TF_exercise_flowers_with_data_augmentation_exercise_TGu

September 8, 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]:
```
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.

# Import os, plt, tf, keras and ImageDataGenerator

[4]:
```
## 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]:
```
print('Tensorflow version:', tf.__version__)
print('Keras version:', keras.__version__)
```

```
Tensorflow version: 2.16.1
Keras version: 3.3.3
```

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.

[6]:
```
# 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)
```

[7]:
```
# 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:

[8]:
```
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.

[9]:
```
# 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

[10]:
```
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.

[11]:
```
## 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 =
```

[12]:
```
## Task 3:
## Your code here
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.

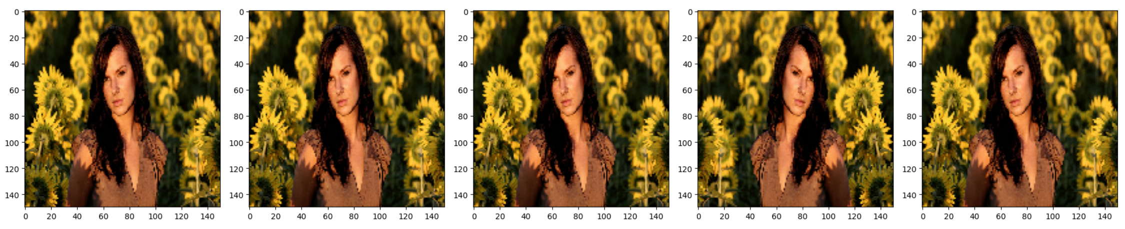
```
# Set training data generator
image_gen =
train_data_gen =
```

[13]:
```
## Task 4:
## Your code here
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 2562 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.

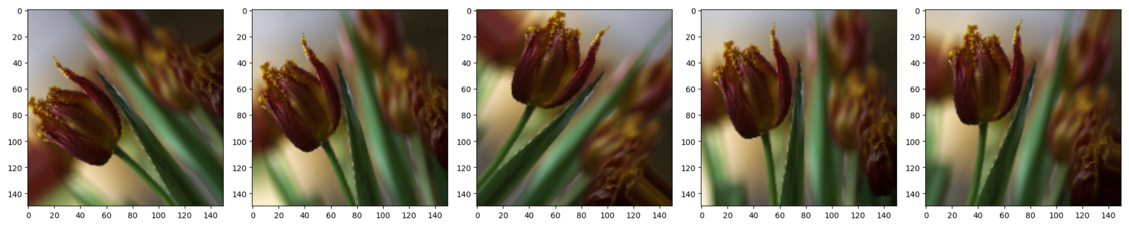
```
# Set training data generator
image_gen =
train_data_gen =
```

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

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))
```

```
Found 2562 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.

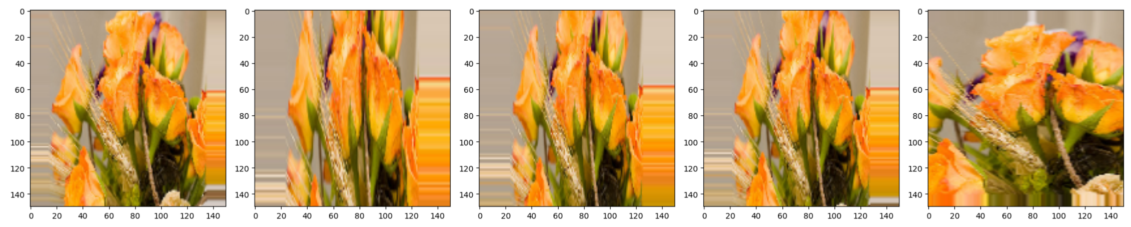
```
# Set training data generator
image_gen =
train_data_gen =
```

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

image_gen = ImageDataGenerator(rescale=1./255, zoom_range=0.5)

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

```
Found 2562 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`.

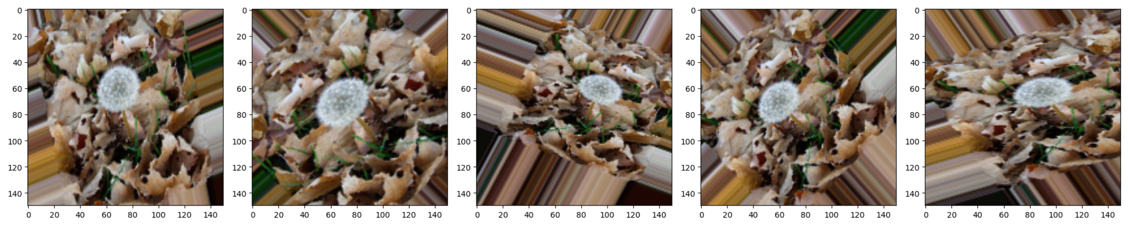
```
# Set training data generator
image_gen =
train_data_gen =
```

[19]:
```
## Task 7:
## Your code here

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

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

```
Found 2562 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.

[20]:
```
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.

