TF_exercise_flowers_multiGPU_TGu

September 11, 2024

[1]:
```
#@title Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# https://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

Print your name

[2]:
```
## Your code here
print("Exercise by: Teemu Gustafsson.")
```

```
Exercise by: Teemu Gustafsson.
```

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.

[3]:
```python
import os
import numpy as np
import glob
import shutil

import matplotlib.pyplot as plt
```

2.0.1 TODO: Import TensorFlow and Keras Layers

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

```python
# Import os, plt, tf, keras and ImageDataGenerator
```

[4]:
```python
## Task 1:
## Your code here
import os
from matplotlib import pyplot as plt
import tensorflow as tf
import keras
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

[5]:
```python
print('Tensorflow version:', tf.__version__)
print('Keras version:', keras.__version__)
```

```
Tensorflow version: 2.17.0
Keras version: 3.5.0
```

[6]:
```python
#Tässä jaetaan GPU neljään osaan:

gpus = tf.config.list_physical_devices('GPU')
gpus
```

[6]:
```
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

[7]:
```python
tf.config.set_logical_device_configuration(
    gpus[0],
    [
        tf.config.LogicalDeviceConfiguration(memory_limit=3072),
        tf.config.LogicalDeviceConfiguration(memory_limit=3072),
        tf.config.LogicalDeviceConfiguration(memory_limit=3072),
        tf.config.LogicalDeviceConfiguration(memory_limit=3072),
    ])
```

[8]:
```python
logical_gpus = tf.config.list_logical_devices('GPU')
print(f'{len(gpus)} Physical GPU(s), {len(logical_gpus)} Logical GPU(s)')
```

```
1 Physical GPU(s), 4 Logical GPU(s)
```

```
I0000 00:00:1726073549.191274      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.191860      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.192304      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.547113      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.547736      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.548203      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.548657      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.549110      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.549555      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.550938      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
```

```
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.551533      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
I0000 00:00:1726073549.552449      91 cuda_executor.cc:1015] successful NUMA
node read from SysFS had negative value (-1), but there must be at least one
NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
```

[9]:
```
logical_gpus
```

[9]:
```
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),
 LogicalDevice(name='/device:GPU:1', device_type='GPU'),
 LogicalDevice(name='/device:GPU:2', device_type='GPU'),
 LogicalDevice(name='/device:GPU:3', device_type='GPU')]
```

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.

[10]:
```
# NOTE: Do not use if you run this in jupyterhub.dclabra.fi Use shared data␣
↪instead.

# _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')
# print(base_dir)
```

[11]:
```
# NOTE: Use shared data in jupyterhub.dclabra.fi
base_dir = "/home/jovyan/shared/flower_photos"
```

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:

[12]:
```
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.

[13]:
```
# NOTE: Do not use if you run this in jupyterhub.dclabra.fi Use shared data␣
↪instead.

# base_dir = "/home/jovyan/.keras/datasets/flower_photos"
#
# train_total = 0
# val_total = 0
#
# 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):
↪]
#   train_total = train_total + len(train)
#   val_total = val_total + len(val)
#
#   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))
#
# print("Train:", train_total)
```

```
# print("Val:", val_total)
```

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

[14]:
```
train_dir = os.path.join(base_dir, 'train')
val_dir = os.path.join(base_dir, 'val')
```

3.0.1 TODO: Print how many training and validation images we have in each category.

[15]:
```
## Task 2:
## Your code here

print("Training images per category ")
print()
print("Daisies ", len(os.listdir('/home/jovyan/shared/flower_photos/test/
↪daisy')))
print("Dandelions ", len(os.listdir('/home/jovyan/shared/flower_photos/test/
↪dandelion')))
print("Roses ", len(os.listdir('/home/jovyan/shared/flower_photos/test/roses')))
print("Sunflowers ", len(os.listdir('/home/jovyan/shared/flower_photos/test/
↪sunflowers')))
print("Tulips ", len(os.listdir('/home/jovyan/shared/flower_photos/test/
↪tulips')))
print()
print("Validation images per category")
print()
print("Daisies ", len(os.listdir('/home/jovyan/shared/flower_photos/val/
↪daisy')))
print("Dandelions ", len(os.listdir('/home/jovyan/shared/flower_photos/val/
↪dandelion')))
print("Roses ", len(os.listdir('/home/jovyan/shared/flower_photos/val/roses')))
print("Sunflowers ", len(os.listdir('/home/jovyan/shared/flower_photos/val/
↪sunflowers')))
print("Tulips ", len(os.listdir('/home/jovyan/shared/flower_photos/val/
↪tulips')))
```

```
Training images per category

Daisies  95
Dandelions  135
Roses  97
Sunflowers  105
Tulips  120

Validation images per category
```

```
Daisies  96
Dandelions  136
Roses  97
Sunflowers  106
Tulips  121
```

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.

```
# Set variables
batch_size =
IMG_SHAPE =
```

[41]:
```
## Task 3:
## Your code here
GLOBAL_BATCH_SIZE = 100 * len(tf.config.list_logical_devices('GPU'))
IMG_SHAPE = 150

print(GLOBAL_BATCH_SIZE)
```

```
400
```

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.

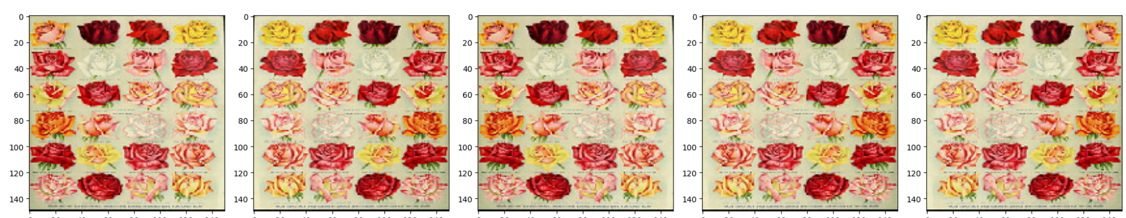
```
# Set training data generator
image_gen =
train_data_gen =
```

[13]:
```
## Task 4:
## Your code here
```

```
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.

[14]:
```
# 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.

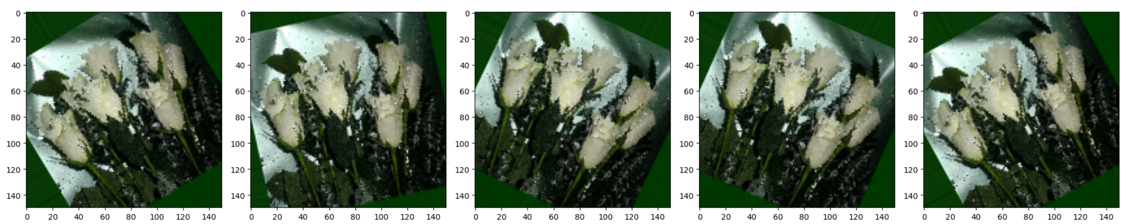
```
# Set training data generator
image_gen =
train_data_gen =
```

[15]:
```
## Task 5:
## Your code here
```

```
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.

[16]:
```
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.

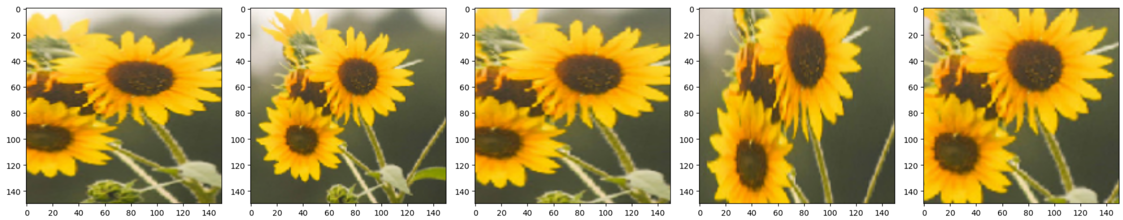
```
# Set training data generator
image_gen =
train_data_gen =
```

[17]:
```
## Task 6:
## Your code here
```

```
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.

[18]:
```
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`.

```
# Set training data generator
image_gen =
train_data_gen =
```

[42]:
```
## Task 7:
## Your code here




train_data_gen = tf.keras.utils.image_dataset_from_directory(
  directory = train_dir,
  image_size=(IMG_SHAPE, IMG_SHAPE),
  batch_size=GLOBAL_BATCH_SIZE,
    shuffle= True
    )
```

```
Found 2562 files 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.

[30]:
```
augmented_images = [train_data_gen[0][0][0] for i in range(5)]
plotImages(augmented_images)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[30], line 1
----> 1 augmented_images = [train_data_gen[0][0][0] for i in range(5)]
      2 plotImages(augmented_images)

Cell In[30], line 1, in <listcomp>(.0)
----> 1 augmented_images = [train_data_gen[0][0][0] for i in range(5)]
      2 plotImages(augmented_images)

TypeError: '_PrefetchDataset' object is not subscriptable
```

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.

```
# Set validation data generator
image_gen_val =
val_data_gen =
```

[43]:
```
## Task 8:
## Your code here

val_data_gen = keras.utils.image_dataset_from_directory(
    directory=val_dir,
    batch_size=GLOBAL_BATCH_SIZE,
    image_size=(IMG_SHAPE, IMG_SHAPE),
    shuffle=False
)
```

```
Found 556 files 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.

```
# Create neural network
model =
```

[44]:
```
## Task 9:
## Your code here
from tensorflow.keras import layers

strategy = tf.distribute.MirroredStrategy()

data_augmentaatio = tf.keras.Sequential([
    layers.Rescaling(1./255),
    layers.RandomFlip("horizontal"),
    layers.RandomRotation(0.125),  # 0.125 * 360 = 45 astetta
    layers.RandomWidth(0.15),
    layers.RandomHeight(0.15),
    layers.RandomZoom(0.5)
])
```

```
INFO:tensorflow:Using MirroredStrategy with devices
('/job:localhost/replica:0/task:0/device:GPU:0',
'/job:localhost/replica:0/task:0/device:GPU:1',
'/job:localhost/replica:0/task:0/device:GPU:2',
'/job:localhost/replica:0/task:0/device:GPU:3')
```

[ ]:
```

```

[45]:
```
with strategy.scope():
    model = keras.models.Sequential([
        keras.Input(shape=[150, 150, 3]), # Kuvat laitetaan tässä koossa?

        data_augmentaatio, # Augmentoidaan

        layers.Resizing(150, 150), #Korjataan kuvan skaalaus, koska␣
 ↪augmentoinnin randomwidht ja randomheight rikkovat skaalausta

        layers.Conv2D(16, (3, 3), activation='relu'), #Eka kerros
        layers.MaxPooling2D(pool_size=(2, 2)),

        layers.Conv2D(32, (3, 3), activation='relu'), # Toka Kerros
        layers.MaxPooling2D(pool_size=(2, 2)),

        layers.Conv2D(64, (3, 3), activation='relu'),# Kolmas kerros
        layers.MaxPooling2D(pool_size=(2, 2)),

        layers.Flatten(),
```

```
        layers.Dense(512, activation='relu'),
        layers.Dropout(0.2),
        layers.Activation('relu'),
        layers.Dense(5, activation='softmax')
    ])

    model.compile(optimizer='adam',
                  loss=keras.losses.SparseCategoricalCrossentropy(from_logits=True),
                  metrics=['accuracy'])
```

[46]:
```
model.summary()
```

Model: "sequential_6"

Layer (type)	Output Shape	Param #
sequential_5 (Sequential)	(None, None, None, 3)	0
resizing_3 (Resizing)	(None, 150, 150, 3)	0
conv2d_9 (Conv2D)	(None, 148, 148, 16)	448
max_pooling2d_9 (MaxPooling2D)	(None, 74, 74, 16)	0
conv2d_10 (Conv2D)	(None, 72, 72, 32)	4,640
max_pooling2d_10 (MaxPooling2D)	(None, 36, 36, 32)	0
conv2d_11 (Conv2D)	(None, 34, 34, 64)	18,496
max_pooling2d_11 (MaxPooling2D)	(None, 17, 17, 64)	0
flatten_3 (Flatten)	(None, 18496)	0
dense_6 (Dense)	(None, 512)	9,470,464
dropout_3 (Dropout)	(None, 512)	0
activation_1 (Activation)	(None, 512)	0
dense_7 (Dense)	(None, 5)	2,565

Total params: 9,496,613 (36.23 MB)

```
Trainable params: 9,496,613 (36.23 MB)

Non-trainable params: 0 (0.00 B)
```

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.

```
# Compile the model
```

[47]:
```
## Task 10:
## Your code here
```

7 TODO: Train the Model

In the cell below, train your model using the **fit** function. Train the model for 80 epochs and make sure you use the proper parameters in the `fit` function.

```
# Train the model
epochs =
history =
```

[50]:
```
## Task 11:
## Your code here
epochs=10
with strategy.scope():
    history = model.fit(
        train_data_gen,
        epochs=epochs,
        validation_data=val_data_gen
    )
```

```
Epoch 1/10
7/7              7s 887ms/step -
accuracy: 0.6350 - loss: 0.9314 - val_accuracy: 0.6763 - val_loss: 0.9072
Epoch 2/10
7/7              6s 850ms/step -
accuracy: 0.6725 - loss: 0.8767 - val_accuracy: 0.7338 - val_loss: 0.8050
Epoch 3/10
7/7              6s 813ms/step -
accuracy: 0.6778 - loss: 0.8349 - val_accuracy: 0.7626 - val_loss: 0.7432
Epoch 4/10
7/7              6s 812ms/step -
accuracy: 0.6797 - loss: 0.8367 - val_accuracy: 0.6619 - val_loss: 0.9163
Epoch 5/10
```

```
7/7              7s 931ms/step -
accuracy: 0.6702 - loss: 0.8574 - val_accuracy: 0.5827 - val_loss: 1.2955
Epoch 6/10
7/7              6s 825ms/step -
accuracy: 0.6634 - loss: 0.8692 - val_accuracy: 0.7842 - val_loss: 0.6774
Epoch 7/10
7/7              6s 839ms/step -
accuracy: 0.6907 - loss: 0.8004 - val_accuracy: 0.7986 - val_loss: 0.5320
Epoch 8/10
7/7              6s 799ms/step -
accuracy: 0.6667 - loss: 0.8169 - val_accuracy: 0.7842 - val_loss: 0.6709
Epoch 9/10
7/7              6s 843ms/step -
accuracy: 0.6781 - loss: 0.8240 - val_accuracy: 0.8129 - val_loss: 0.5627
Epoch 10/10
7/7              6s 829ms/step -
accuracy: 0.7076 - loss: 0.7719 - val_accuracy: 0.8058 - val_loss: 0.6164
```

8 TODO: Plot Training and Validation Graphs.

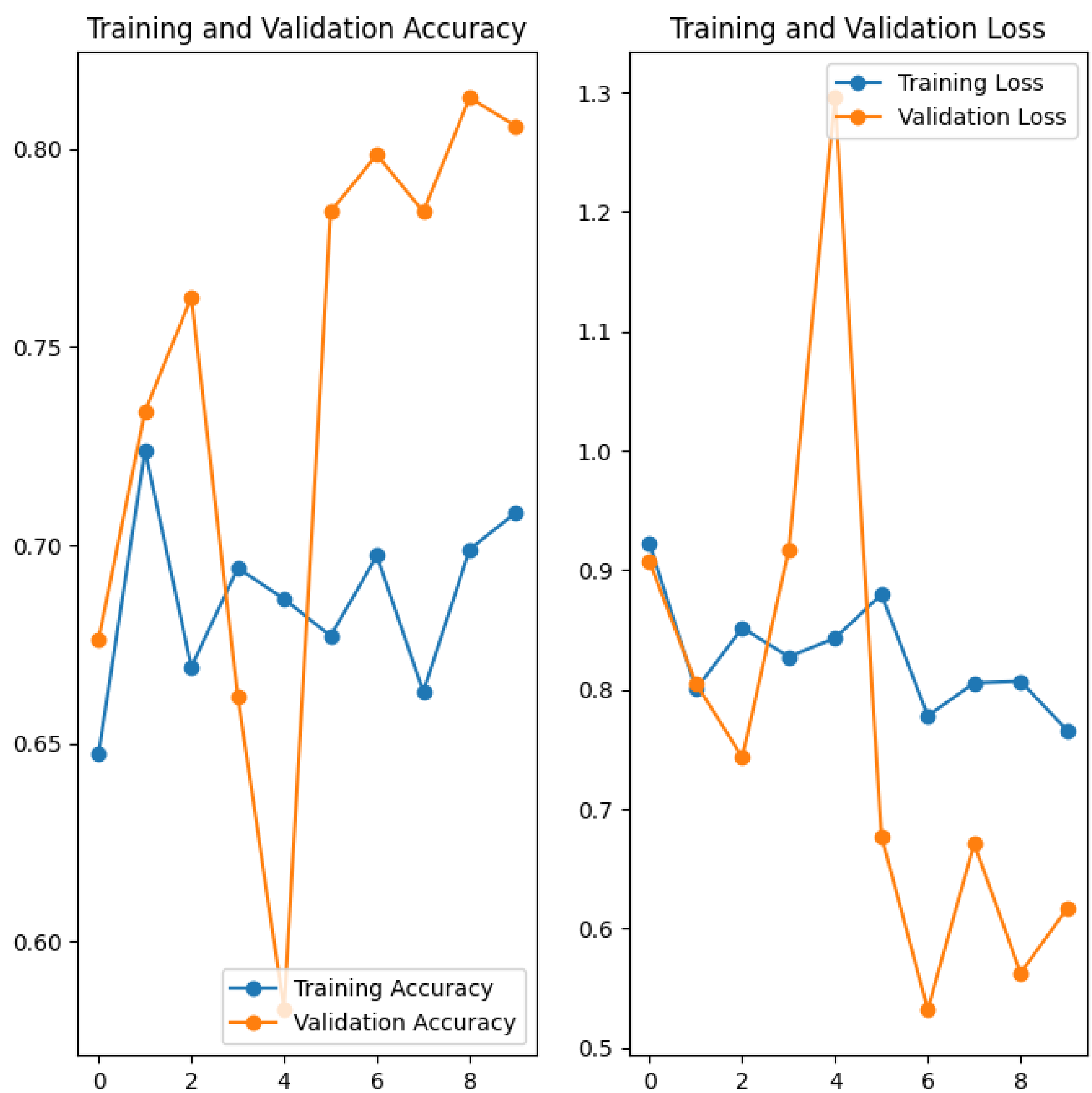
In the cell below, plot the training and validation accuracy/loss graphs.

```
# Plot Training and Validation Graphs
acc =
val_acc =

loss =
val_loss =

epochs_range =
```

[52]:

```
## Task 12:
## Your code here

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=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy', marker='o')
plt.plot(epochs_range, val_acc, label='Validation Accuracy', marker='o')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
```

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



9 OPTIONAL 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?

[ ]:

```

```