

A2: Image classification in the browser

In this assignment, you will gain real-world experience with image classification on small datasets - a common scenario in practice. There are three parts.

1. First, you will train a model on a small existing dataset.
2. Next, you will collect a small dataset yourself. Of course, in practice there often isn't a dataset available for tasks you care about, so it's valuable to get a feel for this process. You will train a small model from scratch, then use data augmentation to improve accuracy.
3. Finally, you will run your model from part two in a browser using TensorFlow.js. You will be able to upload a photo, and your model will classify it.

Instructions

Please complete this notebook by searching for "TODO".

Submission instructions

Please submit this assignment on CourseWorks by uploading a Jupyter notebook that includes saved output. If you are working in Colab, you can prepare your notebook for submission by ensuring that runs end-to-end, then saving and downloading it:

1. Runtime -> Restart and run all
2. File -> Save
3. File -> Download.ipynb

Note: you will need to include a screenshot of your model running in a webpage for part 3, see the final TODO at the bottom of this notebook.

Setup instructions

1. If you are running this notebook in Colab, make sure a GPU is enabled (Edit -> Notebook settings -> Hardware accelerator).

In [1]:

```
import tensorflow as tf
print(tf.__version__)
```

2.9.2

1a) Flowers

In this part of the assignment, you will train a model on a small existing dataset (flowers).



In [2]:

```
import IPython.display as display
import matplotlib.pyplot as plt
import random
import time

from tensorflow.keras import datasets, layers, models

AUTOTUNE = tf.data.experimental.AUTOTUNE
BATCH_SIZE = 32
IMG_SIZE = 192
SHUFFLE_SIZE = 1000
```

Download the flowers dataset

In [3]:

```
import pathlib
data_root_orig = tf.keras.utils.get_file(origin='https://storage.googleapis.com/download-public/keras/datasets/flower_photos.tar.gz',
                                          fname='flower_photos', untar=True)
data_root = pathlib.Path(data_root_orig)
print(data_root)
```

```
/root/.keras/datasets/flower_photos
```

Write an input pipeline from scratch

There are several ways to load images in TensorFlow. Later in this assignment, you'll use the [Keras preprocessing utilities](https://keras.io/preprocessing/image/) (<https://keras.io/preprocessing/image/>). For starters, though, you'll see how-to write your own using `tf.data`, based on this [tutorial](https://www.tensorflow.org/tutorials/load_data/images) (https://www.tensorflow.org/tutorials/load_data/images). This is valuable to do once (just so you can see how the nuts and bolts work) before using the higher level utils. This code is written for you as an example. Follow along and try to understand each piece.

In [4]:

```
all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]
random.shuffle(all_image_paths)

image_count = len(all_image_paths)
image_count
```

Out[4]:

```
3670
```



In [5]:

```
all_image_paths[:5]
```

Out[5]:

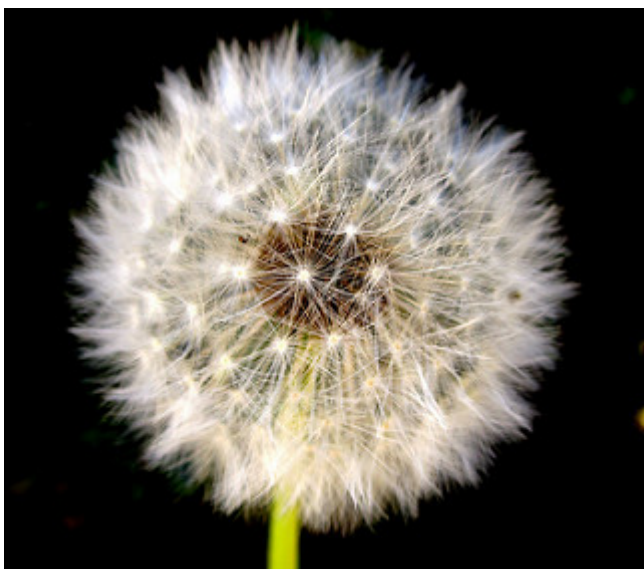
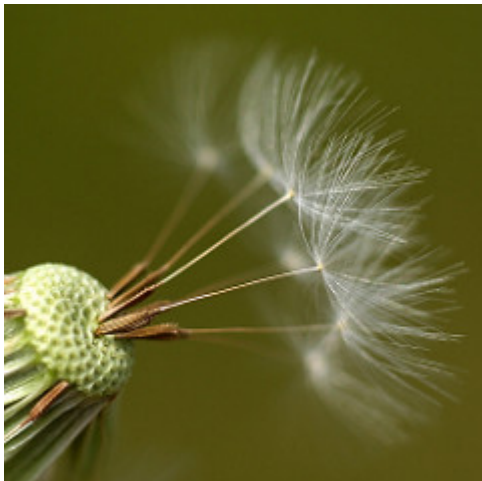
```
['/root/.keras/datasets/flower_photos/dandelion/8831808134_315aedb37b.jpg',  
 '/root/.keras/datasets/flower_photos/sunflowers/15380755137_a2e67839ab_m.jpg',  
 '/root/.keras/datasets/flower_photos/sunflowers/22755811033_cd17b109e0.jpg',  
 '/root/.keras/datasets/flower_photos/dandelion/2521827947_9d237779bb_n.jpg',  
 '/root/.keras/datasets/flower_photos/roses/3500121696_5b6a69effb_n.jpg']
```

Get to know your data



In [6]:

```
for n in range(3):  
    image_path = random.choice(all_image_paths)  
    display.display(display.Image(image_path))  
    print()
```



Classes are given by directory names

In [7]:

```
label_names = sorted(item.name for item in data_root.glob('*/*') if item.is_dir())
label_names
```

Out[7]:

```
['daisy', 'dandelion', 'roses', 'sunflowers', 'tulips']
```

In [8]:

```
label_to_index = dict((name, index) for index, name in enumerate(label_names))
label_to_index
```

Out[8]:

```
{'daisy': 0, 'dandelion': 1, 'roses': 2, 'sunflowers': 3, 'tulips': 4}
```

In [9]:

```
all_labels = [label_to_index[pathlib.Path(path).parent.name]
               for path in all_image_paths]

print("First 10 labels indices: ", all_labels[:10])
```

```
First 10 labels indices:  [1, 3, 3, 1, 2, 1, 3, 3, 0, 2]
```