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

[21]:
```
## Task 8:
## Your code here

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 556 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.

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

[44]:
```
model = keras.models.Sequential([
    keras.Input(shape=[150, 150, 3]),

    # First block
    keras.layers.Conv2D(16, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),

    # Second block
    keras.layers.Conv2D(32, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),

    # Third block
    keras.layers.Conv2D(64, (3, 3), activation='relu'),
    keras.layers.MaxPooling2D(pool_size=(2, 2)),

    # Flattening and fully connected layer
    keras.layers.Flatten(),
    keras.layers.Dense(512, activation='relu'),
    keras.layers.Dropout(0.2),

    # Output layer for 5 classes
    keras.layers.Dense(5, activation='softmax')
])
```

[45]:
```
model.summary()
```

```
Model: "sequential_4"


 Layer (type)                      Output Shape                  Param #

 conv2d_12 (Conv2D)                (None, 148, 148, 16)              448

 max_pooling2d_12 (MaxPooling2D)   (None, 74, 74, 16)                  0

 conv2d_13 (Conv2D)                (None, 72, 72, 32)              4,640

 max_pooling2d_13 (MaxPooling2D)   (None, 36, 36, 32)                  0

 conv2d_14 (Conv2D)                (None, 34, 34, 64)             18,496

 max_pooling2d_14 (MaxPooling2D)   (None, 17, 17, 64)                  0

 flatten_4 (Flatten)               (None, 18496)                       0

 dense_7 (Dense)                   (None, 512)                 9,470,464

 dropout_4 (Dropout)               (None, 512)                         0

 dense_8 (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
```

[46]:
```
## Task 10:
## Your code here

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

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 =
```

[47]:
```
print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))
```

```
Num GPUs Available:  1
```

[48]:
```
## Task 11:
## Your code here
from tensorflow import keras

# Set the number of epochs
epochs = 80

with tf.device('GPU:0'):
    history = model.fit(
        train_data_gen,
        epochs=epochs,
        validation_data=val_data_gen   )
```

```
Epoch 1/80
```

```
/opt/conda/lib/python3.11/site-packages/keras/src/backend/tensorflow/nn.py:602:
UserWarning: "`sparse_categorical_crossentropy` received `from_logits=True`, but
the `output` argument was produced by a Softmax activation and thus does not
represent logits. Was this intended?
  output, from_logits = _get_logits(
/opt/conda/lib/python3.11/site-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1725029468.344382     293 service.cc:145] XLA service 0x7fc2b40071b0
initialized for platform CUDA (this does not guarantee that XLA will be used).
Devices:
I0000 00:00:1725029468.344455     293 service.cc:153]   StreamExecutor device
(0): Tesla V100-PCIE-12GB, Compute Capability 7.0

 1/26            4:46 11s/step - accuracy:
0.1300 - loss: 1.6313

I0000 00:00:1725029471.814907     293 device_compiler.h:188] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.

26/26            31s 762ms/step -
accuracy: 0.2348 - loss: 1.9465 - val_accuracy: 0.3561 - val_loss: 1.4047
Epoch 2/80
26/26            18s 505ms/step -
accuracy: 0.4262 - loss: 1.2707 - val_accuracy: 0.5090 - val_loss: 1.1690
Epoch 3/80
26/26            17s 467ms/step -
accuracy: 0.5206 - loss: 1.1437 - val_accuracy: 0.5647 - val_loss: 1.0540
Epoch 4/80
26/26            17s 468ms/step -
accuracy: 0.5769 - loss: 1.0574 - val_accuracy: 0.6223 - val_loss: 1.0166
Epoch 5/80
26/26            17s 478ms/step -
accuracy: 0.5867 - loss: 1.0332 - val_accuracy: 0.6601 - val_loss: 0.9401
Epoch 6/80
26/26            17s 481ms/step -
accuracy: 0.6426 - loss: 0.9451 - val_accuracy: 0.6277 - val_loss: 0.9226
Epoch 7/80
26/26            17s 471ms/step -
accuracy: 0.6376 - loss: 0.9343 - val_accuracy: 0.6565 - val_loss: 0.8658
Epoch 8/80
26/26            16s 466ms/step -
accuracy: 0.6279 - loss: 0.9294 - val_accuracy: 0.6385 - val_loss: 0.9102
Epoch 9/80
26/26            16s 461ms/step -
accuracy: 0.6523 - loss: 0.8946 - val_accuracy: 0.6727 - val_loss: 0.8521
Epoch 10/80
26/26            17s 472ms/step -
accuracy: 0.6395 - loss: 0.9172 - val_accuracy: 0.6511 - val_loss: 0.9055
```

