Flowers_DataAug

September 12, 2021

1 Image Classification using tf.keras

Run in Google Colab

View source on GitHub

In this Colab you will classify images of flowers. You will build an image classifier using `tf.keras.Sequential` model and load data using `tf.keras.preprocessing.image.ImageDataGenerator`.

2 Importing Packages

Let's start by importing required packages. **os** package is used to read files and directory structure, **numpy** is used to convert python list to numpy array and to perform required matrix operations and **matplotlib.pyplot** is used to plot the graph and display images in our training and validation data.

[2]:
```
import os
import numpy as np
import glob
import shutil

import tensorflow as tf

import matplotlib.pyplot as plt
```

2.0.1 TODO: Import TensorFlow and Keras Layers

In the cell below, import Tensorflow as `tf` and the Keras layers and models you will use to build your CNN. Also, import the `ImageDataGenerator` from Keras so that you can perform image augmentation.

[3]:
```
#import packages
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

3 Data Loading

In order to build our image classifier, we can begin by downloading the flowers dataset. We first need to download the archive version of the dataset and after the download we are storing it to "/tmp/" directory.

After downloading the dataset, we need to extract its contents.

[4]:
```
_URL = "https://storage.googleapis.com/download.tensorflow.org/example_images/
↪flower_photos.tgz"

zip_file = tf.keras.utils.get_file(origin=_URL,
                                   fname="flower_photos.tgz",
                                   extract=True)

base_dir = os.path.join(os.path.dirname(zip_file), 'flower_photos')
```

```
Downloading data from https://storage.googleapis.com/download.tensorflow.org/exa
mple_images/flower_photos.tgz
228818944/228813984 [==============================] - 6s 0us/step
228827136/228813984 [==============================] - 6s 0us/step
```

The dataset we downloaded contains images of 5 types of flowers:

1. Rose
2. Daisy
3. Dandelion
4. Sunflowers
5. Tulips

So, let's create the labels for these 5 classes:

[5]:
```
classes = ['roses', 'daisy', 'dandelion', 'sunflowers', 'tulips']
```

Also, the dataset we have downloaded has following directory structure.

As you can see there are no folders containing training and validation data. Therefore, we will have to create our own training and validation set. Let's write some code that will do this.

The code below creates a `train` and a `val` folder each containing 5 folders (one for each type of flower). It then moves the images from the original folders to these new folders such that 80% of the images go to the training set and 20% of the images go into the validation set. In the end our directory will have the following structure:

Since we don't delete the original folders, they will still be in our `flower_photos` directory, but they will be empty. The code below also prints the total number of flower images we have for each type of flower.

[6]:
```
for cl in classes:
  img_path = os.path.join(base_dir, cl)
  images = glob.glob(img_path + '/*.jpg')
  print("{}: {} Images".format(cl, len(images)))
  train, val = images[:round(len(images)*0.8)], images[round(len(images)*0.8):]

  for t in train:
    if not os.path.exists(os.path.join(base_dir, 'train', cl)):
      os.makedirs(os.path.join(base_dir, 'train', cl))
    shutil.move(t, os.path.join(base_dir, 'train', cl))

  for v in val:
    if not os.path.exists(os.path.join(base_dir, 'val', cl)):
      os.makedirs(os.path.join(base_dir, 'val', cl))
    shutil.move(v, os.path.join(base_dir, 'val', cl))
```

```
roses: 641 Images
daisy: 633 Images
dandelion: 898 Images
sunflowers: 699 Images
tulips: 799 Images
```

[7]:
```
num_train = 0
num_val = 0
for cl in classes:
    num_train += len(os.listdir(os.path.join(base_dir, 'train', cl)))
    num_val += len(os.listdir(os.path.join(base_dir, 'val', cl)))

print("train    : {}".format(num_train))
print("val      : {}".format(num_val))
```

```
train    : 2935
val      : 735
```

For convenience, let us set up the path for the training and validation sets

[8]:
```
train_dir = os.path.join(base_dir, 'train')
val_dir = os.path.join(base_dir, 'val')
```

4 Data Augmentation

Overfitting generally occurs when we have small number of training examples. One way to fix this problem is to augment our dataset so that it has sufficient number of training examples. Data augmentation takes the approach of generating more training data from existing training samples, by augmenting the samples via a number of random transformations that yield believable-looking images. The goal is that at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data and generalize better.

In **tf.keras** we can implement this using the same **ImageDataGenerator** class we used before. We can simply pass different transformations we would want to our dataset as a form of arguments and it will take care of applying it to the dataset during our training process.

4.1 Experiment with Various Image Transformations

In this section you will get some practice doing some basic image transformations. Before we begin making transformations let's define our `batch_size` and our image size. Remember that the input to our CNN are images of the same size. We therefore have to resize the images in our dataset to the same size.

4.1.1 TODO: Set Batch and Image Size

In the cell below, create a `batch_size` of 100 images and set a value to `IMG_SHAPE` such that our training data consists of images with width of 150 pixels and height of 150 pixels.

[9]:
```
batch_size = 100
IMG_SHAPE = 150
```

4.1.2 TODO: Apply Random Horizontal Flip

In the cell below, use ImageDataGenerator to create a transformation that rescales the images by 255 and then applies a random horizontal flip. Then use the `.flow_from_directory` method to apply the above transformation to the images in our training set. Make sure you indicate the batch size, the path to the directory of the training images, the target size for the images, and to shuffle the images.

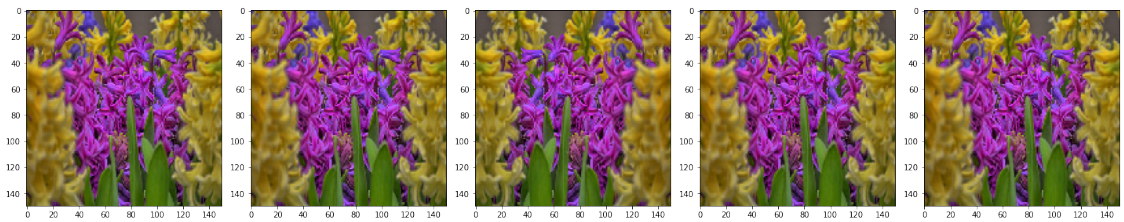
[11]:
```
image_gen = ImageDataGenerator(
    rescale=1./255,
    horizontal_flip=True
)

train_data_gen = image_gen.flow_from_directory(batch_size=batch_size,
                                               directory=train_dir,
                                               shuffle=True,
                                               target_size=(IMG_SHAPE,␣
 ↪IMG_SHAPE))
```

```
Found 2935 images belonging to 5 classes.
```

Let's take 1 sample image from our training examples and repeat it 5 times so that the augmentation can be applied to the same image 5 times over randomly, to see the augmentation in action.

[12]:
```
# This function will plot images in the form of a grid with 1 row and 5 columns␣
 ↪where images are placed in each column.
def plotImages(images_arr):
    fig, axes = plt.subplots(1, 5, figsize=(20,20))
```

```
    axes = axes.flatten()
    for img, ax in zip( images_arr, axes):
        ax.imshow(img)
    plt.tight_layout()
    plt.show()


augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



4.1.3 TODO: Apply Random Rotation

In the cell below, use ImageDataGenerator to create a transformation that rescales the images by 255 and then applies a random 45 degree rotation. Then use the `.flow_from_directory` method to apply the above transformation to the images in our training set. Make sure you indicate the batch size, the path to the directory of the training images, the target size for the images, and to shuffle the images.

[ ]:
```
image_gen = ImageDataGenerator(
            rescale=1./255,
            rotation_range=45
)

train_data_gen = image_gen.flow_from_directory(
                batch_size=batch_size,
                directory=train_dir,
                shuffle=True,
                target_size=(IMG_SHAPE, IMG_SHAPE)
)
```

Let's take 1 sample image from our training examples and repeat it 5 times so that the augmentation can be applied to the same image 5 times over randomly, to see the augmentation in action.

[ ]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```

4.1.4 TODO: Apply Random Zoom

In the cell below, use ImageDataGenerator to create a transformation that rescales the images by 255 and then applies a random zoom of up to 50%. Then use the `.flow_from_directory` method to apply the above transformation to the images in our training set. Make sure you indicate the batch size, the path to the directory of the training images, the target size for the images, and to shuffle the images.

[ ]:
```
image_gen = ImageDataGenerator(
            rescale=1./255,
            zoom_range=0.5
)

train_data_gen = image_gen.flow_from_directory(
                                                batch_size=batch_size,
                                                shuffle=True,
                                                directory=train_dir,
                                                target_size=(IMG_SHAPE, IMG_SHAPE)
)
```

Let's take 1 sample image from our training examples and repeat it 5 times so that the augmentation can be applied to the same image 5 times over randomly, to see the augmentation in action.

[ ]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```

4.1.5 TODO: Put It All Together

In the cell below, use ImageDataGenerator to create a transformation that rescales the images by 255 and that applies:

- random 45 degree rotation
- random zoom of up to 50%
- random horizontal flip
- width shift of 0.15
- height shift of 0.15

Then use the `.flow_from_directory` method to apply the above transformation to the images in our training set. Make sure you indicate the batch size, the path to the directory of the training images, the target size for the images, to shuffle the images, and to set the class mode to `sparse`.

[15]:
```
image_gen_train = ImageDataGenerator(
                rescale=1./255,
                rotation_range=45,
                zoom_range=0.5,
                horizontal_flip=True,
                width_shift_range=0.15,
                height_shift_range=0.15
)
```

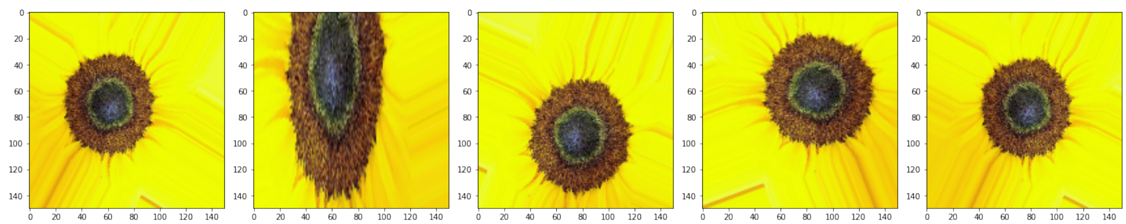
```
train_data_gen = image_gen_train.flow_from_directory(
                    batch_size=batch_size,
                    shuffle=True,
                    directory=train_dir,
                    target_size=(IMG_SHAPE, IMG_SHAPE),
                    class_mode='sparse'
)
```

```
Found 2935 images belonging to 5 classes.
```

Let's visualize how a single image would look like 5 different times, when we pass these augmentations randomly to our dataset.

[16]:

```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```



4.1.6 TODO: Create a Data Generator for the Validation Set

Generally, we only apply data augmentation to our training examples. So, in the cell below, use ImageDataGenerator to create a transformation that only rescales the images by 255. Then use the `.flow_from_directory` method to apply the above transformation to the images in our validation set. Make sure you indicate the batch size, the path to the directory of the validation images, the target size for the images, and to set the class mode to `sparse`. Remember that it is not necessary to shuffle the images in the validation set.

[17]:

```
image_gen_val = ImageDataGenerator(rescale=1./255)
val_data_gen = image_gen_val.flow_from_directory(
                    batch_size=batch_size,
                    directory=val_dir,
                    target_size=(IMG_SHAPE, IMG_SHAPE),
                    class_mode='sparse'
)
```

```
Found 735 images belonging to 5 classes.
```

5 TODO: Create the CNN

In the cell below, create a convolutional neural network that consists of 3 convolution blocks. Each convolutional block contains a `Conv2D` layer followed by a max pool layer. The first convolutional block should have 16 filters, the second one should have 32 filters, and the third one should have 64 filters. All convolutional filters should be 3 x 3. All max pool layers should have a `pool_size` of `(2, 2)`.

After the 3 convolutional blocks you should have a flatten layer followed by a fully connected layer with 512 units. The CNN should output class probabilities based on 5 classes which is done by the **softmax** activation function. All other layers should use a **relu** activation function. You should also add Dropout layers with a probability of 20%, where appropriate.

[31]:
```
model = tf.keras.Sequential([

        tf.keras.layers.Conv2D(16, (3, 3), activation='relu', input_shape=(150,␣
 ↪150, 3)),
        tf.keras.layers.MaxPooling2D(2, 2),

        tf.keras.layers.Conv2D(32, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2, 2),

        tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
        tf.keras.layers.MaxPooling2D(2, 2),

        tf.keras.layers.Flatten(),
        tf.keras.layers.Dropout(0.5),
        tf.keras.layers.Dense(512, activation='relu'),
        tf.keras.layers.Dense(5, activation=tf.nn.softmax)

])
```

6 TODO: Compile the Model

In the cell below, compile your model using the ADAM optimizer, the sparse cross entropy function as a loss function. We would also like to look at training and validation accuracy on each epoch as we train our network, so make sure you also pass the metrics argument.

[32]:
```
# Compile the model
model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

[33]:
```
model.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
conv2d_6 (Conv2D)            (None, 148, 148, 16)      448
_________________________________________________________________
max_pooling2d_6 (MaxPooling2 (None, 74, 74, 16)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 72, 72, 32)        4640
_________________________________________________________________
max_pooling2d_7 (MaxPooling2 (None, 36, 36, 32)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 34, 34, 64)        18496
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 17, 17, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 18496)             0
_________________________________________________________________
dropout (Dropout)            (None, 18496)             0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               9470464
_________________________________________________________________
dense_5 (Dense)              (None, 5)                 2565
=================================================================
Total params: 9,496,613
Trainable params: 9,496,613
Non-trainable params: 0
_________________________________________________________________
```

7 TODO: Train the Model

In the cell below, train your model using the **fit_generator** function instead of the usual **fit** function. We have to use the `fit_generator` function because we are using the **ImageDataGenerator** class to generate batches of training and validation data for our model. Train the model for 80 epochs and make sure you use the proper parameters in the `fit_generator` function.

[34]:

```
epochs = 80

history = model.fit(
    train_data_gen,
    steps_per_epoch=int(np.ceil(num_train / float(batch_size))),
    epochs=epochs,
    validation_data=val_data_gen,
    validation_steps=int(np.ceil(num_val / float(batch_size)))
)
```

```
Epoch 1/80
30/30 [==============================] - 28s 914ms/step - loss: 1.4963 -
accuracy: 0.3349 - val_loss: 1.2529 - val_accuracy: 0.4395
Epoch 2/80
30/30 [==============================] - 27s 897ms/step - loss: 1.1856 -
accuracy: 0.5012 - val_loss: 1.1323 - val_accuracy: 0.5469
```

```
Epoch 3/80
30/30 [==============================] - 27s 895ms/step - loss: 1.0766 -
accuracy: 0.5676 - val_loss: 1.0834 - val_accuracy: 0.5741
Epoch 4/80
30/30 [==============================] - 27s 900ms/step - loss: 1.0283 -
accuracy: 0.5905 - val_loss: 0.9712 - val_accuracy: 0.6190
Epoch 5/80
30/30 [==============================] - 27s 910ms/step - loss: 0.9579 -
accuracy: 0.6228 - val_loss: 0.9124 - val_accuracy: 0.6599
Epoch 6/80
30/30 [==============================] - 27s 893ms/step - loss: 0.9242 -
accuracy: 0.6412 - val_loss: 0.9356 - val_accuracy: 0.6395
Epoch 7/80
30/30 [==============================] - 27s 884ms/step - loss: 0.9109 -
accuracy: 0.6443 - val_loss: 0.8538 - val_accuracy: 0.6667
Epoch 8/80
30/30 [==============================] - 27s 884ms/step - loss: 0.8851 -
accuracy: 0.6504 - val_loss: 0.8395 - val_accuracy: 0.6789
Epoch 9/80
30/30 [==============================] - 26s 881ms/step - loss: 0.8582 -
accuracy: 0.6624 - val_loss: 0.8183 - val_accuracy: 0.6844
Epoch 10/80
30/30 [==============================] - 27s 884ms/step - loss: 0.8274 -
accuracy: 0.6845 - val_loss: 0.7915 - val_accuracy: 0.6816
Epoch 11/80
30/30 [==============================] - 27s 883ms/step - loss: 0.8161 -
accuracy: 0.6903 - val_loss: 0.9259 - val_accuracy: 0.6503
Epoch 12/80
30/30 [==============================] - 27s 884ms/step - loss: 0.8078 -
accuracy: 0.6917 - val_loss: 0.7403 - val_accuracy: 0.7075
Epoch 13/80
30/30 [==============================] - 27s 883ms/step - loss: 0.8003 -
accuracy: 0.6879 - val_loss: 0.7593 - val_accuracy: 0.7007
Epoch 14/80
30/30 [==============================] - 26s 883ms/step - loss: 0.7639 -
accuracy: 0.7043 - val_loss: 0.8355 - val_accuracy: 0.6789
Epoch 15/80
30/30 [==============================] - 27s 884ms/step - loss: 0.7762 -
accuracy: 0.7043 - val_loss: 0.7701 - val_accuracy: 0.7007
Epoch 16/80
30/30 [==============================] - 27s 885ms/step - loss: 0.7500 -
accuracy: 0.7131 - val_loss: 0.7256 - val_accuracy: 0.7333
Epoch 17/80
30/30 [==============================] - 27s 888ms/step - loss: 0.7266 -
accuracy: 0.7193 - val_loss: 0.8231 - val_accuracy: 0.6925
Epoch 18/80
30/30 [==============================] - 27s 895ms/step - loss: 0.7251 -
accuracy: 0.7216 - val_loss: 0.7342 - val_accuracy: 0.7184
```

```
Epoch 19/80
30/30 [==============================] - 27s 890ms/step - loss: 0.7157 -
accuracy: 0.7288 - val_loss: 0.7050 - val_accuracy: 0.7238
Epoch 20/80
30/30 [==============================] - 27s 889ms/step - loss: 0.6995 -
accuracy: 0.7336 - val_loss: 0.6951 - val_accuracy: 0.7211
Epoch 21/80
30/30 [==============================] - 27s 889ms/step - loss: 0.7219 -
accuracy: 0.7210 - val_loss: 0.8425 - val_accuracy: 0.6803
Epoch 22/80
30/30 [==============================] - 27s 893ms/step - loss: 0.7159 -
accuracy: 0.7247 - val_loss: 0.6809 - val_accuracy: 0.7388
Epoch 23/80
30/30 [==============================] - 27s 890ms/step - loss: 0.6812 -
accuracy: 0.7424 - val_loss: 0.6884 - val_accuracy: 0.7401
Epoch 24/80
30/30 [==============================] - 27s 887ms/step - loss: 0.6876 -
accuracy: 0.7339 - val_loss: 0.7133 - val_accuracy: 0.7238
Epoch 25/80
30/30 [==============================] - 27s 892ms/step - loss: 0.7138 -
accuracy: 0.7264 - val_loss: 0.7113 - val_accuracy: 0.7252
Epoch 26/80
30/30 [==============================] - 27s 886ms/step - loss: 0.6860 -
accuracy: 0.7264 - val_loss: 0.7218 - val_accuracy: 0.7279
Epoch 27/80
30/30 [==============================] - 27s 895ms/step - loss: 0.6395 -
accuracy: 0.7547 - val_loss: 0.8155 - val_accuracy: 0.7075
Epoch 28/80
30/30 [==============================] - 27s 893ms/step - loss: 0.6705 -
accuracy: 0.7428 - val_loss: 0.6940 - val_accuracy: 0.7211
Epoch 29/80
30/30 [==============================] - 27s 887ms/step - loss: 0.6562 -
accuracy: 0.7489 - val_loss: 0.6375 - val_accuracy: 0.7646
Epoch 30/80
30/30 [==============================] - 27s 892ms/step - loss: 0.6535 -
accuracy: 0.7468 - val_loss: 0.6676 - val_accuracy: 0.7361
Epoch 31/80
30/30 [==============================] - 27s 895ms/step - loss: 0.6854 -
accuracy: 0.7387 - val_loss: 0.6547 - val_accuracy: 0.7497
Epoch 32/80
30/30 [==============================] - 27s 891ms/step - loss: 0.6346 -
accuracy: 0.7578 - val_loss: 0.6487 - val_accuracy: 0.7456
Epoch 33/80
30/30 [==============================] - 27s 889ms/step - loss: 0.5956 -
accuracy: 0.7663 - val_loss: 0.6571 - val_accuracy: 0.7524
Epoch 34/80
30/30 [==============================] - 27s 887ms/step - loss: 0.6013 -
accuracy: 0.7564 - val_loss: 0.6746 - val_accuracy: 0.7469
```

```
Epoch 35/80
30/30 [==============================] - 26s 883ms/step - loss: 0.6008 -
accuracy: 0.7697 - val_loss: 0.7864 - val_accuracy: 0.7211
Epoch 36/80
30/30 [==============================] - 27s 886ms/step - loss: 0.5987 -
accuracy: 0.7697 - val_loss: 0.6179 - val_accuracy: 0.7374
Epoch 37/80
30/30 [==============================] - 27s 890ms/step - loss: 0.6125 -
accuracy: 0.7646 - val_loss: 0.7201 - val_accuracy: 0.7224
Epoch 38/80
30/30 [==============================] - 27s 890ms/step - loss: 0.5814 -
accuracy: 0.7741 - val_loss: 0.6141 - val_accuracy: 0.7565
Epoch 39/80
30/30 [==============================] - 27s 913ms/step - loss: 0.5807 -
accuracy: 0.7704 - val_loss: 0.6558 - val_accuracy: 0.7551
Epoch 40/80
30/30 [==============================] - 27s 891ms/step - loss: 0.5811 -
accuracy: 0.7789 - val_loss: 0.6462 - val_accuracy: 0.7510
Epoch 41/80
30/30 [==============================] - 27s 891ms/step - loss: 0.5951 -
accuracy: 0.7731 - val_loss: 0.6896 - val_accuracy: 0.7306
Epoch 42/80
30/30 [==============================] - 27s 892ms/step - loss: 0.5708 -
accuracy: 0.7748 - val_loss: 0.6605 - val_accuracy: 0.7361
Epoch 43/80
30/30 [==============================] - 27s 915ms/step - loss: 0.5766 -
accuracy: 0.7796 - val_loss: 0.6507 - val_accuracy: 0.7469
Epoch 44/80
30/30 [==============================] - 27s 892ms/step - loss: 0.5634 -
accuracy: 0.7843 - val_loss: 0.6554 - val_accuracy: 0.7429
Epoch 45/80
30/30 [==============================] - 27s 888ms/step - loss: 0.5567 -
accuracy: 0.7853 - val_loss: 0.7618 - val_accuracy: 0.7279
Epoch 46/80
30/30 [==============================] - 27s 892ms/step - loss: 0.5544 -
accuracy: 0.7956 - val_loss: 0.6454 - val_accuracy: 0.7442
Epoch 47/80
30/30 [==============================] - 27s 887ms/step - loss: 0.5615 -
accuracy: 0.7847 - val_loss: 0.6682 - val_accuracy: 0.7551
Epoch 48/80
30/30 [==============================] - 27s 886ms/step - loss: 0.5561 -
accuracy: 0.7850 - val_loss: 0.6684 - val_accuracy: 0.7401
Epoch 49/80
30/30 [==============================] - 27s 896ms/step - loss: 0.5319 -
accuracy: 0.7928 - val_loss: 0.6360 - val_accuracy: 0.7565
Epoch 50/80
30/30 [==============================] - 27s 892ms/step - loss: 0.5275 -
accuracy: 0.8007 - val_loss: 0.6785 - val_accuracy: 0.7524
```

```
Epoch 51/80
30/30 [==============================] - 27s 896ms/step - loss: 0.5438 -
accuracy: 0.7871 - val_loss: 0.6935 - val_accuracy: 0.7333
Epoch 52/80
30/30 [==============================] - 27s 888ms/step - loss: 0.5299 -
accuracy: 0.7983 - val_loss: 0.6784 - val_accuracy: 0.7551
Epoch 53/80
30/30 [==============================] - 27s 891ms/step - loss: 0.5267 -
accuracy: 0.7980 - val_loss: 0.6110 - val_accuracy: 0.7442
Epoch 54/80
30/30 [==============================] - 27s 887ms/step - loss: 0.4988 -
accuracy: 0.8085 - val_loss: 0.7003 - val_accuracy: 0.7415
Epoch 55/80
30/30 [==============================] - 27s 906ms/step - loss: 0.5260 -
accuracy: 0.8000 - val_loss: 0.6551 - val_accuracy: 0.7605
Epoch 56/80
30/30 [==============================] - 27s 889ms/step - loss: 0.5184 -
accuracy: 0.8044 - val_loss: 0.6588 - val_accuracy: 0.7633
Epoch 57/80
30/30 [==============================] - 27s 887ms/step - loss: 0.5037 -
accuracy: 0.8112 - val_loss: 0.6262 - val_accuracy: 0.7524
Epoch 58/80
30/30 [==============================] - 27s 889ms/step - loss: 0.5230 -
accuracy: 0.8000 - val_loss: 0.6817 - val_accuracy: 0.7347
Epoch 59/80
30/30 [==============================] - 26s 882ms/step - loss: 0.4953 -
accuracy: 0.8078 - val_loss: 0.6497 - val_accuracy: 0.7361
Epoch 60/80
30/30 [==============================] - 27s 892ms/step - loss: 0.5085 -
accuracy: 0.8075 - val_loss: 0.6402 - val_accuracy: 0.7483
Epoch 61/80
30/30 [==============================] - 27s 889ms/step - loss: 0.4816 -
accuracy: 0.8174 - val_loss: 0.7000 - val_accuracy: 0.7483
Epoch 62/80
30/30 [==============================] - 26s 883ms/step - loss: 0.4715 -
accuracy: 0.8174 - val_loss: 0.6951 - val_accuracy: 0.7633
Epoch 63/80
30/30 [==============================] - 27s 886ms/step - loss: 0.4697 -
accuracy: 0.8215 - val_loss: 0.6185 - val_accuracy: 0.7537
Epoch 64/80
30/30 [==============================] - 27s 883ms/step - loss: 0.4526 -
accuracy: 0.8232 - val_loss: 0.6261 - val_accuracy: 0.7619
Epoch 65/80
30/30 [==============================] - 27s 886ms/step - loss: 0.4566 -
accuracy: 0.8286 - val_loss: 0.5879 - val_accuracy: 0.7701
Epoch 66/80
30/30 [==============================] - 26s 881ms/step - loss: 0.4876 -
accuracy: 0.8099 - val_loss: 0.5940 - val_accuracy: 0.7619
```

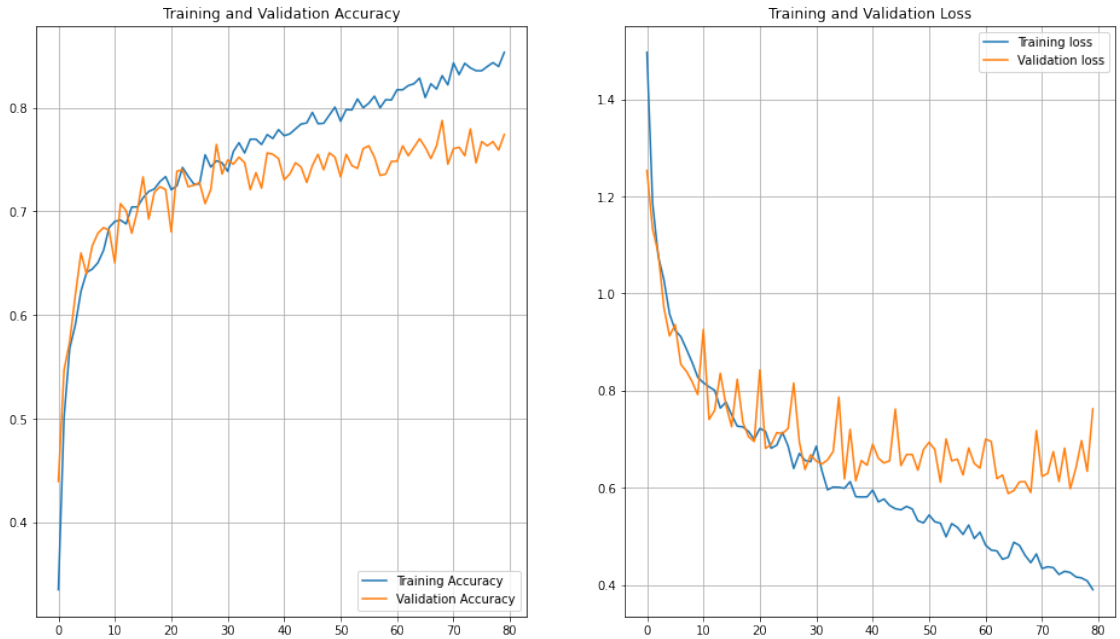
```
Epoch 67/80
30/30 [==============================] - 26s 883ms/step - loss: 0.4811 -
accuracy: 0.8232 - val_loss: 0.6119 - val_accuracy: 0.7510
Epoch 68/80
30/30 [==============================] - 27s 883ms/step - loss: 0.4607 -
accuracy: 0.8181 - val_loss: 0.6122 - val_accuracy: 0.7633
Epoch 69/80
30/30 [==============================] - 26s 874ms/step - loss: 0.4455 -
accuracy: 0.8310 - val_loss: 0.5900 - val_accuracy: 0.7878
Epoch 70/80
30/30 [==============================] - 26s 880ms/step - loss: 0.4635 -
accuracy: 0.8221 - val_loss: 0.7175 - val_accuracy: 0.7456
Epoch 71/80
30/30 [==============================] - 26s 883ms/step - loss: 0.4335 -
accuracy: 0.8433 - val_loss: 0.6234 - val_accuracy: 0.7605
Epoch 72/80
30/30 [==============================] - 26s 898ms/step - loss: 0.4371 -
accuracy: 0.8320 - val_loss: 0.6293 - val_accuracy: 0.7619
Epoch 73/80
30/30 [==============================] - 26s 882ms/step - loss: 0.4352 -
accuracy: 0.8429 - val_loss: 0.6743 - val_accuracy: 0.7537
Epoch 74/80
30/30 [==============================] - 26s 898ms/step - loss: 0.4213 -
accuracy: 0.8385 - val_loss: 0.6127 - val_accuracy: 0.7796
Epoch 75/80
30/30 [==============================] - 26s 876ms/step - loss: 0.4278 -
accuracy: 0.8358 - val_loss: 0.6814 - val_accuracy: 0.7469
Epoch 76/80
30/30 [==============================] - 26s 882ms/step - loss: 0.4253 -
accuracy: 0.8358 - val_loss: 0.5978 - val_accuracy: 0.7673
Epoch 77/80
30/30 [==============================] - 26s 881ms/step - loss: 0.4162 -
accuracy: 0.8399 - val_loss: 0.6405 - val_accuracy: 0.7633
Epoch 78/80
30/30 [==============================] - 26s 874ms/step - loss: 0.4142 -
accuracy: 0.8436 - val_loss: 0.6969 - val_accuracy: 0.7673
Epoch 79/80
30/30 [==============================] - 26s 880ms/step - loss: 0.4080 -
accuracy: 0.8399 - val_loss: 0.6336 - val_accuracy: 0.7592
Epoch 80/80
30/30 [==============================] - 26s 877ms/step - loss: 0.3900 -
accuracy: 0.8535 - val_loss: 0.7622 - val_accuracy: 0.7741
```

8 TODO: Plot Training and Validation Graphs.

In the cell below, plot the training and validation accuracy/loss graphs.

[35]:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(epochs)

plt.figure(figsize=(16, 9))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
plt.grid()
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label="Training loss")
plt.plot(epochs_range, val_loss, label="Validation loss")
plt.legend(loc='upper right')
plt.grid()
plt.title('Training and Validation Loss')

plt.show()
```



9 TODO: Experiment with Different Parameters

So far you've created a CNN with 3 convolutional layers and followed by a fully connected layer with 512 units. In the cells below create a new CNN with a different architecture. Feel free to experiment by changing as many parameters as you like. For example, you can add more convolutional layers, or more fully connected layers. You can also experiment with different filter sizes in your convolutional layers, different number of units in your fully connected layers, different dropout rates, etc… You can also experiment by performing image augmentation with more image transformations that we have seen so far. Take a look at the ImageDataGenerator Documentation to see a full list of all the available image transformations. For example, you can add shear transformations, or you can vary the brightness of the images, etc… Experiment as much as you can and compare the accuracy of your various models. Which parameters give you the best result?