Create a train/test split

In [10]:

```
from sklearn.model_selection import train_test_split
train_paths, test_paths, train_labels, test_labels = train_test_split(all_image_paths,
```

Display a few images and their labels



In [11]:

```
for n in range(3):  
    i = random.randint(0, len(train_paths))  
    image_path = train_paths[i]  
    print(label_names[train_labels[i]])  
    display.display(display.Image(image_path))  
    print()
```

dandelion



sunflowers



dandelion





Begin using TensorFlow ops to read and decode the images

In [12]:

```
img_raw = tf.io.read_file(train_paths[0])
print(repr(img_raw)[:100]+"...")
```

```
<tf.Tensor: shape=(), dtype=string, numpy=b'\xff\xd8\xff\xe0\x00\x10JF
IF\x00\x01\x01\x01\x00H\x00H\x...
```

In [13]:

```
img_tensor = tf.image.decode_image(img_raw)
print(img_tensor.shape)
print(img_tensor.dtype)
```

```
(375, 500, 3)
<dtype: 'uint8'>
```

In [14]:

```
img_final = tf.image.resize(img_tensor, [IMG_SIZE, IMG_SIZE])
img_final = img_final / 255.0 # normalize pixel values
print(img_final.shape)
print(img_final.numpy().min())
print(img_final.numpy().max())
```

```
(192, 192, 3)
0.0
1.0
```

Wrap those in a function

Tip: pay careful attention to the preprocessing. When you deploy models in the browser, you will need to ensure that images are preprocessed identically in JavaScript as they are in Python.



In [15]:

```
def load_and_preprocess_image(path):  
    img = tf.io.read_file(path)  
    img = tf.image.decode_jpeg(img, channels=3)  
    img = tf.image.resize(img, [IMG_SIZE, IMG_SIZE])  
    img /= 255.0 # normalize pixels to 0,1  
    return img
```

In [16]:

```
def show(img, label):  
    plt.imshow(img)  
    plt.title(label)  
    plt.xticks([])  
    plt.yticks([])  
    print()  
  
img_path = train_paths[0]  
img = load_and_preprocess_image(img_path)  
label = label_names[train_labels[0]]  
show(img, label)
```

tulips



Build an input pipeline to return images and labels

I realize this is complicated. The problem we're trying to solve using `tf.data` is performance (we want our preprocessing to run in C, but to write our code in Python). There are a bunch of advanced tricks you can do with `tf.data` as well (e.g. prefetching images to the GPU).

Note: although your *peak* performance can be higher, it's also very easy to make mistakes and end up with code that's super slow. Always benchmark your input pipelines before using them (shown in a bit).



In [17]:

```
# a dataset that returns image paths
path_ds = tf.data.Dataset.from_tensor_slices(train_paths)
for n, img_path in enumerate(path_ds.take(4)):
    print(n, img_path)

0 tf.Tensor(b'/root/.keras/datasets/flower_photos/tulips/110147301_ad9
21e2828.jpg', shape=(), dtype=string)
1 tf.Tensor(b'/root/.keras/datasets/flower_photos/roses/3141434519_aaa
64c4f65_n.jpg', shape=(), dtype=string)
2 tf.Tensor(b'/root/.keras/datasets/flower_photos/roses/13231224664_4a
f5293a37.jpg', shape=(), dtype=string)
3 tf.Tensor(b'/root/.keras/datasets/flower_photos/roses/1562198683_8cd
8cb5876_n.jpg', shape=(), dtype=string)
```

In [18]:

```
# a dataset that returns images (loaded off disk, decoded, and preprocessed)
image_ds = path_ds.map(load_and_preprocess_image, num_parallel_calls=AUTOTUNE)
for n, image in enumerate(image_ds.take(4)):
    print(n, image.shape)

0 (192, 192, 3)
1 (192, 192, 3)
2 (192, 192, 3)
3 (192, 192, 3)
```

In [19]:

```
# a dataset that returns labels
label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(train_labels, tf.int64))
for label in label_ds.take(4):
    print(label_names[label.numpy()])
```

```
tulips
roses
roses
roses
```

In [20]:

```
# a dataset that returns images and labels
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
print(image_label_ds)
```

```
<ZipDataset element_spec=(TensorSpec(shape=(192, 192, 3), dtype=tf.flo
at32, name=None), TensorSpec(shape=(), dtype=tf.int64, name=None))>
```

In [21]:

```
for img, label in image_label_ds.take(2):
    print(img.shape, label_names[label.numpy()])
```

```
(192, 192, 3) tulips
(192, 192, 3) roses
```



Batch and shuffle

Why do we need to specify a `shuffle_size` parameter? `tf.data` works with streams (it doesn't know their length). To shuffle items, we maintain an in-memory buffer of this size.

In [22]:

```
ds = image_label_ds.shuffle(SHUFFLE_SIZE)
ds = ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)

for img, label in ds.take(2):
    print(img.shape, label.shape) # notice it's returning batches of data now

(32, 192, 192, 3) (32,)
(32, 192, 192, 3) (32,)
```

At this point, you could use the dataset above to train a model with `model.fit(ds)` but first, let's improve performance. As written, the dataset will load each image off disk, one at a time (super slow). Instead, we want to cache them in memory.

Improve performance

In [23]:

```
# A benchmark utility to time how long it takes
# to iterate once over the entire dataset
def time_one_epoch(ds):
    start = time.time()
    batches = 0
    for i, (images, labels) in enumerate(ds):
        batches += 1
        if i % 10 == 0:
            print('.', end='')
    print()
    end = time.time()
    duration = end - start
    print("Read {} batches".format(batches))
    print("{:0.2f} Batches/s".format(batches/duration))
    print("{:0.2f} Images/s".format(BATCH_SIZE*batches/duration))
    print("Total time: {}s".format(duration))
```

Use in-memory caching

This is a small dataset, so let's keep it in memory. The first time we iterate over this dataset, images will be loaded off disk, then cached. The first iteration will be quite slow, and subsequent ones will be faster. Let's show that.