```
Epoch 11/80
26/26            17s 480ms/step -
accuracy: 0.6866 - loss: 0.8469 - val_accuracy: 0.6817 - val_loss: 0.8707
Epoch 12/80
26/26            17s 479ms/step -
accuracy: 0.6707 - loss: 0.8376 - val_accuracy: 0.6205 - val_loss: 0.9902
Epoch 13/80
26/26            17s 473ms/step -
accuracy: 0.6650 - loss: 0.8641 - val_accuracy: 0.6565 - val_loss: 0.9114
Epoch 14/80
26/26            17s 476ms/step -
accuracy: 0.6780 - loss: 0.8058 - val_accuracy: 0.6781 - val_loss: 0.8314
Epoch 15/80
26/26            17s 481ms/step -
accuracy: 0.7059 - loss: 0.7684 - val_accuracy: 0.6924 - val_loss: 0.8089
Epoch 16/80
26/26            17s 469ms/step -
accuracy: 0.7181 - loss: 0.7694 - val_accuracy: 0.6960 - val_loss: 0.7572
Epoch 17/80
26/26            17s 472ms/step -
accuracy: 0.7216 - loss: 0.7497 - val_accuracy: 0.6906 - val_loss: 0.7420
Epoch 18/80
26/26            16s 468ms/step -
accuracy: 0.6791 - loss: 0.7926 - val_accuracy: 0.7194 - val_loss: 0.7040
Epoch 19/80
26/26            17s 482ms/step -
accuracy: 0.7165 - loss: 0.7463 - val_accuracy: 0.7176 - val_loss: 0.7084
Epoch 20/80
26/26            17s 472ms/step -
accuracy: 0.7274 - loss: 0.6992 - val_accuracy: 0.7176 - val_loss: 0.7361
Epoch 21/80
26/26            17s 470ms/step -
accuracy: 0.7190 - loss: 0.7058 - val_accuracy: 0.6978 - val_loss: 0.7919
Epoch 22/80
26/26            17s 473ms/step -
accuracy: 0.7285 - loss: 0.7081 - val_accuracy: 0.6996 - val_loss: 0.8396
Epoch 23/80
26/26            17s 468ms/step -
accuracy: 0.7262 - loss: 0.7218 - val_accuracy: 0.7050 - val_loss: 0.7525
Epoch 24/80
26/26            17s 472ms/step -
accuracy: 0.7501 - loss: 0.6680 - val_accuracy: 0.7410 - val_loss: 0.6470
Epoch 25/80
26/26            17s 485ms/step -
accuracy: 0.7482 - loss: 0.6432 - val_accuracy: 0.7302 - val_loss: 0.7175
Epoch 26/80
26/26            17s 471ms/step -
accuracy: 0.7527 - loss: 0.6486 - val_accuracy: 0.7248 - val_loss: 0.7295
```

```
Epoch 27/80
26/26            17s 474ms/step -
accuracy: 0.7407 - loss: 0.6598 - val_accuracy: 0.7086 - val_loss: 0.8112
Epoch 28/80
26/26            17s 486ms/step -
accuracy: 0.7251 - loss: 0.6760 - val_accuracy: 0.7176 - val_loss: 0.7319
Epoch 29/80
26/26            17s 487ms/step -
accuracy: 0.7402 - loss: 0.6820 - val_accuracy: 0.7464 - val_loss: 0.6811
Epoch 30/80
26/26            17s 483ms/step -
accuracy: 0.7518 - loss: 0.6374 - val_accuracy: 0.7410 - val_loss: 0.6636
Epoch 31/80
26/26            17s 479ms/step -
accuracy: 0.7554 - loss: 0.6324 - val_accuracy: 0.7392 - val_loss: 0.6849
Epoch 32/80
26/26            17s 474ms/step -
accuracy: 0.7508 - loss: 0.6152 - val_accuracy: 0.7248 - val_loss: 0.6963
Epoch 33/80
26/26            17s 501ms/step -
accuracy: 0.7625 - loss: 0.6166 - val_accuracy: 0.7482 - val_loss: 0.6383
Epoch 34/80
26/26            17s 470ms/step -
accuracy: 0.7586 - loss: 0.6124 - val_accuracy: 0.7518 - val_loss: 0.6703
Epoch 35/80
26/26            17s 497ms/step -
accuracy: 0.7806 - loss: 0.5666 - val_accuracy: 0.7446 - val_loss: 0.6313
Epoch 36/80
26/26            17s 476ms/step -
accuracy: 0.7484 - loss: 0.6149 - val_accuracy: 0.7572 - val_loss: 0.6619
Epoch 37/80
26/26            17s 471ms/step -
accuracy: 0.7910 - loss: 0.5702 - val_accuracy: 0.7374 - val_loss: 0.6829
Epoch 38/80
26/26            17s 482ms/step -
accuracy: 0.7661 - loss: 0.5721 - val_accuracy: 0.7194 - val_loss: 0.7233
Epoch 39/80
26/26            17s 468ms/step -
accuracy: 0.7779 - loss: 0.5505 - val_accuracy: 0.7464 - val_loss: 0.7346
Epoch 40/80
26/26            17s 475ms/step -
accuracy: 0.7716 - loss: 0.6034 - val_accuracy: 0.7518 - val_loss: 0.6711
Epoch 41/80
26/26            17s 470ms/step -
accuracy: 0.7881 - loss: 0.5638 - val_accuracy: 0.7374 - val_loss: 0.6571
Epoch 42/80
26/26            17s 474ms/step -
accuracy: 0.7714 - loss: 0.5705 - val_accuracy: 0.7590 - val_loss: 0.6657
```

```
Epoch 43/80
26/26            17s 472ms/step -
accuracy: 0.7745 - loss: 0.5607 - val_accuracy: 0.7536 - val_loss: 0.7577
Epoch 44/80
26/26            16s 468ms/step -
accuracy: 0.7762 - loss: 0.5722 - val_accuracy: 0.7608 - val_loss: 0.6562
Epoch 45/80
26/26            17s 478ms/step -
accuracy: 0.7834 - loss: 0.5307 - val_accuracy: 0.7608 - val_loss: 0.6839
Epoch 46/80
26/26            17s 472ms/step -
accuracy: 0.7986 - loss: 0.5268 - val_accuracy: 0.7536 - val_loss: 0.6471
Epoch 47/80
26/26            16s 469ms/step -
accuracy: 0.8074 - loss: 0.5120 - val_accuracy: 0.7446 - val_loss: 0.6737
Epoch 48/80
26/26            17s 475ms/step -
accuracy: 0.8130 - loss: 0.4894 - val_accuracy: 0.7662 - val_loss: 0.6530
Epoch 49/80
26/26            17s 473ms/step -
accuracy: 0.7844 - loss: 0.5557 - val_accuracy: 0.7302 - val_loss: 0.7285
Epoch 50/80
26/26            17s 470ms/step -
accuracy: 0.7984 - loss: 0.5265 - val_accuracy: 0.7608 - val_loss: 0.6726
Epoch 51/80
26/26            17s 473ms/step -
accuracy: 0.8021 - loss: 0.5045 - val_accuracy: 0.7320 - val_loss: 0.7461
Epoch 52/80
26/26            17s 478ms/step -
accuracy: 0.7991 - loss: 0.5145 - val_accuracy: 0.7518 - val_loss: 0.7116
Epoch 53/80
26/26            17s 473ms/step -
accuracy: 0.8103 - loss: 0.4918 - val_accuracy: 0.7410 - val_loss: 0.8059
Epoch 54/80
26/26            17s 475ms/step -
accuracy: 0.8056 - loss: 0.4951 - val_accuracy: 0.7572 - val_loss: 0.6842
Epoch 55/80
26/26            17s 471ms/step -
accuracy: 0.8298 - loss: 0.4482 - val_accuracy: 0.7500 - val_loss: 0.6918
Epoch 56/80
26/26            16s 483ms/step -
accuracy: 0.8094 - loss: 0.4988 - val_accuracy: 0.7608 - val_loss: 0.6979
Epoch 57/80
26/26            17s 478ms/step -
accuracy: 0.8168 - loss: 0.4781 - val_accuracy: 0.7356 - val_loss: 0.7538
Epoch 58/80
26/26            17s 485ms/step -
accuracy: 0.8353 - loss: 0.4365 - val_accuracy: 0.7554 - val_loss: 0.7459
```

```
Epoch 59/80
26/26            17s 475ms/step -
accuracy: 0.8198 - loss: 0.4643 - val_accuracy: 0.7482 - val_loss: 0.7176
Epoch 60/80
26/26            17s 471ms/step -
accuracy: 0.8289 - loss: 0.4629 - val_accuracy: 0.7464 - val_loss: 0.7230
Epoch 61/80
26/26            17s 477ms/step -
accuracy: 0.8172 - loss: 0.4613 - val_accuracy: 0.7608 - val_loss: 0.7331
Epoch 62/80
26/26            16s 451ms/step -
accuracy: 0.8412 - loss: 0.4339 - val_accuracy: 0.7518 - val_loss: 0.6861
Epoch 63/80
26/26            16s 454ms/step -
accuracy: 0.8275 - loss: 0.4445 - val_accuracy: 0.7500 - val_loss: 0.7241
Epoch 64/80
26/26            16s 468ms/step -
accuracy: 0.8285 - loss: 0.4629 - val_accuracy: 0.7788 - val_loss: 0.6524
Epoch 65/80
26/26            16s 454ms/step -
accuracy: 0.8429 - loss: 0.4040 - val_accuracy: 0.7770 - val_loss: 0.6560
Epoch 66/80
26/26            16s 449ms/step -
accuracy: 0.8292 - loss: 0.4435 - val_accuracy: 0.7518 - val_loss: 0.7633
Epoch 67/80
26/26            16s 466ms/step -
accuracy: 0.8322 - loss: 0.4289 - val_accuracy: 0.7734 - val_loss: 0.6385
Epoch 68/80
26/26            16s 464ms/step -
accuracy: 0.8344 - loss: 0.4283 - val_accuracy: 0.7788 - val_loss: 0.6329
Epoch 69/80
26/26            16s 454ms/step -
accuracy: 0.8426 - loss: 0.4133 - val_accuracy: 0.7356 - val_loss: 0.7341
Epoch 70/80
26/26            17s 495ms/step -
accuracy: 0.8360 - loss: 0.4202 - val_accuracy: 0.7680 - val_loss: 0.6748
Epoch 71/80
26/26            16s 462ms/step -
accuracy: 0.8496 - loss: 0.4052 - val_accuracy: 0.7464 - val_loss: 0.7357
Epoch 72/80
26/26            16s 447ms/step -
accuracy: 0.8571 - loss: 0.3997 - val_accuracy: 0.7698 - val_loss: 0.6658
Epoch 73/80
26/26            16s 456ms/step -
accuracy: 0.8505 - loss: 0.3881 - val_accuracy: 0.7770 - val_loss: 0.6845
Epoch 74/80
26/26            16s 453ms/step -
accuracy: 0.8511 - loss: 0.3956 - val_accuracy: 0.7716 - val_loss: 0.6605
```

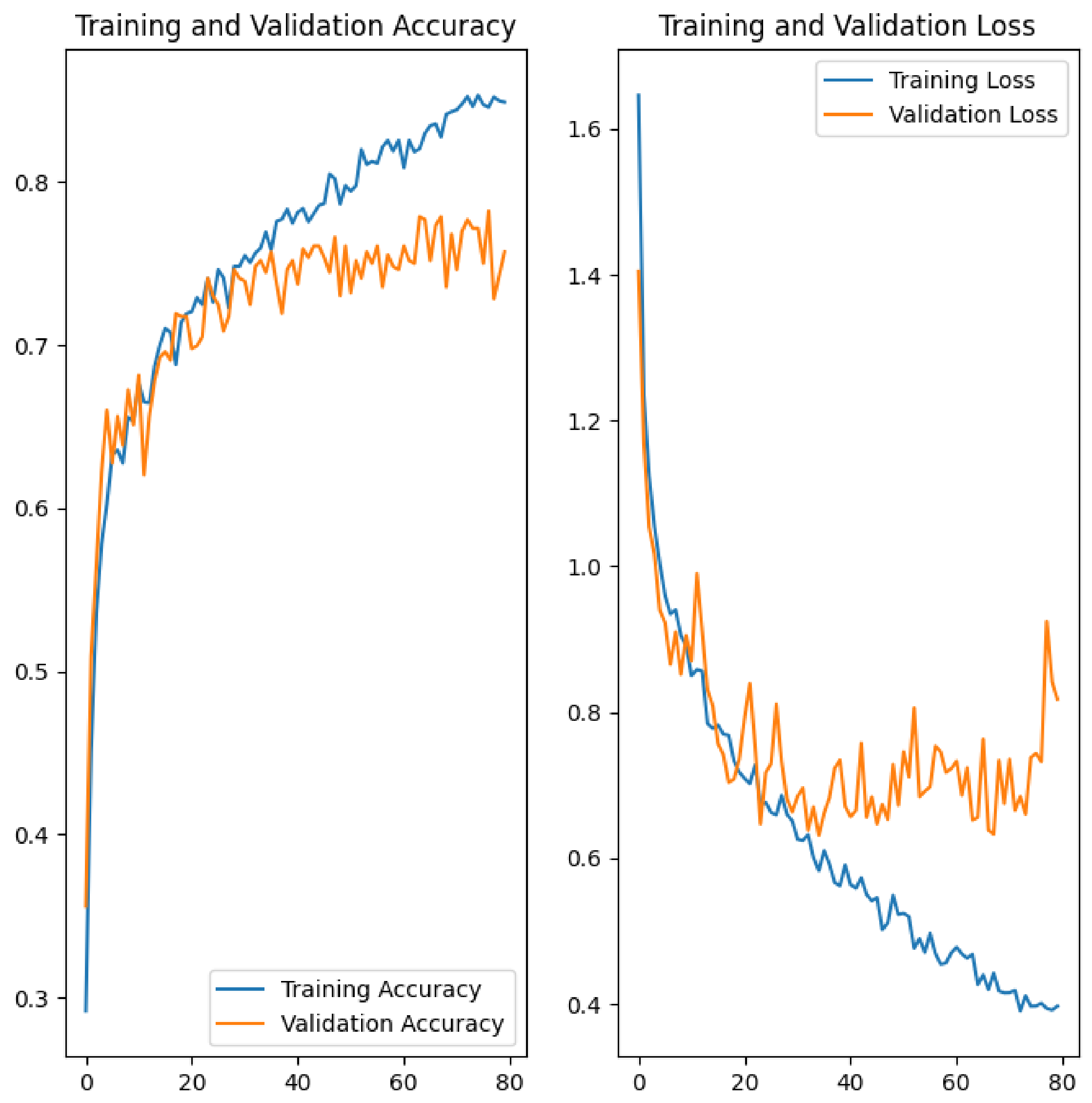
```
Epoch 75/80
26/26                16s 449ms/step -
accuracy: 0.8616 - loss: 0.3799 - val_accuracy: 0.7716 - val_loss: 0.7381
Epoch 76/80
26/26                16s 455ms/step -
accuracy: 0.8555 - loss: 0.3890 - val_accuracy: 0.7500 - val_loss: 0.7437
Epoch 77/80
26/26                16s 460ms/step -
accuracy: 0.8439 - loss: 0.4050 - val_accuracy: 0.7824 - val_loss: 0.7323
Epoch 78/80
26/26                16s 449ms/step -
accuracy: 0.8638 - loss: 0.3705 - val_accuracy: 0.7284 - val_loss: 0.9249
Epoch 79/80
26/26                16s 446ms/step -
accuracy: 0.8475 - loss: 0.3902 - val_accuracy: 0.7428 - val_loss: 0.8421
Epoch 80/80
26/26                16s 446ms/step -
accuracy: 0.8396 - loss: 0.4163 - val_accuracy: 0.7572 - val_loss: 0.8178
```

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 =
```

[51]:
```
## 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')
plt.plot(epochs_range, val_acc, label='Validation Accuracy')
plt.legend(loc='lower right')
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.title('Training and Validation Loss')
plt.show()
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[51], line 26
```

```
     23 plt.title('Training and Validation Loss')
     24 plt.show()
---> 26 print(round(history.history['accuracy']))

TypeError: type list doesn't define __round__ method
```

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?

[ ]: