORTS ####

# Run this cell first to import all required packages. Do not make any imports elsewhe

import tensorflow as tf
from tensorflow.keras.preprocessing.image import load_img, img_to_array
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
```



EuroSAT overview image

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
import os
import numpy as np
import pandas as pd

# If you would like to make further imports from tensorflow, add them here
```

The EuroSAT dataset In this assignment, you will use the EuroSAT dataset. It consists of 27000 labelled Sentinel-2 satellite images of different land uses: residential, industrial, highway, river, forest, pasture, herbaceous vegetation, annual crop, permanent crop and sea/lake. For a reference, see the following papers: - Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. Patrick Helber, Benjamin Bischke, Andreas Dengel, Damian Borth. IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, 2019. - Introducing EuroSAT: A Novel Dataset and Deep Learning Benchmark for Land Use and Land Cover Classification. Patrick Helber, Benjamin Bischke, Andreas Dengel. 2018 IEEE International Geoscience and Remote Sensing Symposium, 2018.

Your goal is to construct a neural network that classifies a satellite image into one of these 10 classes, as well as applying some of the saving and loading techniques you have learned in the previous sessions.

Import the data The dataset you will train your model on is a subset of the total data, with 4000 training images and 1000 testing images, with roughly equal numbers of each class. The code to import the data is provided below.

In [2]:
```
# Run this cell to import the Eurosat data

def load_eurosat_data():
    data_dir = 'data/'
    x_train = np.load(os.path.join(data_dir, 'x_train.npy'))
    y_train = np.load(os.path.join(data_dir, 'y_train.npy'))
    x_test  = np.load(os.path.join(data_dir, 'x_test.npy'))
```

```
    y_test  = np.load(os.path.join(data_dir, 'y_test.npy'))
    return (x_train, y_train), (x_test, y_test)

(x_train, y_train), (x_test, y_test) = load_eurosat_data()
x_train = x_train / 255.0
x_test = x_test / 255.0
```

Build the neural network model You can now construct a model to fit to the data. Using the Sequential API, build your model according to the following specifications:

- The model should use the input_shape in the function argument to set the input size in the first layer.
- The first layer should be a Conv2D layer with 16 filters, a 3x3 kernel size, a ReLU activation function and 'SAME' padding. Name this layer 'conv_1'.
- The second layer should also be a Conv2D layer with 8 filters, a 3x3 kernel size, a ReLU activation function and 'SAME' padding. Name this layer 'conv_2'.
- The third layer should be a MaxPooling2D layer with a pooling window size of 8x8. Name this layer 'pool_1'.
- The fourth layer should be a Flatten layer, named 'flatten'.
- The fifth layer should be a Dense layer with 32 units, a ReLU activation. Name this layer 'dense_1'.
- The sixth and final layer should be a Dense layer with 10 units and softmax activation. Name this layer 'dense_2'.

In total, the network should have 6 layers.

In [3]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.

def get_new_model(input_shape):
    """
    This function should build a Sequential model according to the above specification
    weights are initialised by providing the input_shape argument in the first layer,
    function argument.
    Your function should also compile the model with the Adam optimiser, sparse catego
    entropy loss function, and a single accuracy metric.
    """
    model = Sequential([
        Conv2D(filters = 16, input_shape = input_shape, kernel_size = (3, 3),
               activation = 'relu', padding = 'SAME', name = 'conv_1'),
        Conv2D(filters = 8, kernel_size = (3, 3), activation = 'relu',
               padding = 'SAME', name = 'conv_2'),
        MaxPooling2D(pool_size = (8, 8), name = 'pool_1'),
        Flatten(name = 'flatten'),
        Dense(units = 32, activation = 'relu', name = 'dense_1'),
        Dense(units = 10, activation = 'softmax', name = 'dense_2')
```

```
    ])

    model.compile(optimizer = 'adam', loss = 'sparse_categorical_crossentropy',
                  metrics = ['accuracy'])
    return model
```

Compile and evaluate the model

```
In [4]: # Run your function to create the model

        model = get_new_model(x_train[0].shape)

In [5]: # Run this cell to define a function to evaluate a model's test accuracy

        def get_test_accuracy(model, x_test, y_test):
            """Test model classification accuracy"""
            test_loss, test_acc = model.evaluate(x=x_test, y=y_test, verbose=0)
            print('accuracy: {acc:0.3f}'.format(acc=test_acc))

In [6]: # Print the model summary and calculate its initialised test accuracy

        model.summary()
        get_test_accuracy(model, x_test, y_test)
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_1 (Conv2D)              (None, 64, 64, 16)        448
_________________________________________________________________
conv_2 (Conv2D)              (None, 64, 64, 8)         1160
_________________________________________________________________
pool_1 (MaxPooling2D)        (None, 8, 8, 8)           0
_________________________________________________________________
flatten (Flatten)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                16416
_________________________________________________________________
dense_2 (Dense)              (None, 10)                330
=================================================================
Total params: 18,354
Trainable params: 18,354
Non-trainable params: 0
_________________________________________________________________
accuracy: 0.129
```

Create checkpoints to save model during training, with a criterion You will now create three callbacks: - `checkpoint_every_epoch`: checkpoint that saves the model weights every epoch during training - `checkpoint_best_only`: checkpoint that saves only the weights with the highest validation accuracy. Use the testing data as the validation data. - `early_stopping`: early stopping object that ends training if the validation accuracy has not improved in 3 epochs.

In [7]:
```
#### GRADED CELL ####

# Complete the following functions.
# Make sure to not change the function names or arguments.

def get_checkpoint_every_epoch():
    """
    This function should return a ModelCheckpoint object that:
    - saves the weights only at the end of every epoch
    - saves into a directory called 'checkpoints_every_epoch' inside the current worki
    - generates filenames in that directory like 'checkpoint_XXX' where
      XXX is the epoch number formatted to have three digits, e.g. 001, 002, 003, etc.
    """
    checkpoint_path = 'checkpoints_every_epoch/checkpoint_{epoch:03d}'
    checkpoint_every_epoch = ModelCheckpoint(filepath = checkpoint_path,
                                             save_freq = 'epoch',
                                             save_weights_only = True,
                                             verbose = 1)
    return checkpoint_every_epoch


def get_checkpoint_best_only():
    """
    This function should return a ModelCheckpoint object that:
    - saves only the weights that generate the highest validation (testing) accuracy
    - saves into a directory called 'checkpoints_best_only' inside the current working
    - generates a file called 'checkpoints_best_only/checkpoint'
    """
    checkpoint_path = 'checkpoints_best_only/checkpoint'
    checkpoint_best_only = ModelCheckpoint(filepath = checkpoint_path,
                                           save_weights_only = True,
                                           monitor = 'val_accuracy',
                                           save_best_only = True,
                                           verbose = 1)
    return checkpoint_best_only
```

In [8]:
```
#### GRADED CELL ####

# Complete the following function.
# Make sure to not change the function name or arguments.
```

```
def get_early_stopping():
    """
    This function should return an EarlyStopping callback that stops training when
    the validation (testing) accuracy has not improved in the last 3 epochs.
    HINT: use the EarlyStopping callback with the correct 'monitor' and 'patience'
    """
    early_stopping = EarlyStopping(monitor = 'val_accuracy',
                                   patience = 3,
                                   mode = 'max')
    return early_stopping
```

In [9]:
```
# Run this cell to create the callbacks

checkpoint_every_epoch = get_checkpoint_every_epoch()
checkpoint_best_only = get_checkpoint_best_only()
early_stopping = get_early_stopping()
```

Train model using the callbacks Now, you will train the model using the three callbacks you created. If you created the callbacks correctly, three things should happen: - At the end of every epoch, the model weights are saved into a directory called `checkpoints_every_epoch` - At the end of every epoch, the model weights are saved into a directory called `checkpoints_best_only` **only** if those weights lead to the highest test accuracy - Training stops when the testing accuracy has not improved in three epochs.

You should then have two directories: - A directory called `checkpoints_every_epoch` containing filenames that include `checkpoint_001`, `checkpoint_002`, etc with the `001`, `002` corresponding to the epoch - A directory called `checkpoints_best_only` containing filenames that include `checkpoint`, which contain only the weights leading to the highest testing accuracy

In [10]:
```
# Train model using the callbacks you just created

callbacks = [checkpoint_every_epoch, checkpoint_best_only, early_stopping]
model.fit(x_train, y_train, epochs=50, validation_data=(x_test, y_test), callbacks=cal
```

```
Train on 4000 samples, validate on 1000 samples
Epoch 1/50
3968/4000 [============================>.] - ETA: 0s - loss: 2.0410 - accuracy: 0.2220
Epoch 00001: saving model to checkpoints_every_epoch/checkpoint_001

Epoch 00001: val_accuracy improved from -inf to 0.35000, saving model to checkpoints_best_only/
4000/4000 [==============================] - 83s 21ms/sample - loss: 2.0373 - accuracy: 0.2230
Epoch 2/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.4732 - accuracy: 0.4252
Epoch 00002: saving model to checkpoints_every_epoch/checkpoint_002

Epoch 00002: val_accuracy improved from 0.35000 to 0.45900, saving model to checkpoints_best_on
4000/4000 [==============================] - 83s 21ms/sample - loss: 1.4716 - accuracy: 0.4252
Epoch 3/50
```

```
3968/4000 [============================>.] - ETA: 0s - loss: 1.3050 - accuracy: 0.5035
Epoch 00003: saving model to checkpoints_every_epoch/checkpoint_003

Epoch 00003: val_accuracy improved from 0.45900 to 0.49300, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 20ms/sample - loss: 1.3040 - accuracy: 0.5050
Epoch 4/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.2309 - accuracy: 0.5481
Epoch 00004: saving model to checkpoints_every_epoch/checkpoint_004

Epoch 00004: val_accuracy improved from 0.49300 to 0.52400, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 20ms/sample - loss: 1.2296 - accuracy: 0.5477
Epoch 5/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.1875 - accuracy: 0.5628
Epoch 00005: saving model to checkpoints_every_epoch/checkpoint_005

Epoch 00005: val_accuracy improved from 0.52400 to 0.56100, saving model to checkpoints_best_or
4000/4000 [==============================] - 81s 20ms/sample - loss: 1.1869 - accuracy: 0.5638
Epoch 6/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.1256 - accuracy: 0.5862
Epoch 00006: saving model to checkpoints_every_epoch/checkpoint_006

Epoch 00006: val_accuracy did not improve from 0.56100
4000/4000 [==============================] - 82s 21ms/sample - loss: 1.1236 - accuracy: 0.5872
Epoch 7/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.0898 - accuracy: 0.6071
Epoch 00007: saving model to checkpoints_every_epoch/checkpoint_007

Epoch 00007: val_accuracy improved from 0.56100 to 0.56300, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 21ms/sample - loss: 1.0921 - accuracy: 0.6062
Epoch 8/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.0812 - accuracy: 0.6026
Epoch 00008: saving model to checkpoints_every_epoch/checkpoint_008

Epoch 00008: val_accuracy improved from 0.56300 to 0.56800, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 20ms/sample - loss: 1.0784 - accuracy: 0.6040
Epoch 9/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.0447 - accuracy: 0.6197
Epoch 00009: saving model to checkpoints_every_epoch/checkpoint_009

Epoch 00009: val_accuracy improved from 0.56800 to 0.59800, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 21ms/sample - loss: 1.0443 - accuracy: 0.6198
Epoch 10/50
3968/4000 [============================>.] - ETA: 0s - loss: 1.0105 - accuracy: 0.6283
Epoch 00010: saving model to checkpoints_every_epoch/checkpoint_010

Epoch 00010: val_accuracy did not improve from 0.59800
4000/4000 [==============================] - 83s 21ms/sample - loss: 1.0145 - accuracy: 0.6265
Epoch 11/50
```

```
3968/4000 [============================>.] - ETA: 0s - loss: 0.9958 - accuracy: 0.6419
Epoch 00011: saving model to checkpoints_every_epoch/checkpoint_011

Epoch 00011: val_accuracy did not improve from 0.59800
4000/4000 [==============================] - 80s 20ms/sample - loss: 0.9935 - accuracy: 0.6423
Epoch 12/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.9816 - accuracy: 0.6426
Epoch 00012: saving model to checkpoints_every_epoch/checkpoint_012

Epoch 00012: val_accuracy improved from 0.59800 to 0.61800, saving model to checkpoints_best_or
4000/4000 [==============================] - 83s 21ms/sample - loss: 0.9804 - accuracy: 0.6430
Epoch 13/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.9373 - accuracy: 0.6575
Epoch 00013: saving model to checkpoints_every_epoch/checkpoint_013

Epoch 00013: val_accuracy did not improve from 0.61800
4000/4000 [==============================] - 80s 20ms/sample - loss: 0.9364 - accuracy: 0.6585
Epoch 14/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.9053 - accuracy: 0.6628
Epoch 00014: saving model to checkpoints_every_epoch/checkpoint_014

Epoch 00014: val_accuracy improved from 0.61800 to 0.63200, saving model to checkpoints_best_or
4000/4000 [==============================] - 82s 21ms/sample - loss: 0.9044 - accuracy: 0.6630
Epoch 15/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.8854 - accuracy: 0.6774
Epoch 00015: saving model to checkpoints_every_epoch/checkpoint_015

Epoch 00015: val_accuracy did not improve from 0.63200
4000/4000 [==============================] - 82s 20ms/sample - loss: 0.8847 - accuracy: 0.6773
Epoch 16/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.8559 - accuracy: 0.6888
Epoch 00016: saving model to checkpoints_every_epoch/checkpoint_016

Epoch 00016: val_accuracy did not improve from 0.63200
4000/4000 [==============================] - 82s 21ms/sample - loss: 0.8584 - accuracy: 0.6873
Epoch 17/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.8363 - accuracy: 0.6908
Epoch 00017: saving model to checkpoints_every_epoch/checkpoint_017

Epoch 00017: val_accuracy improved from 0.63200 to 0.66200, saving model to checkpoints_best_or
4000/4000 [==============================] - 85s 21ms/sample - loss: 0.8365 - accuracy: 0.6908
Epoch 18/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.7998 - accuracy: 0.7069
Epoch 00018: saving model to checkpoints_every_epoch/checkpoint_018

Epoch 00018: val_accuracy improved from 0.66200 to 0.67300, saving model to checkpoints_best_or
4000/4000 [==============================] - 84s 21ms/sample - loss: 0.7990 - accuracy: 0.7075
Epoch 19/50
```

```
3968/4000 [============================>.] - ETA: 0s - loss: 0.7725 - accuracy: 0.7175
Epoch 00019: saving model to checkpoints_every_epoch/checkpoint_019

Epoch 00019: val_accuracy did not improve from 0.67300
4000/4000 [==============================] - 84s 21ms/sample - loss: 0.7750 - accuracy: 0.7168
Epoch 20/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.7693 - accuracy: 0.7162
Epoch 00020: saving model to checkpoints_every_epoch/checkpoint_020

Epoch 00020: val_accuracy improved from 0.67300 to 0.67500, saving model to checkpoints_best_or
4000/4000 [==============================] - 83s 21ms/sample - loss: 0.7726 - accuracy: 0.7145
Epoch 21/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.7327 - accuracy: 0.7245
Epoch 00021: saving model to checkpoints_every_epoch/checkpoint_021

Epoch 00021: val_accuracy did not improve from 0.67500
4000/4000 [==============================] - 84s 21ms/sample - loss: 0.7310 - accuracy: 0.7260
Epoch 22/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.7336 - accuracy: 0.7233
Epoch 00022: saving model to checkpoints_every_epoch/checkpoint_022

Epoch 00022: val_accuracy did not improve from 0.67500
4000/4000 [==============================] - 84s 21ms/sample - loss: 0.7337 - accuracy: 0.7232
Epoch 23/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.7197 - accuracy: 0.7354
Epoch 00023: saving model to checkpoints_every_epoch/checkpoint_023

Epoch 00023: val_accuracy improved from 0.67500 to 0.67800, saving model to checkpoints_best_or
4000/4000 [==============================] - 84s 21ms/sample - loss: 0.7175 - accuracy: 0.7358
Epoch 24/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.6923 - accuracy: 0.7419
Epoch 00024: saving model to checkpoints_every_epoch/checkpoint_024

Epoch 00024: val_accuracy did not improve from 0.67800
4000/4000 [==============================] - 82s 20ms/sample - loss: 0.6938 - accuracy: 0.7412
Epoch 25/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.6929 - accuracy: 0.7432
Epoch 00025: saving model to checkpoints_every_epoch/checkpoint_025

Epoch 00025: val_accuracy improved from 0.67800 to 0.70100, saving model to checkpoints_best_or
4000/4000 [==============================] - 83s 21ms/sample - loss: 0.6928 - accuracy: 0.7427
Epoch 26/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.6850 - accuracy: 0.7409
Epoch 00026: saving model to checkpoints_every_epoch/checkpoint_026

Epoch 00026: val_accuracy did not improve from 0.70100
4000/4000 [==============================] - 82s 20ms/sample - loss: 0.6844 - accuracy: 0.7402
Epoch 27/50
```

```
3968/4000 [============================>.] - ETA: 0s - loss: 0.6607 - accuracy: 0.7535
Epoch 00027: saving model to checkpoints_every_epoch/checkpoint_027

Epoch 00027: val_accuracy did not improve from 0.70100
4000/4000 [==============================] - 82s 21ms/sample - loss: 0.6609 - accuracy: 0.7535
Epoch 28/50
3968/4000 [============================>.] - ETA: 0s - loss: 0.6747 - accuracy: 0.7533
Epoch 00028: saving model to checkpoints_every_epoch/checkpoint_028

Epoch 00028: val_accuracy did not improve from 0.70100
4000/4000 [==============================] - 83s 21ms/sample - loss: 0.6756 - accuracy: 0.7523
```

```
Out[10]: <tensorflow.python.keras.callbacks.History at 0x7f88d856e6a0>
```

Create new instance of model and load on both sets of weights Now you will use the weights you just saved in a fresh model. You should create two functions, both of which take a freshly instantiated model instance: - `model_last_epoch` should contain the weights from the latest saved epoch - `model_best_epoch` should contain the weights from the saved epoch with the highest testing accuracy

Hint: use the `tf.train.latest_checkpoint` *function to get the filename of the latest saved checkpoint file. Check the docs here.*

```
In [11]: #### GRADED CELL ####

         # Complete the following functions.
         # Make sure to not change the function name or arguments.

         def get_model_last_epoch(model):
             """
             This function should create a new instance of the CNN you created earlier,
             load on the weights from the last training epoch, and return this model.
             """
             checkpoint_last_epoch = tf.train.latest_checkpoint(checkpoint_dir = 'checkpoints_
                                                                latest_filename = None)
             model.load_weights(checkpoint_last_epoch)
             return model


         def get_model_best_epoch(model):
             """
             This function should create a new instance of the CNN you created earlier, load
             on the weights leading to the highest validation accuracy, and return this model.
             """
             checkpoint_best_epoch = tf.train.latest_checkpoint(checkpoint_dir = 'checkpoints_
                                                                latest_filename = None)
             model.load_weights(checkpoint_best_epoch)
             return model
```

```
In [15]: # Run this cell to create two models: one with the weights from the last training
         # epoch, and one with the weights leading to the highest validation (testing) accurac
         # Verify that the second has a higher validation (testing) accuarcy.

         model_last_epoch = get_model_last_epoch(get_new_model(x_train[0].shape))
         model_best_epoch = get_model_best_epoch(get_new_model(x_train[0].shape))
         print('Model with last epoch weights:')
         get_test_accuracy(model_last_epoch, x_test, y_test)
         print('')
         print('Model with best epoch weights:')
         get_test_accuracy(model_best_epoch, x_test, y_test)
```

```
Model with last epoch weights:
accuracy: 0.693

Model with best epoch weights:
accuracy: 0.701
```

Load, from scratch, a model trained on the EuroSat dataset. In your workspace, you will find another model trained on the `EuroSAT` dataset in `.h5` format. This model is trained on a larger subset of the EuroSAT dataset and has a more complex architecture. The path to the model is `models/EuroSatNet.h5`. See how its testing accuracy compares to your model!

```
In [16]: #### GRADED CELL ####

         # Complete the following functions.
         # Make sure to not change the function name or arguments.

         def get_model_eurosatnet():
             """
             This function should return the pretrained EuroSatNet.h5 model.
             """
             model = load_model('models/EuroSatNet.h5')
             return model
```

```
In [17]: # Run this cell to print a summary of the EuroSatNet model, along with its validation

         model_eurosatnet = get_model_eurosatnet()
         model_eurosatnet.summary()
         get_test_accuracy(model_eurosatnet, x_test, y_test)
```

```
Model: "sequential_21"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_1 (Conv2D)              (None, 64, 64, 16)        448
```

```
_________________________________________________________________
conv_2 (Conv2D)              (None, 64, 64, 16)        6416
_________________________________________________________________
pool_1 (MaxPooling2D)        (None, 32, 32, 16)        0
_________________________________________________________________
conv_3 (Conv2D)              (None, 32, 32, 16)        2320
_________________________________________________________________
conv_4 (Conv2D)              (None, 32, 32, 16)        6416
_________________________________________________________________
pool_2 (MaxPooling2D)        (None, 16, 16, 16)        0
_________________________________________________________________
conv_5 (Conv2D)              (None, 16, 16, 16)        2320
_________________________________________________________________
conv_6 (Conv2D)              (None, 16, 16, 16)        6416
_________________________________________________________________
pool_3 (MaxPooling2D)        (None, 8, 8, 16)          0
_________________________________________________________________
conv_7 (Conv2D)              (None, 8, 8, 16)          2320
_________________________________________________________________
conv_8 (Conv2D)              (None, 8, 8, 16)          6416
_________________________________________________________________
pool_4 (MaxPooling2D)        (None, 4, 4, 16)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                8224
_________________________________________________________________
dense_2 (Dense)              (None, 10)                330
=================================================================
Total params: 41,626
Trainable params: 41,626
Non-trainable params: 0
_________________________________________________________________
accuracy: 0.810
```

Congratulations for completing this programming assignment! You're now ready to move on to the capstone project for this course.