In [24]:

```
ds = image_label_ds.cache() # cache data in mempry
ds = ds.shuffle(SHUFFLE_SIZE)
ds = ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)
```



In [25]:

```
time_one_epoch(ds) # this will be slow
```

```
.....
Read 86 batches
14.57 Batches/s
466.24 Images/s
Total time: 5.902532577514648s
```

Now that the cache is built, iteration will be much faster.

In [26]:

```
time_one_epoch(ds) # this will be fast
```

```
.....
Read 86 batches
205.91 Batches/s
6588.97 Images/s
Total time: 0.4176676273345947s
```

How-to use on-disk caching

If the dataset did not fit into memory, you could use a cache file on disk, like this:

```
ds = image_label_ds.cache(filename='./cache.tf-data')
ds = ds.shuffle(buffer_size=BUFFER_SIZE)
ds = ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)
```

This can be useful to perform expensive preprocessing only once, and/or to improve file I/O (TF saves the cache file in an efficient format - it can be faster to read one large file than a bunch of small ones). For now, we'll keep it in memory.

In [27]:

```
# here's our final training dataset
train_ds = image_label_ds.cache()
train_ds = train_ds.shuffle(SHUFFLE_SIZE)
train_ds = train_ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)

# let's make a test dataset as well
path_ds = tf.data.Dataset.from_tensor_slices(test_paths)
image_ds = path_ds.map(load_and_preprocess_image, num_parallel_calls=AUTOTUNE)
label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(test_labels, tf.int64))
image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
test_ds = image_label_ds.cache().batch(BATCH_SIZE)
```

Finally, we have a fast pipeline ready to go, written from scratch

Now, let's define a model.



1b) Create a simple CNN

This is our baseline model, it will not be very accurate. You'll improve it below.

In [28]:

```
model = models.Sequential()  
model.add(layers.Conv2D(16, (3, 3), activation='relu',  
                        input_shape=(IMG_SIZE, IMG_SIZE, 3)))  
model.add(layers.MaxPooling2D())
```

In [29]:

```
model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 190, 190, 16)	448
max_pooling2d (MaxPooling2D)	(None, 95, 95, 16)	0
Total params: 448		
Trainable params: 448		
Non-trainable params: 0		

In [30]:

```
model.add(layers.Flatten())  
model.add(layers.Dense(5, activation='softmax'))
```

Tip: pay attention to the size of the model

Later, when you export a model to run in the webpage, you will want a small one (in terms of the number of parameters) that downloads quickly. Notice how much more efficient convolutional layers are than the dense layers (ask yourself, why?)

Tip: pay attention to exactly how your images are preprocessed

Later, when you run your model in a browser, you'll need to preprocess images in JavaScript in exactly the same way.



In [31]:

```
model.summary()
```

```
Model: "sequential"
```

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 190, 190, 16)	448
max_pooling2d (MaxPooling2D)	(None, 95, 95, 16)	0
flatten (Flatten)	(None, 144400)	0
dense (Dense)	(None, 5)	722005
Total params: 722,453		
Trainable params: 722,453		
Non-trainable params: 0		

In [32]:

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

In [33]:

```
# I realize we're not using a separate test set, that's fine
# for this assignment
model.fit(train_ds, validation_data=test_ds, epochs=5)
```

```
Epoch 1/5
86/86 [=====] - 19s 131ms/step - loss: 2.6577
- accuracy: 0.3993 - val_loss: 1.2820 - val_accuracy: 0.4880
Epoch 2/5
86/86 [=====] - 2s 20ms/step - loss: 0.9924 -
accuracy: 0.6286 - val_loss: 1.2296 - val_accuracy: 0.5185
Epoch 3/5
86/86 [=====] - 2s 19ms/step - loss: 0.6318 -
accuracy: 0.7874 - val_loss: 1.1849 - val_accuracy: 0.5370
Epoch 4/5
86/86 [=====] - 2s 19ms/step - loss: 0.3688 -
accuracy: 0.9037 - val_loss: 1.3583 - val_accuracy: 0.5512
Epoch 5/5
86/86 [=====] - 2s 19ms/step - loss: 0.2096 -
accuracy: 0.9542 - val_loss: 1.2690 - val_accuracy: 0.5806
```

Out[33]:

```
<keras.callbacks.History at 0x7fa8e02c6810>
```

Make predictions on a single image



Tip: models are implemented to make predictions on batches of images for efficiency. This means that to make a prediction on a single image, you'll need to first wrap it in a batch. The syntax can feel a little unusual at first, but gets easier with time.

In [34]:

```
# load an image off disk
img_index = 0
img = load_and_preprocess_image(train_paths[img_index])

print(img.shape) # before

#####
## TODO: your code here
## use tf.expand_dims to create an empty batch dimension
## the starting image shape is (192, 192, 3)
## you want it to be (1, 192, 192, 3)
## that's read as "a batch of 1 image, with 192 rows, 192 cols,
## and 3 color channels"
#####

im_batch = tf.expand_dims(img, axis=0) # YOUR CODE HERE, use tf.expand_dims

print(im_batch.shape) # after

# make predictions
batch_pred = model.predict(im_batch) # returns a list of predictions
pred = batch_pred[0]

print("Prediction", label_names[tf.argmax(pred)])
print("Actual", label_names[train_labels[img_index]])

(192, 192, 3)
(1, 192, 192, 3)
1/1 [=====] - 0s 79ms/step
Prediction tulips
Actual tulips
```

TODO: Improve accuracy

In the code cell below, write a new model that's more accurate than the baseline above. Define and train your model, and create plots of loss / accuracy as a function of epochs. Try to train your model to high accuracy, without overfitting. For this assignment, it is not necessary to build a super accurate model (just experiment a bit and try to improve over the baseline).



In [35]:

```
# TODO: your code here
model_v2 = models.Sequential()
model_v2.add(layers.Conv2D(16, (3, 3), activation='relu', input_shape=(IMG_SIZE, IMG_SIZE, 3)))
model_v2.add(layers.MaxPooling2D())

model_v2.add(layers.Conv2D(32, (3, 3), activation='relu'))
model_v2.add(layers.MaxPooling2D())

model_v2.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_v2.add(layers.MaxPooling2D())

model_v2.add(layers.Conv2D(128, (3, 3), activation='relu'))
model_v2.add(layers.MaxPooling2D())

model_v2.add(layers.Conv2D(64, (3, 3), activation='relu'))
model_v2.add(layers.MaxPooling2D())

model_v2.add(layers.Flatten())
model_v2.add(layers.Dense(5, activation='softmax'))
```

In [36]:

```
model_v2.compile(optimizer='adam',
                 loss='sparse_categorical_crossentropy',
                 metrics=['accuracy'])


model_v2.fit(train_ds, validation_data=test_ds, epochs=5)
```

```
Epoch 1/5
86/86 [=====] - 4s 34ms/step - loss: 1.2960 - accuracy: 0.4295 - val_loss: 1.2060 - val_accuracy: 0.4586
Epoch 2/5
86/86 [=====] - 3s 30ms/step - loss: 1.0995 - accuracy: 0.5447 - val_loss: 1.1167 - val_accuracy: 0.5512
Epoch 3/5
86/86 [=====] - 3s 30ms/step - loss: 0.9962 - accuracy: 0.5985 - val_loss: 0.9572 - val_accuracy: 0.6122
Epoch 4/5
86/86 [=====] - 3s 29ms/step - loss: 0.8726 - accuracy: 0.6690 - val_loss: 0.8898 - val_accuracy: 0.6351
Epoch 5/5
86/86 [=====] - 2s 29ms/step - loss: 0.7990 - accuracy: 0.6944 - val_loss: 0.8559 - val_accuracy: 0.6449
```

Out[36]:

```
<keras.callbacks.History at 0x7fa88c620c90>
```

1c) Practice deploying your flowers classifier using TensorFlow.js

In the final part of the assignment, you'll export the model you build to recognize landmarks on  Columbia's campus, and get it working in the browser. If you're new to JavaScript (as most of us are), the mechanics will be difficult at first.

Let's practice by exporting your flowers classifier, and getting it working with TensorFlow.js (the starter code provided is written to work with flowers model).

Save your model

In [37]:

```
# model.save("/content/my_model_flower.h5")
```

In [38]:

```
!ls -lha
```

```
total 12M
drwxr-xr-x 1 root root 4.0K Oct 19 03:16 .
drwxr-xr-x 1 root root 4.0K Oct 19 01:57 ..
drwxr-xr-x 4 root root 4.0K Oct 17 13:43 .config
drwx----- 5 root root 4.0K Oct 19 02:13 drive
-rw-r--r-- 1 root root 3.7M Oct 19 05:52 my_model_columbia.h5
-rw-r--r-- 1 root root 8.3M Oct 19 02:46 my_model_flower.h5
drwxr-xr-x 1 root root 4.0K Oct 17 13:44 sample_data
```

Download your saved model to your local machine

In [39]:

```
from google.colab import files
# files.download("/content/my_model_flower.h5")
```

Visit notebook #2 (a2-2.ipynb on CourseWorks) to convert your saved model into to TensorFlow.js format

Follow the instructions there to prepare a webpage to run your flowers model in the browser. Once you have that working, you can continue with the rest of this assignment in this notebook.

1d) Classify flowers using transfer learning

In this part of the assignment, you'll use transfer learning to take advantage of a large pretrained model. It is not necessary to deploy this part in the browser.

Read this tutorial before completing this section:

- https://www.tensorflow.org/tutorials/images/transfer_learning
(https://www.tensorflow.org/tutorials/images/transfer_learning)



In [40]:

```
## TODO: your code here
# Choose a pretrained model, and import the application
# See https://keras.io/applications/ for a few choices
# When you import the model, you will want to remove the
# final dense layer that performs classification (include_top=False)
# you will also want to import weights from ImageNet,
# and you will want to specify the input shape to match your images.

base_model =tf.keras.applications.Xception(
    include_top=False,
    weights="imagenet",
    input_tensor=None,
    input_shape=(IMG_SIZE, IMG_SIZE, 3),
    pooling=None,
    classes=1000,
    classifier_activation="softmax",
)
# fix me. base_model = tf.keras.applications...
```

In [41]:

```
# A hack to show you the output shape of the model
for image_batch, label_batch in train_ds.take(1):
    pass

feature_batch = base_model(image_batch)
print(feature_batch.shape)

(32, 6, 6, 2048)
```

In [42]:

```
# do not update the pretrained weights during training
# (we won't use finetuning here)
base_model.trainable = False
```



In [43]:

```
base_model.summary()
```

```
Model: "xception"
```

Layer (type) connected to	Output Shape	Param #	Connections
input_1 (InputLayer)	(None, 192, 192, 3)	0	input_1
block1_conv1 (Conv2D)	(None, 95, 95, 32)	864	input_1[0][0]
block1_conv1_bn (BatchNormalization)	(None, 95, 95, 32)	128	block1_conv1[0][0]
block1_conv1_act (Activation)	(None, 95, 95, 32)	0	block1_conv1_bn[0][0]

In [44]:

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
prediction_layer = tf.keras.layers.Dense(5, activation='softmax')
```

In [45]:

```
# build a new model reusing the pretrained base
model = tf.keras.Sequential([
    base_model,
    global_average_layer,
    prediction_layer
])
```

In [46]:

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



In [47]:

```
model.fit(train_ds, validation_data=test_ds, epochs=10)
```

```
Epoch 1/10
86/86 [=====] - 16s 152ms/step - loss: 0.7310
- accuracy: 0.7475 - val_loss: 0.5047 - val_accuracy: 0.8257
Epoch 2/10
86/86 [=====] - 11s 130ms/step - loss: 0.4242
- accuracy: 0.8612 - val_loss: 0.4300 - val_accuracy: 0.8464
Epoch 3/10
86/86 [=====] - 11s 133ms/step - loss: 0.3501
- accuracy: 0.8874 - val_loss: 0.4156 - val_accuracy: 0.8540
Epoch 4/10
86/86 [=====] - 11s 132ms/step - loss: 0.3026
- accuracy: 0.9092 - val_loss: 0.3894 - val_accuracy: 0.8671
Epoch 5/10
86/86 [=====] - 11s 131ms/step - loss: 0.2668
- accuracy: 0.9251 - val_loss: 0.3730 - val_accuracy: 0.8769
Epoch 6/10
86/86 [=====] - 11s 131ms/step - loss: 0.2398
- accuracy: 0.9281 - val_loss: 0.3694 - val_accuracy: 0.8736
Epoch 7/10
86/86 [=====] - 11s 129ms/step - loss: 0.2204
- accuracy: 0.9382 - val_loss: 0.3627 - val_accuracy: 0.8769
Epoch 8/10
86/86 [=====] - 11s 131ms/step - loss: 0.2013
- accuracy: 0.9451 - val_loss: 0.3535 - val_accuracy: 0.8791
Epoch 9/10
86/86 [=====] - 11s 131ms/step - loss: 0.1870
- accuracy: 0.9488 - val_loss: 0.3470 - val_accuracy: 0.8791
Epoch 10/10
86/86 [=====] - 11s 127ms/step - loss: 0.1693
- accuracy: 0.9575 - val_loss: 0.3468 - val_accuracy: 0.8802
```

Out[47]:

```
<keras.callbacks.History at 0x7fa88c7f4150>
```

2a) Recognize landmarks on Columbia's campus

In this part of the assignment, you will train a model to recognize landmarks (famous places, like the [Alma Mater sculpture \(https://en.wikipedia.org/wiki/Alma_Mater_\(New_York_sculpture\)\)](https://en.wikipedia.org/wiki/Alma_Mater_(New_York_sculpture)), or Butler library) on Columbia's campus. Instead of tf.data, you will gain experience with the higher level Keras utilities. You will also experiment with data augmentation to increase the effective size of your dataset.

Starter code is not provided for this part of the assignment.

The goal is to train a small, relatively simple model from scratch (without using transfer learning) that you can convert to TensorFlow.js format, and run in a webpage.

You should base your work off the following tutorials:

- <https://www.tensorflow.org/tutorials/images/classification>
(<https://www.tensorflow.org/tutorials/images/classification>)



Note: the `layers.Rescaling` used in this tutorial is not yet supported by TensorFlow.js. You must apply the rescaling layer to your dataset (and not include it inside your model) before heading to the next step in this assignment. An example of applying rescaling to the dataset is given inside the tutorial. And the starter code (.js and .html files) provided in part two of the assignment already includes the appropriate rescaling in JavaScript).

Here are steps you should complete.

1. Collect a dataset of at least three landmarks. Your dataset should include at least 50 images of each landmark in train, and 50 in validation (using more images is fine). Randomly shuffle your data to create these splits. You do not need to use a separate test set in this assignment.

You will need to upload your dataset to Colab in order to train a model. To do so, you can either upload it using the file browser (slow and tedious, you'll need to repeat that if your runtime is reset), or (better) you can upload your dataset to Google Drive, then mount your drive as a filesystem in Colab (View -> Table of contents -> Code snippets -> search for "drive"). This will enable you to access the contents of your drive with commands like `!ls /gdrive`. As a another option, you could upload your dataset to a cloud provider or your Columbia account, then download it similarly to flowers at the top of this notebook.

2. Write a model to classify your data. Try to train a small model (in terms of the number of parameters). To make it as easy as possible to convert your layer to TensorFlow.js format later, do not use transfer learning (if you would like to in the future after you get your simple model working, go for it - but it's not necessary).
3. Show predictions on several images.
4. Use data augmentation, see if this helps to improve accuracy.
5. Produce plots of accuracy / loss as a function of epochs. Determine the right place to stop training without overfitting.

When you have a reasonably accurate model, proceed to the next step. There are no guidelines for accuracy, try to build something you feel works well, given the small amount of data you have.

In [48]:

```
###
# TODO: your code for 2a here.
###
import numpy as np
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.models import Sequential
import random
import IPython.display as display
import PIL
from google.colab import drive
from tensorflow.keras.preprocessing.image import ImageDataGenerator
```

In [49]:

```
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call `drive.mount("/content/drive", force_remount=True)`.



In [50]:

```
!ls drive/MyDrive/dataset
```

AlmaMater butler lion

In [51]:

```
data_root = pathlib.Path('drive/MyDrive/dataset')  
print(data_root)
```

drive/MyDrive/dataset

In [52]:

```
dataroot = list(data_root.glob('*'))  
dataroot
```

Out[52]:

```
[PosixPath('drive/MyDrive/dataset/.DS_Store'),  
 PosixPath('drive/MyDrive/dataset/butler'),  
 PosixPath('drive/MyDrive/dataset/AlmaMater'),  
 PosixPath('drive/MyDrive/dataset/lion')]
```

In [53]:

```
lion = list(data_root.glob('lion/*'))  
PIL.Image.open(str(lion[56]))
```

Out[53]:



In [54]:

```
Alma = list(data_root.glob('AlmaMater/*'))  
PIL.Image.open(str(Alma[11]))
```

Out[54]:



In [55]:

```
butler = list(data_root.glob('butler/*'))
PIL.Image.open(str(butler[9]))
```

Out[55]:



In [56]:

```
all_image_paths = list(data_root.glob('*/*'))
all_image_paths = [str(path) for path in all_image_paths]
random.shuffle(all_image_paths)

label_names = sorted(item.name for item in data_root.glob('*/*') if item.is_dir())
label_to_index = dict((name, index) for index, name in enumerate(label_names))
all_labels = [label_to_index[pathlib.Path(path).parent.name] for path in all_image_paths]
image_count = len(all_image_paths)
image_count
```

Out[56]:

300

In [57]:

```
all_image_paths[:10]
```

Out[57]:

```
['drive/MyDrive/dataset/lion/images (61).jpeg',
 'drive/MyDrive/dataset/butler/1280_profile.jpeg',
 'drive/MyDrive/dataset/lion/下载 (15).jpeg',
 'drive/MyDrive/dataset/lion/下载 (9).jpeg',
 'drive/MyDrive/dataset/butler/olyeblora46awr0vewgi.jpeg',
 'drive/MyDrive/dataset/AlmaMater/下载 (18).jpeg',
 'drive/MyDrive/dataset/AlmaMater/images (18).jpeg',
 'drive/MyDrive/dataset/AlmaMater/images (53).jpeg',
 'drive/MyDrive/dataset/butler/Butler-Banner-2019.jpeg',
 'drive/MyDrive/dataset/butler/images (81).jpeg']
```



In [58]:

```
for n in range(3):  
    path = random.choice(all_image_paths)  
  
    display.display(display.Image(path))  
    print(path)
```



drive/MyDrive/dataset/lion/Unknown.jpeg



drive/MyDrive/dataset/butler/DW_Butler_2019-09_11_jeh.jpeg





drive/MyDrive/dataset/lion/images (22).jpeg

In [59]:

```
BATCH_SIZE = 16  
img_height = 192  
img_width = 192
```

In [59]:



In [69]:

```

train_ds = tf.keras.utils.image_dataset_from_directory(
    data_root,
    validation_split=0.5,
    subset="training",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=BATCH_SIZE)

val_ds = tf.keras.utils.image_dataset_from_directory(
    data_root,
    validation_split=0.5,
    subset="validation",
    seed=42,
    image_size=(img_height, img_width),
    batch_size=BATCH_SIZE)

for image_batch, labels_batch in val_ds:
    print(image_batch.shape)
    print(labels_batch.shape)
    break

class_names = train_ds.class_names
print(class_names)

AUTOTUNE = tf.data.AUTOTUNE

train_ds = train_ds.cache().shuffle(1000).prefetch(buffer_size=AUTOTUNE)
val_ds = val_ds.cache().prefetch(buffer_size=AUTOTUNE)

# normalization_layer = layers.Rescaling(1./255)
# normalized_ds = train_ds.map(lambda x, y: (normalization_layer(x), y))
# image_batch, labels_batch = next(iter(normalized_ds))
# first_image = image_batch[0]
# # Notice the pixel values are now in `[0,1]`.
# print(np.min(first_image), np.max(first_image))

```

```

Found 300 files belonging to 3 classes.
Using 150 files for training.
Found 300 files belonging to 3 classes.
Using 150 files for validation.
(16, 192, 192, 3)
(16,)
['AlmaMater', 'butler', 'lion']

```

In [60]:

In [60]:

```

# following code replacing the above should also works well

```



In [61]:

```
# just similar as the above flower sample code
# train_paths, test_paths, train_labels, test_labels = train_test_split(all_image_pa

# path_ds = tf.data.Dataset.from_tensor_slices(train_paths)
# image_ds = path_ds.map(load_and_preprocess_image, num_parallel_calls=AUTOTUNE)
# label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(train_labels, tf.int64))
# image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
# ds = image_label_ds.shuffle(SHUFFLE_SIZE)
# ds = ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)

# train_ds = image_label_ds.cache()
# train_ds = train_ds.shuffle(SHUFFLE_SIZE)
# train_ds = train_ds.batch(BATCH_SIZE).prefetch(buffer_size=AUTOTUNE)

# path_ds = tf.data.Dataset.from_tensor_slices(test_paths)
# image_ds = path_ds.map(load_and_preprocess_image, num_parallel_calls=AUTOTUNE)
# label_ds = tf.data.Dataset.from_tensor_slices(tf.cast(test_labels, tf.int64))
# image_label_ds = tf.data.Dataset.zip((image_ds, label_ds))
# test_ds = image_label_ds.cache().batch(BATCH_SIZE)
```

In [61]:

In [72]:

```
# simple model without data augmentation

model = Sequential([
    layers.Rescaling(1./255, input_shape=(img_height, img_width, 3)),
    layers.Conv2D(16, (3,3), activation='relu', input_shape=(img_height, img_width, 3)),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(3, activation = 'sigmoid')
])

# tip for myself
# The difference between sparse_categorical_crossentropy and categorical_crossentropy
# whether your targets are one-hot encoded.
# https://stackoverflow.com/questions/58398491/valueerror-shape-mismatch-the-shape-c

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accu
```



In [73]:

```

epochs = 40
history = model.fit(
    train_ds,
    validation_data=val_ds,
    epochs=epochs
)

```

Epoch 1/40

```

10/10 [=====] - 1s 61ms/step - loss: 15.8861
- accuracy: 0.3667 - val_loss: 7.9521 - val_accuracy: 0.4533

```

Epoch 2/40

```

10/10 [=====] - 0s 35ms/step - loss: 4.7472 -
accuracy: 0.5067 - val_loss: 1.5687 - val_accuracy: 0.5667

```

Epoch 3/40

```

10/10 [=====] - 0s 32ms/step - loss: 1.0571 -
accuracy: 0.7000 - val_loss: 0.7046 - val_accuracy: 0.7533

```

Epoch 4/40

```

10/10 [=====] - 0s 22ms/step - loss: 0.2787 -
accuracy: 0.9000 - val_loss: 0.5589 - val_accuracy: 0.8000

```

Epoch 5/40

```

10/10 [=====] - 0s 19ms/step - loss: 0.0465 -
accuracy: 1.0000 - val_loss: 0.6583 - val_accuracy: 0.7467

```

Epoch 6/40

```

10/10 [=====] - 0s 17ms/step - loss: 0.0334 -
accuracy: 0.9933 - val_loss: 0.4826 - val_accuracy: 0.8133

```

Epoch 7/40

```

10/10 [=====] - 0s 17ms/step - loss: 0.0210 -

```



In [74]:

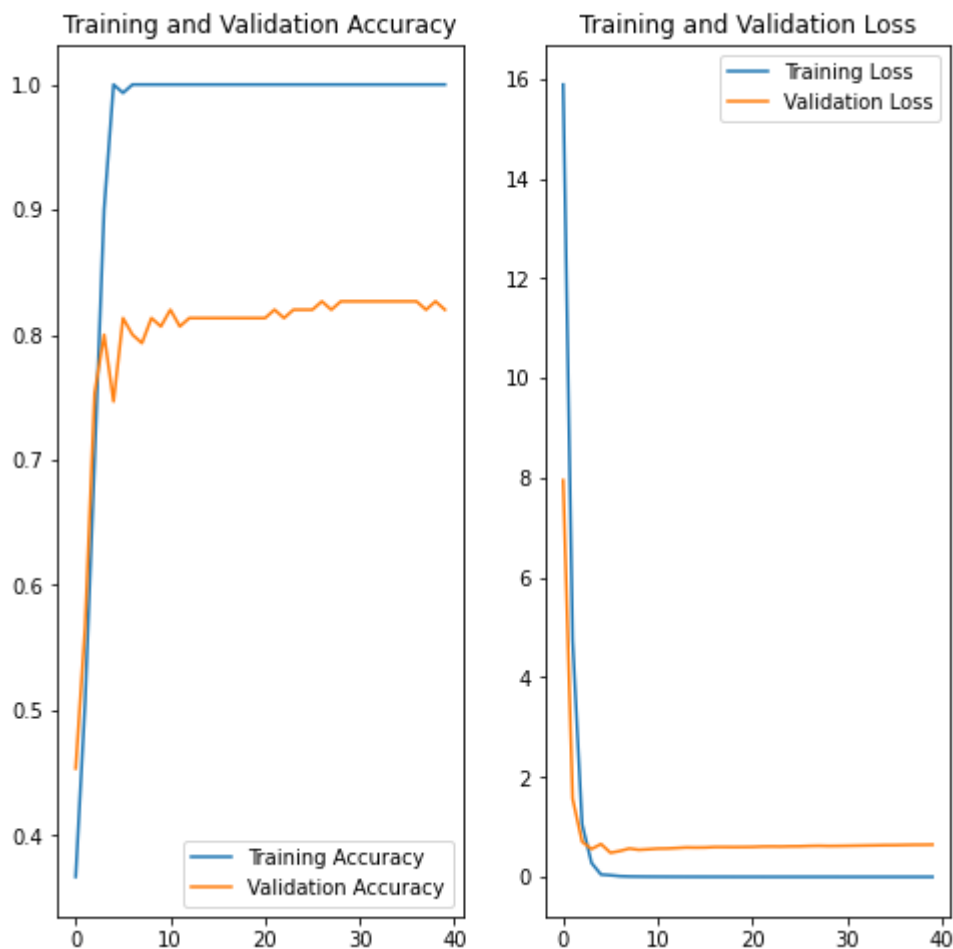
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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()
```



In [75]:

```
data_augmentation_train = ImageDataGenerator(
    rescale=1./255,
    rotation_range= 90,
    width_shift_range=0.2,
    height_shift_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,
    vertical_flip=True)

data_augmentation_val = ImageDataGenerator(
    rescale=1./255)

train = data_augmentation_train.flow_from_directory(
    data_root,
    target_size=(img_height, img_width),
    batch_size=BATCH_SIZE,
    )
valid = data_augmentation_val.flow_from_directory(
    data_root,
    target_size=(img_height, img_width),
    batch_size=BATCH_SIZE)

for data_batch, labels_batch in train:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break

for data_batch, labels_batch in train_ds:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
```

```
Found 300 images belonging to 3 classes.
Found 300 images belonging to 3 classes.
data batch shape: (16, 192, 192, 3)
labels batch shape: (16, 3)
data batch shape: (16, 192, 192, 3)
labels batch shape: (16,)
```



In [76]:

```

model = Sequential([
    # layers.Rescaling(1./255),
    layers.Conv2D(32, 3, padding='same', activation='relu', input_shape=(img_height, i
    layers.Dropout(0.2),
    layers.MaxPooling2D(),
    layers.Conv2D(64, 3, padding='same', activation='relu'),
    layers.Dropout(0.2),
    layers.MaxPooling2D(),
    layers.Conv2D(128, 3, padding='same', activation='relu'),
    layers.Dropout(0.2),
    layers.MaxPooling2D(),
    layers.Flatten(),
    # layers.Dense(32, activation='relu'),
    layers.Dense(3, activation = 'sigmoid')
])

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

```

In [77]:

```

epochs=40
history = model.fit(
    train,
    validation_data=valid,
    epochs=epochs
)

```

```

Epoch 1/40
19/19 [=====] - 6s 285ms/step - loss: 1.3501
- accuracy: 0.3733 - val_loss: 1.0880 - val_accuracy: 0.4533
Epoch 2/40
19/19 [=====] - 5s 270ms/step - loss: 1.1066
- accuracy: 0.3300 - val_loss: 1.0895 - val_accuracy: 0.3867
Epoch 3/40
19/19 [=====] - 5s 265ms/step - loss: 1.0643
- accuracy: 0.4433 - val_loss: 1.0696 - val_accuracy: 0.3767
Epoch 4/40
19/19 [=====] - 5s 265ms/step - loss: 1.0189
- accuracy: 0.4767 - val_loss: 1.0021 - val_accuracy: 0.5600
Epoch 5/40
19/19 [=====] - 6s 330ms/step - loss: 0.9815
- accuracy: 0.4900 - val_loss: 0.9492 - val_accuracy: 0.4667
Epoch 6/40
19/19 [=====] - 5s 263ms/step - loss: 1.0109
- accuracy: 0.5267 - val_loss: 0.9601 - val_accuracy: 0.6033
Epoch 7/40
19/19 [=====] - 5s 260ms/step - loss: 0.9462
- accuracy: 0.5400 - val_loss: 0.9492 - val_accuracy: 0.6467

```



In [78]:

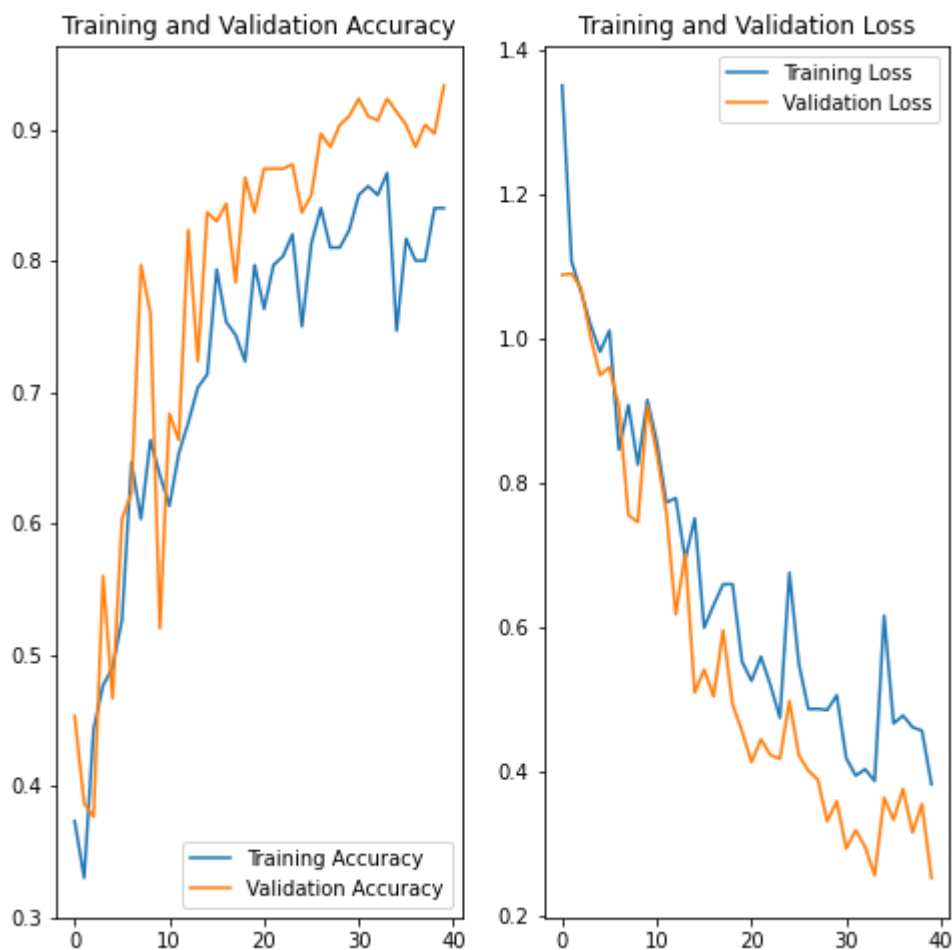
```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

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()
```



In [79]:

```
# model.save("/content/my_model_columbia.h5")
# files.download("/content/my_model_columbia.h5")
```

In []:

As we can see the augmented data works better than the previous one (improved around 10% of accuracy). So it is effective in this case.

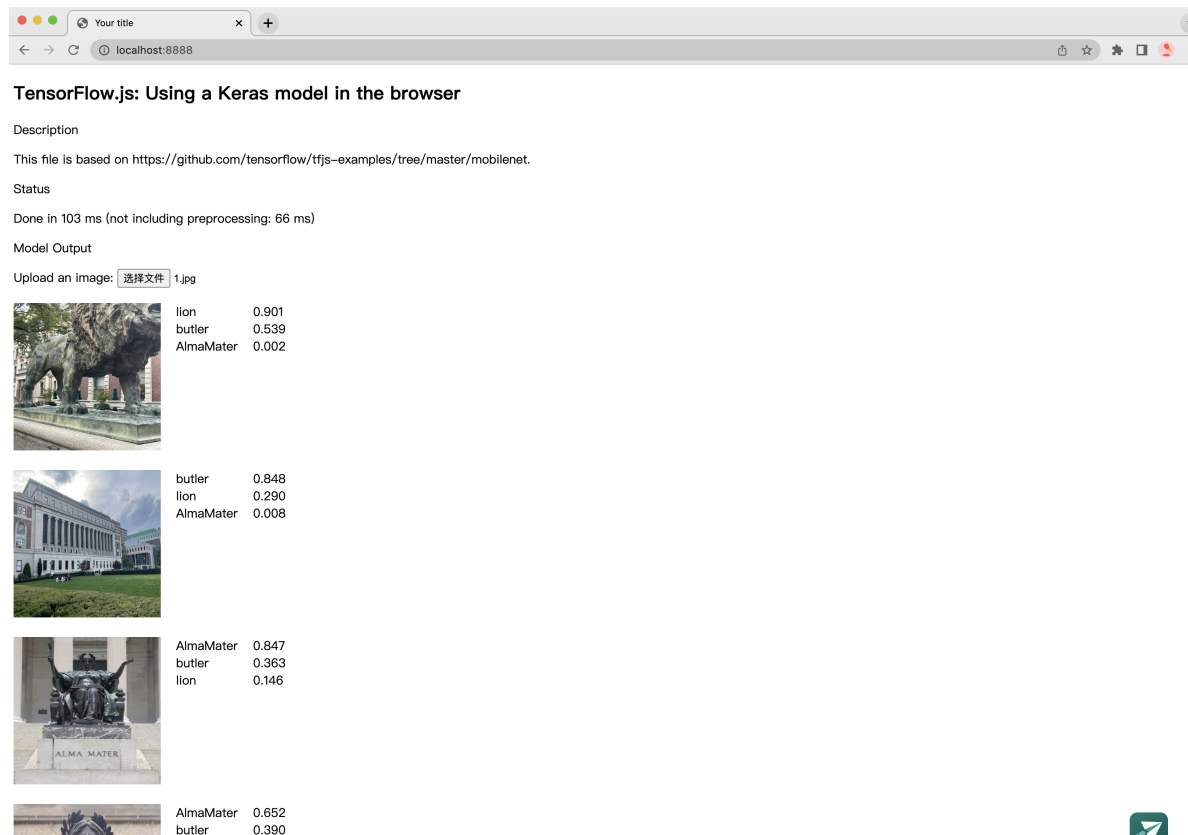
2b) Run your model in the browser

Save and download your model, and head to notebook two again. You may not need to make many changes to the starter code, but pay attention to the preprocessing, number of output classes, image size, etc. Take a screenshot of your model running in a webpage and include it with your submission.

In [81]:

```
from IPython.display import Image
Image('/content/screenshot.png')
```

Out[81]:



TensorFlow.js: Using a Keras model in the browser

Description



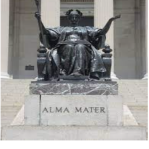

This file is based on <https://github.com/tensorflow/tfjs-examples/tree/master/mobilenet>.

Status

Done in 103 ms (not including preprocessing: 66 ms)

Model Output

Upload an image: 1.jpg

	lion: 0.901 butler: 0.539 AlmaMater: 0.002
	butler: 0.848 lion: 0.290 AlmaMater: 0.008
	AlmaMater: 0.847 butler: 0.363 lion: 0.146
	AlmaMater: 0.652 butler: 0.390

In []:



