Dog-Vision

October 1, 2020

1 End-to-end Multi-class Dog Breed Classification

This notebook builds and end-to-end multi-class image classifier using TensorFlow 2.0 and TensorFlow Hub.

1.1 1. Problem

Identifying the breed of a dog given an image of a dog.

When I'm sitting at the cafe and I take a photo of a dog, I want to know what breed of dog it is.

1.2 2. Data

The data we're using is from kaggle's dog breed identification competition.

https://www.kaggle.com/c/dog-breed-identification/data

1.3 3. Evaluation

The evaluation is a file with the prediction probabilities for each dog breed of each test image.

https://www.kaggle.com/c/dog-breed-identification/overview/evaluation

1.4 4. Features

Some information about the data: * We're dealing with images (unstructured data) so it's probably best we use deep learning/transfer learning * There are 120 breeds of dogs (This means 120 different classes) * Each image has a filename that is its unique id * There are around 10,000+ images in the training set (these images have labels). * There are around 10,000+ images in the test set (these images have no labels, because we'll want to predict them).

[ ]:
```
# Unzip the uploaded data into Google Drive
#!unzip "drive/My Drive/Dog Vision/dog-breed-identification.zip" -d "drive/My␣
 ↪Drive/Dog Vision"
```

1.4.1 Get our workspace ready

- Import TensorFlow 2.2.0
- Import TensorFlow Hub
- Make sure we're using GPU

[1]:
```
# Import necessary tools
import tensorflow as tf
import tensorflow_hub as hub
print("TF version: ", tf.__version__)
print("TF Hub version: ", hub.__version__)

# Check for GPU availability
print("GPU", "available (YESS!!!!!!!!)" if tf.config.
 ↪list_physical_devices("GPU") else "Not available :(")
```

```
TF version:  2.2.0
TF Hub version:  0.8.0
GPU available (YESS!!!!!!!!)
```

1.5 Getting our data ready (turning into Tensors)

With all machine learning models, our data has to be in numerical format. So that's what we'll be doing first. Turning our images into Tensors (numerical representations).

[4]:
```
# Check the labels of our data
import pandas as pd
labels_csv = pd.read_csv("drive/My Drive/Dog Vision/labels.csv")
print(labels_csv.describe())
print(labels_csv.head())
```

```
                                      id               breed
count                              10222               10222
unique                             10222                 120
top     f8d48f89aaa55962d4beb853a128eac7  scottish_deerhound
freq                                   1                 126
                                 id             breed
0  000bec180eb18c7604dcecc8fe0dba07       boston_bull
1  001513dfcb2ffafc82cccf4d8bbaba97             dingo
2  001cdf01b096e06d78e9e5112d419397          pekinese
3  00214f311d5d2247d5dfe4fe24b2303d          bluetick
4  0021f9ceb3235effd7fcde7f7538ed62  golden_retriever
```

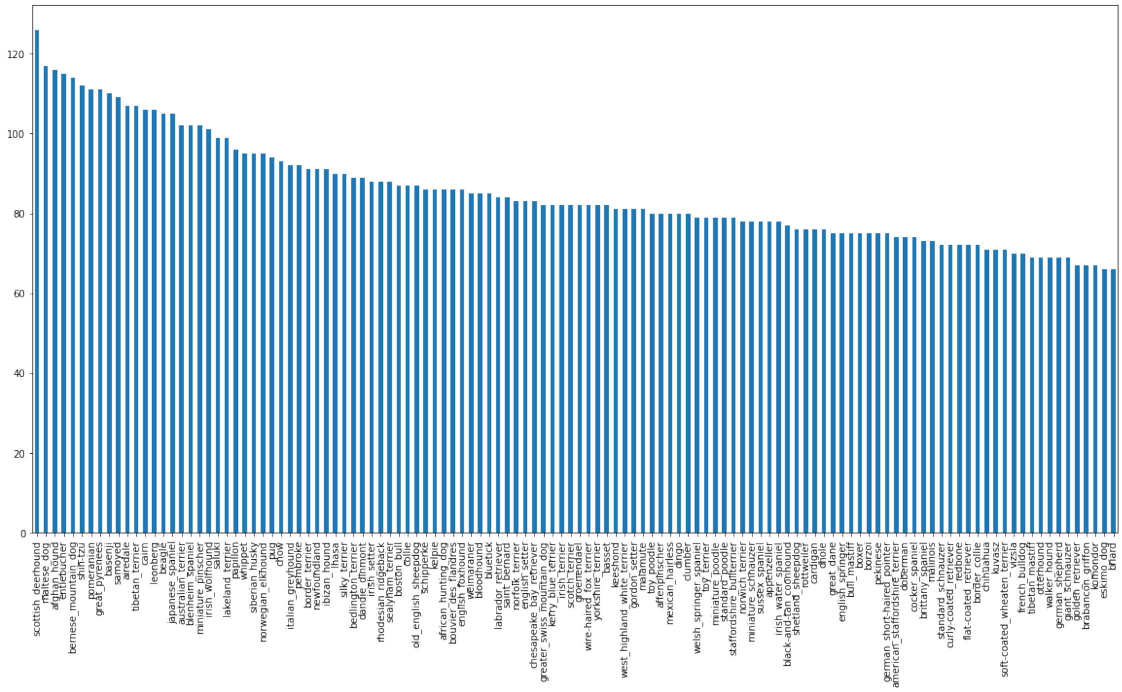
[ ]:
```
labels_csv.head()
```

[ ]:

	id	breed
0	000bec180eb18c7604dcecc8fe0dba07	boston_bull
1	001513dfcb2ffafc82cccf4d8bbaba97	dingo
2	001cdf01b096e06d78e9e5112d419397	pekinese
3	00214f311d5d2247d5dfe4fe24b2303d	bluetick
4	0021f9ceb3235effd7fcde7f7538ed62	golden_retriever

[ ]:
```
labels_csv["breed"].value_counts().plot.bar(figsize=(20, 10))
```

[ ]: <matplotlib.axes._subplots.AxesSubplot at 0x7f8de005fac8>



[ ]:
```
labels_csv.breed.value_counts().median()
```

[ ]: 82.0

[5]:
```
# Let's view as image
from IPython.display import Image
Image("drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg")
```

[5]:

1.5.1 Getting images and their labels

Let's get a list of all our image file pathnames.

[6]:
```
# Create pathnames from image ID's
filenames = ["drive/My Drive/Dog Vision/train/" + fname + ".jpg" for fname in␣
↪labels_csv["id"]]

# Check first 10
filenames[:10]
```

[6]:
```
['drive/My Drive/Dog Vision/train/000bec180eb18c7604dcecc8fe0dba07.jpg',
 'drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg',
 'drive/My Drive/Dog Vision/train/001cdf01b096e06d78e9e5112d419397.jpg',
 'drive/My Drive/Dog Vision/train/00214f311d5d2247d5dfe4fe24b2303d.jpg',
 'drive/My Drive/Dog Vision/train/0021f9ceb3235effd7fcde7f7538ed62.jpg',
 'drive/My Drive/Dog Vision/train/002211c81b498ef88e1b40b9abf84e1d.jpg',
 'drive/My Drive/Dog Vision/train/00290d3e1fdd27226ba27a8ce248ce85.jpg',
 'drive/My Drive/Dog Vision/train/002a283a315af96eaea0e28e7163b21b.jpg',
 'drive/My Drive/Dog Vision/train/003df8b8a8b05244b1d920bb6cf451f9.jpg',
 'drive/My Drive/Dog Vision/train/0042188c895a2f14ef64a918ed9c7b64.jpg']
```

[7]:
```python
# Check whether number of filenames matches number of actual image files
import os
if len(os.listdir("drive/My Drive/Dog Vision/train/")) == len(filenames):
  print("Filenames match actual amount of files!!! Proceed.")
else:
  print("Filenames do no match actual amount of files, check the target␣
 ↪dictionary.")
```

```
Filenames match actual amount of files!!! Proceed.
```

[ ]:
```python
# One more check
Image(filenames[9000])
```

[ ]:

[ ]:
```
labels_csv["breed"][9000]
```

[ ]: 'tibetan_mastiff'

Since we've now got our training image filepaths in a list, let's prepare our labels.

[8]:
```
import numpy as np
labels = labels_csv["breed"].to_numpy()
# np.array(labels)
(labels)
```

[8]:
```
array(['boston_bull', 'dingo', 'pekinese', …, 'airedale',
       'miniature_pinscher', 'chesapeake_bay_retriever'], dtype=object)
```

[ ]:
```
# See if number of labels matches the number of filenames
if len(labels) == len(filenames):
  print("Number of labels matches number of filenames!")
else:
  print("Number of labels does not match number of filenames, check data␣
↪directories!")
```

Number of labels matches number of filenames!

[9]:
```
# Find the unique label values
unique_breeds = np.unique(labels)
len(unique_breeds)
```

[9]: 120

[ ]:
```
# Turn a single label into an array of booleans
print(labels[0])
labels[0] == unique_breeds
```

boston_bull

[ ]:
```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False,  True, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False, False, False, False,
```

```
        False, False, False, False, False, False, False, False, False,
        False, False, False])
```

[10]:
```
# Turn every label into a boolean array
boolean_labels = [label == unique_breeds for label in labels]
boolean_labels[:2]
```

[10]:
```
[array([False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False,  True, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False]),
 array([False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False,  True, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False])]
```

[ ]:
```
len(boolean_labels)
```

[ ]:
```
10222
```

[ ]:
```
# Example: Turning boolean array into integers
print(labels[0]) # Original label
print(np.where(unique_breeds == labels[0]))  # index where label occurs
print(boolean_labels[0].argmax()) # index where label occurs in boolean array
print(boolean_labels[0].astype(int)) # Converting bool into int
```

```
boston_bull
```

```
(array([19]),)
19
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0]
```

Since we've got our training image file paths in a list, let's prepare our labels.

1.5.2 Creating our own validation set

Since the dataset from kaggle doesn't come with a validation set, we're going to create our own.

[11]:
```python
# Setup X & y variables
X = filenames
y = boolean_labels
```

We're going to start off experimenting with ~1000 images and increase as needed.

[14]:
```python
# Set number of images to use for expermenting
NUM_IMAGES = 1000 #@param {type: "slider", min:1000, max:10000, step:1000}
```

[15]:
```python
# Let's split our data into train and validation sets
from sklearn.model_selection import train_test_split

# Split them into training and validation of total size NUM_IMAGES
X_train, X_val, y_train, y_val = train_test_split(X[:NUM_IMAGES],
                                                  y[:NUM_IMAGES],
                                                  test_size=0.2,
                                                  random_state=42)

len(X_train), len(X_val), len(y_train), len(y_val)
```

[15]:
```
(800, 200, 800, 200)
```

[ ]:
```python
X_train[:10], y_train[:2]
```

[ ]:
```
(['drive/My Drive/Dog Vision/train/00bee065dcec471f26394855c5c2f3de.jpg',
  'drive/My Drive/Dog Vision/train/0d2f9e12a2611d911d91a339074c8154.jpg',
  'drive/My Drive/Dog Vision/train/1108e48ce3e2d7d7fb527ae6e40ab486.jpg',
  'drive/My Drive/Dog Vision/train/0dc3196b4213a2733d7f4bdcd41699d3.jpg',
  'drive/My Drive/Dog Vision/train/146fbfac6b5b1f0de83a5d0c1b473377.jpg',
  'drive/My Drive/Dog Vision/train/0ea5759640f2e1c2d1a06adaf8a54ca7.jpg',
  'drive/My Drive/Dog Vision/train/03e1d2ee5fd90aef036c90a9e7f81177.jpg',
  'drive/My Drive/Dog Vision/train/16941a6728ddb9cb7423a6cc97fbe071.jpg',
  'drive/My Drive/Dog Vision/train/0bedbecd92390ef9f4f7c8b06a629340.jpg',
  'drive/My Drive/Dog Vision/train/143b9484273e57668d03bfc26755810a.jpg'],
```

```
[array([False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False,  True,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False]),
 array([False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False,  True, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False, False, False, False, False, False, False,
        False, False, False])])
```

1.6 Preprocessing Images (turning images into Tensors)

1. Take an image filepath as input
2. Use TensorFlow to read the file and save it to a variable, image
3. Turn our image (a jpg) into Tensors
4. Normalize our image (convert color channel values from 0-225 to 0-1)
5. Resize the image to be a shape of (224, 224)
6. Return the modified image

Before we do, let's see what importing an image looks like.

[16]:
```
# Convert image into NumPy array
from matplotlib.pyplot import imread
image = imread(filenames[42])
image.shape
```

[16]: (257, 350, 3)

[ ]:
```
image.max(), image.min(), type(image)
```

[ ]: (255, 0, numpy.ndarray)

[ ]:
```
image[:2]
```

[ ]:
```
array([[[ 89, 137,  87],
        [ 76, 124,  74],
        [ 63, 111,  59],
        …,
        [ 76, 134,  86],
        [ 76, 134,  86],
        [ 76, 134,  86]],

       [[ 72, 119,  73],
        [ 67, 114,  68],
        [ 63, 111,  63],
        …,
        [ 75, 131,  84],
        [ 74, 132,  84],
        [ 74, 131,  86]]], dtype=uint8)
```

[ ]:
```
# Turn image into a tensor
tf.constant(image)[:2]
```

[ ]:
```
<tf.Tensor: shape=(2, 350, 3), dtype=uint8, numpy=
array([[[ 89, 137,  87],
        [ 76, 124,  74],
        [ 63, 111,  59],
        …,
        [ 76, 134,  86],
        [ 76, 134,  86],
        [ 76, 134,  86]],

       [[ 72, 119,  73],
        [ 67, 114,  68],
        [ 63, 111,  63],
        …,
        [ 75, 131,  84],
        [ 74, 132,  84],
        [ 74, 131,  86]]], dtype=uint8)>
```

Now we've seen what an image looks like as a Tensor, let's make a function to preprocess them. 1. Take an image filepath as input 2. Use TensorFlow to read the file and save it to a variable, image 3. Turn our image (a jpg) into Tensors 4. Normalize our image (convert color channel values from 0-225 to 0-1) 5. Resize the image to be a shape of (224, 224) 6. Return the modified image

More information on loading images in TensorFlow can be seen here: https://www.tensorflow.org/tutorials/load_data/images

```
[17]: # Define image size
IMG_SIZE = 224

# Create a function for preprocessing images
def process_image(image_path, img_size=IMG_SIZE):
  """
  Take an image file path and turns the image into a Tensor.
  """
  # Read in an image file
  image = tf.io.read_file(image_path) # returns a tensor type object/string,␣
↪here reads the image from the path
  # Turn the jpeg image into numerical Tensor with 3 color channels (Red,␣
↪Green, Blue)
  image = tf.image.decode_jpeg(image, channels = 3) # returns a np.array
  # Convert the colour channel values from 0-225 to 0-1 values
  image = tf.image.convert_image_dtype(image, tf.float32) # scales & changes␣
↪dtype
  # Resize the image to our desired values (224, 224)
  image = tf.image.resize(image, size = [IMG_SIZE, IMG_SIZE])

  return image
```

1.6.1 Turning our data into batches

Why turn our data into batches?

Let's say you're trying to process 10,000+ images in one go… they all might not fit into memory.

So that's why we do about 32 (this is the batch size) images at a time (you can manually adjust the batch size if need be).

In order to use TensorFlow effectively, we need our data in the form of Tensor tuples which look like this: (image, label).

```
[18]: # Create a simple function to return a tuple (image, label)
def get_image_label(image_path, label):
  """
  Takes an image file path name and the associated label,
  processes the image and returns a tuple of (image, label)
  """
  image = process_image(image_path)
  return image, label
```

```
[ ]: # Demo of get_image_label function
(process_image(X[32])[:2], tf.constant(y[32])[:2])
```

```
[ ]: (<tf.Tensor: shape=(2, 224, 3), dtype=float32, numpy=
 array([[[2.1821149e-02, 9.6624352e-02, 3.9579429e-02],
         [3.1774971e-01, 3.9849764e-01, 3.4007537e-01],
         [1.9432701e-02, 7.9156667e-02, 2.5785400e-02],
         …,
         [0.0000000e+00, 2.8685225e-02, 7.3815696e-03],
         [1.9170513e-03, 4.3444425e-02, 1.0881329e-02],
         [1.4592139e-02, 6.1650965e-02, 2.2435278e-02]],

        [[1.0875124e-01, 1.8718261e-01, 1.4012378e-01],
         [1.7272735e-04, 4.8262980e-02, 1.3897762e-03],
         [6.4935260e-02, 1.2673499e-01, 8.3912849e-02],
         …,
         [1.0160362e-04, 1.2352975e-02, 9.8086741e-05],
         [1.2390460e-02, 5.1494777e-02, 2.0159349e-02],
         [2.1063086e-02, 6.0197882e-02, 2.8852297e-02]]], dtype=float32)>,
 <tf.Tensor: shape=(2,), dtype=bool, numpy=array([False, False])>)
```

Now we've got a way to turn our data into typles of Tensors in the form: (image, label), let's make a function to turn all of our data (X & y) into batches!

```
[19]: # Define the batch size, 32 is a good start
BATCH_SIZE = 32

# Create a function to turn data into batches
def create_data_batches(X, y=None, batch_size=BATCH_SIZE, valid_data=False,␣
 ↪test_data=False):
  """
  Creates batches of data out of image (X) and label (y) pairs.
  Shuffles the data if it's training data but doesn't shuffle if it's␣
 ↪validation data.
  Also accepts test data as input (no labels).
  """
  # If the data is a test dataset, we probably don't have labels
  if test_data:
    print("Creating test data batches....")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X))) # only␣
 ↪filepaths (no labels)
    data_batch = data.map(process_image).batch(BATCH_SIZE)
    return data_batch

  # If the data is a valid dataset, we don't need to shuffle it
  elif valid_data:
    print("Creating validation data batches.......")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X),   # filepaths
                                               tf.constant(y)))  # labels
    data_batch = data.map(get_image_label).batch(BATCH_SIZE)
```

```
    return data_batch

  else:
    print("Creating training data batches........")
    # Turn filepaths and labels into Tensors
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X),
                                                tf.constant(y)))
    # Shuffling pathnames and labels before mapping image processor function is␣
↪faster than shuffling images
    data = data.shuffle(buffer_size = len(X))

    # Create (image, label) tuples (this also turns the image path into a␣
↪preprocessed image)
    data = data.map(get_image_label)

    # Turn the training data into batches
    data_batch = data.batch(BATCH_SIZE)

    return data_batch
```

[20]:
```
# Create training and validation data batches
train_data = create_data_batches(X_train, y_train)
val_data = create_data_batches(X_val, y_val, valid_data=True)
```

```
Creating training data batches…
Creating validation data batches…
```

[ ]:
```
# Check out the different attributes of our data batches
train_data.element_spec, val_data.element_spec
```

[ ]:
```
((TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
  TensorSpec(shape=(None, 120), dtype=tf.bool, name=None)),
 (TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None),
  TensorSpec(shape=(None, 120), dtype=tf.bool, name=None)))
```

[ ]:
```
# Testing to create data batches
# list(tf.data.Dataset.from_tensor_slices((tf.constant(X[:10]), tf.constant(y[:
↪10]))).map(get_image_label).batch(32))
```

1.7 Let's Visualize our data batches

Our data is now in batches, however, its little hard to show/comprehend, let's visualize !

[21]:
```
import matplotlib.pyplot as plt
# Create a function to view images in a data batch
def show_25_images(images, labels):
```

```
    """
    Display a plot of 25 images and their labels from the data batch.
    """
    # Setup the figure
    plt.figure(figsize=(10, 10))
    # Loop through 25 (for displaying 25 images)
    for i in range(25):
      # Create subplots 5 rows, 5 columns
      ax = plt.subplot(5, 5, i+1)
      # Display an image
      plt.imshow(images[i])
      # Add the image label as the title
      plt.title(unique_breeds[labels[i].argmax()])
      # Turn the grid lines off
      plt.axis("off")
```

[22]:
```
# Unbatching the images data and visualizing the data
train_images, train_labels = next(train_data.as_numpy_iterator())
show_25_images(train_images, train_labels)
```

[ ]:
```
chows = ["drive/My Drive/Dog Vision/train/" + id + ".jpg" for id in␣
↪labels_csv[labels_csv["breed"]=="chow"]["id"]]
len(chows)
```

[ ]: 93

[23]:
```
# Unbatching the validation data and visualizing it
val_images, val_labels = next(val_data.as_numpy_iterator())
show_25_images(val_images, val_labels)
```

1.8 Buliding a model

Before building a model here are few things we need to define first: * The input shape (our images shape, in the form of Tensors) to our model. * The labels shape (images labels, in the form of Tensors) to our model. * The URL of the model we want to use from TensorFlow Hub - https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4

[24]:
```
# Setup input shape to the model
INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3] # batch, height, width, colour channels

# Setup output shape of our model
OUTPUT_SHAPE = len(unique_breeds)
```

```
# Setup the model URL from TensorFlow Hub
MODEL_URL = "https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/
↪classification/4"
```

Now we've got our inputs, outputs and model ready to go. Let's put them together into keras deep learning model!

Knowing this let's create a function: * Takes the input shape, output shape and the model we've chosen as parameters. * Defines the layers in a Keras model in sequential fashion(do this first, then this, then that). * Compiles the model (says it should be evaluated and improved). * Builds the model (tells the model the input shape it'll be getting). * Returns the model.

All of the steps can be found here: https://www.tensorflow.org/guide/keras/sequential_model

[25]:
```
# Create a function which builds a Keras model
def create_model(input_shape=INPUT_SHAPE, output_shape=OUTPUT_SHAPE,␣
↪model_url=MODEL_URL):
  print("Building model with:", MODEL_URL)

  # Setup the model layers
  model = tf.keras.Sequential([                     # Creating a keras model
                  hub.KerasLayer(MODEL_URL),  # Layer 1 (input layer)
                  tf.keras.layers.Dense(units=OUTPUT_SHAPE,
                                        activation="softmax")  # Layer 2␣
↪(output layer)
  ])

  # Compile the model
  model.compile(
      loss=tf.keras.losses.CategoricalCrossentropy(),
      optimizer=tf.keras.optimizers.Adam(),
      metrics=["accuracy"]
  )

  # Build the model
  model.build(INPUT_SHAPE)

  return model
```

[ ]:
```
model = create_model()
model.summary()
```

```
Building model with:
https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Model: "sequential"

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

```
=================================================================
keras_layer (KerasLayer)     multiple                  5432713
_________________________________________________________________
dense (Dense)                multiple                  120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

1.9 Creating callbacks

Callbacks are helper functions a model can use during training to do such things as save its progress, check its progress or stop training early if a model stops improving.

We'll create two callbacks, one for TensorBoard which helps track our models progress and another for early stopping which prevents our model from training for too long. ### TensorBoard Callback To setup a TensorBoard callback, we need to do 3 things: 1. Load the TensorBoard notebook extension. 2. Create a TensorBoard callback which is able to save logs to a directory and pass it to our models fit() function. 3. Visualize our models training logs with the %tensorboard magic function (we'll do this after model training).

[27]:
```
# Load TensorBoard notebook extension
%load_ext tensorboard
```

[26]:
```
import datetime

# Create a function to build a TensorBoard callback
def create_tensorboard_callback():
  # Create a log directory for storing TensorBoard logs
  logdir = os.path.join("drive/My Drive/Dog Vision/logs",
                        # Make it so the logs get tracked whenever we run an
                        # experiment
                        datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
  return tf.keras.callbacks.TensorBoard(logdir)
```

1.10 Early Stopping Callback

Early stopping helps stop our model from overfitting by stopping training if a certain evaluation metric stops improving. https://www.tensorflow.org/api_docs/python/tf/keras

[25]:
```
# Create early stopping callback
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_accuracy",
                                                  patience=3)
```

1.11 Training a model (on subset of data)

Our first model is only going to train on 1000 images, to make sure everything is working.

[26]:
```
NUM_EPOCHS =  100 #@param {type:"slider", min:10, max:100, step:10}
```

[ ]:
```
# Checking GPU again
print("GPU:", "available!!" if tf.config.list_physical_devices("GPU") else "not␣
↪available :(")
```

```
GPU: available!!
```

Let's create a function which trains a model. * Create a model using create_model() * Setup a TensorBoard callback using create_tensorboard_callback() * Call the fit() function on our model passing it the training data, validation data, number of epochs to train for (NUM_EPOCHS) and the callbacks we'd like to use * Return the model

[28]:
```
# Build a function to train and return a trained model
def train_model():
    """
    Trains a given model and returns the trained version.
    """
    # Create a model
    model = create_model()

    # Create a new TensorBoard session everytime we train a model
    tensorboard = create_tensorboard_callback()

    # Fit the model to the data passing it the callbacks we created
    model.fit(x=train_data,  # Batches of data with (images, labels)
              epochs=NUM_EPOCHS,
              validation_data=val_data,
              validation_freq=1,
              callbacks=[tensorboard, early_stopping])

    # Return the fitted model
    return model
```

[ ]:
```
# Fit the model to the data
model = train_model()
```

```
Building model with:
https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Epoch 1/100
25/25 [==============================] - 393s 16s/step - loss: 4.5833 -
accuracy: 0.0838 - val_loss: 3.3170 - val_accuracy: 0.2500
Epoch 2/100
```

```
25/25 [==============================] - 4s 165ms/step - loss: 1.6246 -
accuracy: 0.7113 - val_loss: 2.0720 - val_accuracy: 0.5050
Epoch 3/100
25/25 [==============================] - 4s 167ms/step - loss: 0.5589 -
accuracy: 0.9388 - val_loss: 1.6555 - val_accuracy: 0.6450
Epoch 4/100
25/25 [==============================] - 4s 166ms/step - loss: 0.2569 -
accuracy: 0.9862 - val_loss: 1.4775 - val_accuracy: 0.6550
Epoch 5/100
25/25 [==============================] - 4s 169ms/step - loss: 0.1478 -
accuracy: 0.9950 - val_loss: 1.3985 - val_accuracy: 0.6500
Epoch 6/100
25/25 [==============================] - 4s 173ms/step - loss: 0.1008 -
accuracy: 0.9987 - val_loss: 1.3537 - val_accuracy: 0.6700
Epoch 7/100
25/25 [==============================] - 4s 170ms/step - loss: 0.0756 -
accuracy: 1.0000 - val_loss: 1.3273 - val_accuracy: 0.6750
Epoch 8/100
25/25 [==============================] - 4s 165ms/step - loss: 0.0597 -
accuracy: 1.0000 - val_loss: 1.3000 - val_accuracy: 0.6800
Epoch 9/100
25/25 [==============================] - 4s 165ms/step - loss: 0.0492 -
accuracy: 1.0000 - val_loss: 1.2850 - val_accuracy: 0.6750
Epoch 10/100
25/25 [==============================] - 4s 164ms/step - loss: 0.0415 -
accuracy: 1.0000 - val_loss: 1.2689 - val_accuracy: 0.6650
Epoch 11/100
25/25 [==============================] - 4s 162ms/step - loss: 0.0356 -
accuracy: 1.0000 - val_loss: 1.2544 - val_accuracy: 0.6700
```

Question: It looks like our model is overfitting because it's performing far better on the training dataset than the vaidation dataset, what are some ways to prevent model overfitting in deep learning neural networks?

Note: Overfitting to begin with is a good thing! It means our model is learning!!!

Checking the TensorBoard logs The TensorBoard magic function (%tensorboard) will access the logs directory we created earlier and visualize its contents.

[ ]:
```
%tensorboard --logdir drive/My\ Drive/Dog\ Vision/logs
```

Output hidden; open in https://colab.research.google.com to view.

1.12 Making and evaluating predictions using a trained model

[ ]:
```
val_data
```

[ ]: <BatchDataset shapes: ((None, 224, 224, 3), (None, 120)), types: (tf.float32, tf.bool)>

[ ]:
```python
# Make predictions on the validation data (not used to train on)
predictions = model.predict(val_data, verbose=1)
predictions
```

```
7/7 [==============================] - 1s 107ms/step
```

[ ]:
```
array([[5.96020080e-04, 1.10000423e-04, 3.89112718e-03, …,
         7.14927883e-05, 9.53733906e-05, 5.91167063e-03],
        [4.42847656e-03, 1.29401521e-03, 8.00199136e-02, …,
         7.30087457e-04, 3.08877532e-03, 7.80404444e-05],
        [2.84436810e-05, 1.10064375e-05, 8.62644720e-06, …,
         7.86195596e-06, 1.58558159e-05, 3.71111935e-04],
        …,
        [2.89466061e-06, 6.45891705e-05, 1.27511477e-04, …,
         1.21909989e-05, 2.33309460e-04, 9.19244776e-05],
        [5.41963708e-03, 4.10278881e-04, 2.58074666e-04, …,
         2.93089135e-04, 8.38370470e-05, 3.75716342e-03],
        [1.08708322e-04, 4.05985593e-05, 5.68513176e-04, …,
         1.22044689e-03, 1.54267973e-03, 1.76055357e-04]], dtype=float32)
```

[ ]:
```python
# First prediction
index = 42
print(predictions[index])
print(f"Max value (probability of prediction): {np.max(predictions[index])}")
print(f"Sum: {np.sum(predictions[index])}")
print(f"Max index: {np.argmax(predictions[index])}")
print(f"Predicted label: {unique_breeds[predictions[index].argmax()]}")
```

```
[6.54019677e-05 2.91014621e-05 6.00882086e-05 2.36707856e-05
 1.33853068e-03 2.84487542e-05 4.97295769e-05 3.46933230e-04
 3.95653117e-03 2.33118199e-02 2.88168376e-05 1.96835063e-05
 2.93711346e-04 2.64067110e-03 1.05843891e-03 8.96163809e-04
 2.99201947e-05 5.57112740e-04 1.59982825e-04 3.74111405e-04
 8.03620969e-06 3.69936170e-04 6.29026981e-05 1.36227336e-05
 5.65784844e-03 5.98674742e-05 8.86038688e-05 1.24138358e-04
 2.86689814e-04 5.36176958e-05 1.77834896e-04 1.35289534e-04
 1.24454979e-04 6.38167185e-05 5.60622357e-05 1.78006358e-05
 3.89409470e-05 9.62642007e-05 6.28956768e-05 3.70254368e-01
 1.23924969e-04 9.45436932e-06 3.34318005e-03 5.83183828e-06
 5.30544166e-05 3.31340852e-05 3.28770184e-05 4.18092415e-04
 1.47250803e-05 2.72123580e-04 1.00478836e-04 3.13261640e-04
 8.23333248e-05 1.29575829e-03 1.89105587e-04 3.87985288e-04
 1.21712575e-04 2.59372682e-05 8.38025007e-05 2.45857809e-05
 2.38150660e-05 1.95547516e-04 1.19637352e-05 5.80485139e-05
 2.95700611e-05 1.25892460e-04 1.29156848e-04 3.64421256e-04
```

```
 1.16065756e-04 9.44906969e-06 4.14229435e-05 8.70555814e-05
 1.54565842e-05 2.45425035e-04 3.96104624e-05 2.85494403e-04
 2.57657899e-04 9.23837160e-06 6.81466772e-05 2.73866084e-04
 1.48293238e-05 2.35594653e-05 6.73338654e-05 1.00657798e-03
 1.83795215e-04 7.77120295e-05 5.16850851e-04 3.09584834e-06
 4.48588107e-05 1.30798889e-03 1.90537437e-04 7.74064392e-06
 2.05309293e-03 2.01036251e-04 1.05184636e-05 2.02377210e-04
 8.00932321e-06 4.26355837e-05 2.25068761e-05 1.48559571e-04
 1.00886813e-04 5.92726719e-05 1.54239475e-04 3.17567392e-05
 4.34035428e-05 3.57699864e-05 4.46249433e-05 5.33133698e-06
 8.08046461e-05 1.54136447e-03 2.22316856e-04 4.54666151e-04
 2.09582489e-04 5.66058218e-01 1.72258166e-04 3.37481004e-04
 7.79146212e-05 6.28818889e-05 1.64948311e-03 1.83613549e-04]
Max value (probability of prediction): 0.5660582184791565
Sum: 1.0000001192092896
Max index: 113
Predicted label: walker_hound
```

[ ]:
```
unique_breeds[113]
```

[ ]:
```
'walker_hound'
```

Having the above functionality is great but we want to be able to do it at scale.

And it would be even better if we could see the image the prediction is been made on!

Note: Prediction probabilities are also known as confidence levels.

[29]:
```
# Turn prediction probabilities into their respective label (easier to␣
 ↪understand)
def get_pred_label(prediction_probabilities):
  """
  Turns an array of prediction probabilities into label.
  """
  return unique_breeds[prediction_probabilities.argmax()]

# Get a predicted label based on an array of prediction probabilities
#pred_label = get_pred_label(predictions[0])
#pred_label
```

Now since our validation data is still in a batch dataset, we'll have to unbatchify to make predictions on the validation images and then compare those predictions to the validation labels (truth labels).

[30]:
```
# Create a function to unbatch a batch dataset
def unbatchify(data):
  """
```

```
  Takes a batched dataset of (image, label) Tensors and returns separate arrays␣
↪of images and labels.
  """
  images_ = []
  labels_ = []
  # Loop through unbatched data
  for image, label in data.unbatch().as_numpy_iterator():
    images_.append(image)
    labels_.append(get_pred_label(label))

  return images_, labels_
```

[ ]:
```
# Unbatchify the validation data
val_images, val_labels = unbatchify(val_data)
val_images[0], val_labels[0]
```

[ ]:
```
(array([[[0.29599646, 0.43284872, 0.3056691 ],
         [0.26635826, 0.32996926, 0.22846507],
         [0.31428418, 0.2770141 , 0.22934894],
         …,
         [0.77614343, 0.82320225, 0.8101595 ],
         [0.81291157, 0.8285351 , 0.8406944 ],
         [0.8209297 , 0.8263737 , 0.8423668 ]],

        [[0.2344871 , 0.31603682, 0.19543913],
         [0.3414841 , 0.36560842, 0.27241898],
         [0.45016077, 0.40117094, 0.33964607],
         …,
         [0.7663987 , 0.8134138 , 0.81350833],
         [0.7304248 , 0.75012016, 0.76590735],
         [0.74518913, 0.76002574, 0.7830809 ]],

        [[0.30157745, 0.3082587 , 0.21018331],
         [0.2905954 , 0.27066195, 0.18401104],
         [0.4138316 , 0.36170745, 0.2964005 ],
         …,
         [0.79871625, 0.8418535 , 0.8606443 ],
         [0.7957738 , 0.82859945, 0.8605655 ],
         [0.75181633, 0.77904975, 0.8155256 ]],

        …,

        [[0.9746779 , 0.9878955 , 0.9342279 ],
         [0.99153054, 0.99772066, 0.9427856 ],
         [0.98925114, 0.9792082 , 0.9137934 ],
         …,
         [0.0987601 , 0.0987601 , 0.0987601 ],
```

```
        [0.05703771, 0.05703771, 0.05703771],
        [0.03600177, 0.03600177, 0.03600177]],

       [[0.98197854, 0.9820659 , 0.9379411 ],
        [0.9811992 , 0.97015417, 0.9125648 ],
        [0.9722316 , 0.93666023, 0.8697186 ],
        …,
        [0.09682598, 0.09682598, 0.09682598],
        [0.07196062, 0.07196062, 0.07196062],
        [0.0361607 , 0.0361607 , 0.0361607 ]],

       [[0.97279435, 0.9545954 , 0.92389745],
        [0.963602  , 0.93199134, 0.88407487],
        [0.9627158 , 0.9125331 , 0.8460338 ],
        …,
        [0.08394483, 0.08394483, 0.08394483],
        [0.0886985 , 0.0886985 , 0.0886985 ],
        [0.04514172, 0.04514172, 0.04514172]]], dtype=float32), 'cairn')
```

[ ]:
```
get_pred_label(labels_[9])
```

[ ]:
```
'border_collie'
```

Now we've got ways to get: * Prediction labels * Validation labels (truth labels) * Validation images

Let's make some function to make these all a bit more visualize.

We'll create a function which: * Takes an array of prediction probabilities, an array of truth labels and an array of images and an integer. * Convert the prediction probabilities to a predicted label. * Plot the predicted label, its predicted probability, the truth label and the target image on a single plot.

[31]:
```
def plot_pred(prediction_probabilities, labels, images, n=1):
  """
  View the prediction, ground truth and image for sample n
  """
  pred_prob, true_label, image = prediction_probabilities[n], labels[n],␣
↪images[n]

  # Get the pred label
  pred_label = get_pred_label(pred_prob)

  # Plot image & remove ticks
  plt.imshow(image)
  plt.xticks([])
  plt.yticks([])
```

```
    # Change the colour of the title depending on if the prediction is right or␣
 ↪wrong
    if pred_label == true_label:
      color = "green"
    else:
      color = "red"

    # Change plot title to be predicted, probability of prediction and truth label
    plt.title("{} {:2.0f}% {}".format(pred_label,
                                      np.max(pred_prob)*100,
                                      true_label), color=color)
```

[ ]:
```
plot_pred(predictions,
          val_labels,
          val_images,
          n=0)
```

Now we've got one function to visualize our models top prediction, let's make another to view our models top 10 predictions.

This function will: * Take an input of prediction probabilities array and a ground truth array and an integer. * Find the prediction using get_pred_label() * Find the top 10: * Prediction probabilities indexes * Prediction probabilities values * Prediction labels * Plot the top 10 prediction probability values and labels, coloring the true label grees

[32]:
```python
def plot_pred_conf(prediction_probabilities, labels, n=1):
  """
  Plus the top 10 highest prediction confidences along with the truth label for␣
↪sample n.
  """
  pred_prob, true_label = prediction_probabilities[n], labels[n]

  # Get the predicted label
  pred_label = get_pred_label(pred_prob)

  # Find the top 10 prediction confidence indexes
  top_10_pred_indexes = pred_prob.argsort()[-10:][::-1]

  # Find the top 10 prediction confidence values
  top_10_pred_values = pred_prob[top_10_pred_indexes]

  # Find the top 10 prediction labels
  top_10_pred_labels = unique_breeds[top_10_pred_indexes]

  # Setup plot
  top_plot = plt.bar(np.arange(len(top_10_pred_labels)),
                     top_10_pred_values,
                     color="grey")

  plt.xticks(np.arange(len(top_10_pred_labels)),
             labels=top_10_pred_labels,
             rotation="vertical")

  # Change color of true label
  if np.isin(true_label, top_10_pred_labels):
    top_plot[np.argmax(top_10_pred_labels == true_label)].set_color("green")
  else:
    pass
```

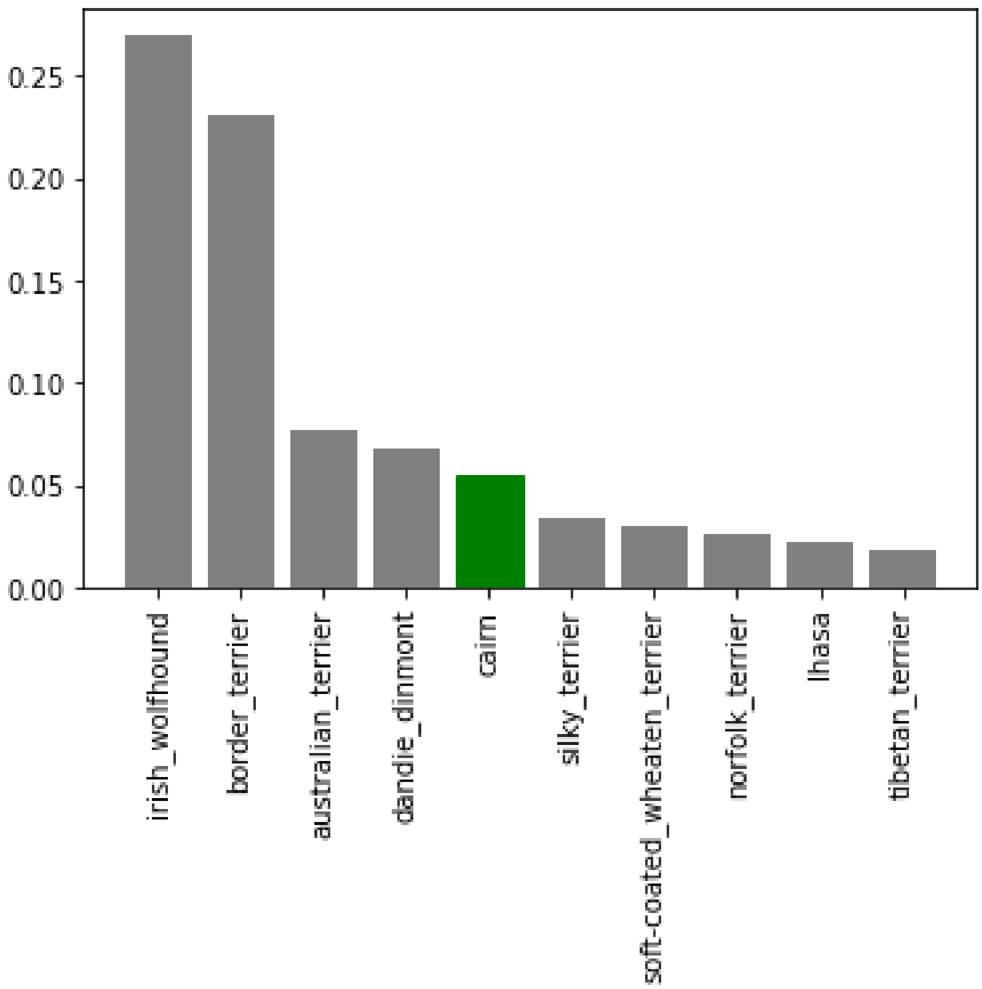
[ ]:
```python
unique_breeds[predictions[0].argsort()[-10:][::-1]] # top 10 indexes
```

[ ]:
```
array(['border_terrier', 'irish_wolfhound', 'cairn',
       'soft-coated_wheaten_terrier', 'silky_terrier', 'lhasa',
       'norfolk_terrier', 'komondor', 'australian_terrier', 'maltese_dog'],
      dtype=object)
```

[ ]:
```python
(unique_breeds[predictions[0].argsort()[-10:][::-1]] == val_labels[0]).argmax()
```

[ ]:
```
4
```

[ ]:
```python
plot_pred_conf(predictions, val_labels, n = 0)
```



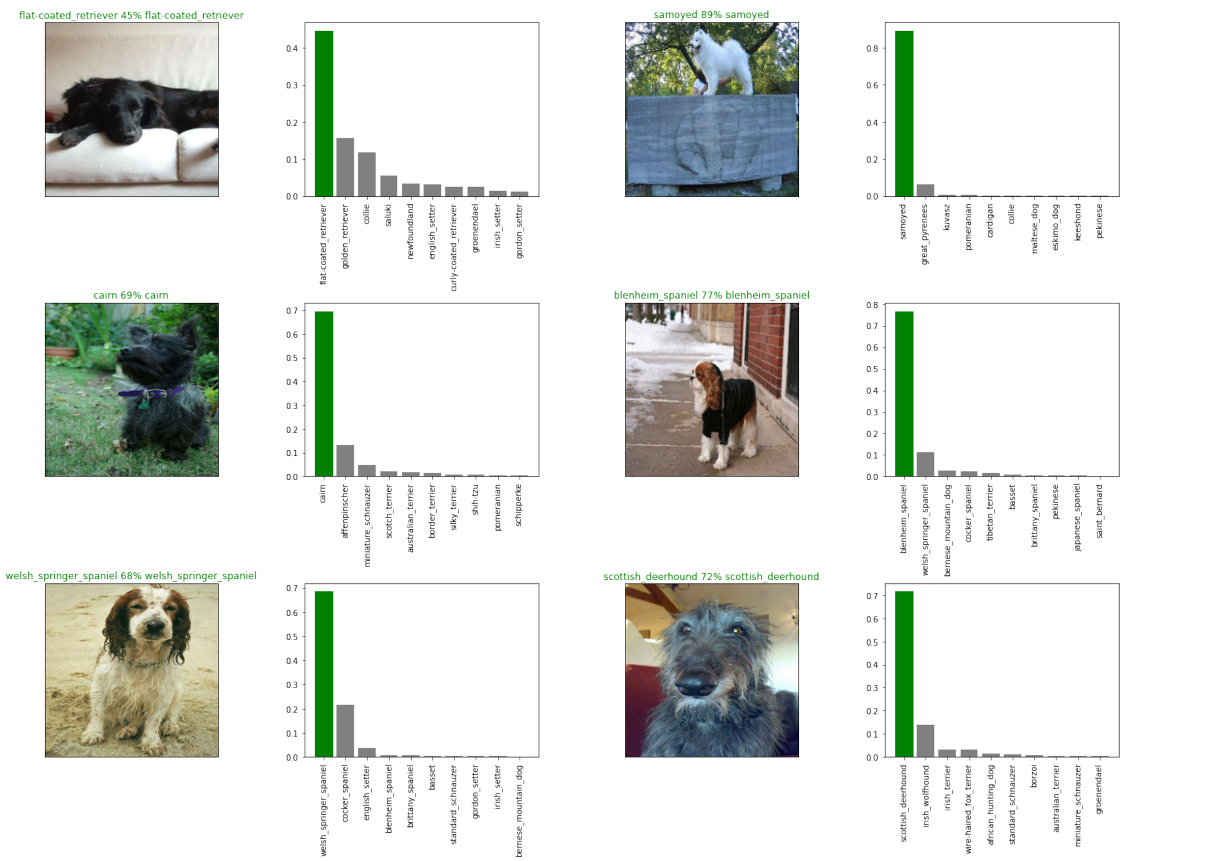
Now we've got some function to help us visualize our predictions and evaluate our model, let's check out a few.

[ ]:
```
# Let's check out a few predictions and their different values
i_multiplier = 20
num_rows = 3
num_cols = 2
num_images = num_rows * num_cols
plt.figure(figsize=(10*num_cols, 5*num_rows))

for i in range(num_images):
  # Showing Dog image
  plt.subplot(num_rows, 2*num_cols, 2*i + 1)
  plot_pred(prediction_probabilities = predictions,
            labels = val_labels,
            images = val_images,
```

```
            n = i + i_multiplier)
  # Plotting top 10 predictions probability bar graph
  plt.subplot(num_rows, 2*num_cols, 2*i + 2)
  plot_pred_conf(prediction_probabilities = predictions,
                 labels = val_labels,
                 n = i + i_multiplier)

plt.tight_layout(h_pad=1.0)
plt.show()
```



Challenge: How would you create a confusion matrix with our models predictions and true labels?

[60]:
```
# Creating confusion matrix
from tensorflow.math import confusion_matrix
from sklearn.metrics import multilabel_confusion_matrix

preds = loaded_1000_images_model.predict(val_data, verbose=1)
pred_lab = [unique_breeds[pred.argmax()] for pred in preds]
imgs, labs = unbatchify(val_data)
# confusion_matrix(val_labels, pred_lab)
```

```
cf_matrix = multilabel_confusion_matrix(labs, pred_lab, labels=unique_breeds)
```

```
7/7 [==============================] - 1s 145ms/step
```

[64]:
```
for i in range(len(unique_breeds)):
  print(unique_breeds[i], ":")
  print(cf_matrix[i])
  print()
```

```
affenpinscher :
[[196   2]
 [  0   2]]

afghan_hound :
[[196   0]
 [  0   4]]

african_hunting_dog :
[[198   0]
 [  0   2]]

airedale :
[[197   0]
 [  1   2]]

american_staffordshire_terrier :
[[200   0]
 [  0   0]]

appenzeller :
[[196   0]
 [  3   1]]

australian_terrier :
[[197   2]
 [  1   0]]

basenji :
[[197   1]
 [  0   2]]

basset :
[[198   0]
 [  0   2]]

beagle :
[[198   0]
 [  0   2]]
```

```
bedlington_terrier :
[[196   0]
 [  1   3]]

bernese_mountain_dog :
[[196   1]
 [  0   3]]

black-and-tan_coonhound :
[[199   1]
 [  0   0]]

blenheim_spaniel :
[[196   0]
 [  1   3]]

bloodhound :
[[199   0]
 [  0   1]]

bluetick :
[[197   0]
 [  1   2]]

border_collie :
[[198   0]
 [  2   0]]

border_terrier :
[[199   0]
 [  0   1]]

borzoi :
[[199   0]
 [  0   1]]

boston_bull :
[[198   1]
 [  0   1]]

bouvier_des_flandres :
[[197   2]
 [  0   1]]

boxer :
[[199   0]
 [  0   1]]
```

```
brabancon_griffon :
[[198   0]
 [  2   0]]

briard :
[[199   0]
 [  1   0]]

brittany_spaniel :
[[197   1]
 [  1   1]]

bull_mastiff :
[[199   0]
 [  0   1]]

cairn :
[[195   3]
 [  1   1]]

cardigan :
[[196   3]
 [  0   1]]

chesapeake_bay_retriever :
[[196   0]
 [  1   3]]

chihuahua :
[[197   0]
 [  0   3]]

chow :
[[199   1]
 [  0   0]]

clumber :
[[198   0]
 [  0   2]]

cocker_spaniel :
[[196   1]
 [  1   2]]

collie :
[[195   2]
 [  0   3]]
```

```
curly-coated_retriever :
[[196   1]
 [  0   3]]

dandie_dinmont :
[[196   0]
 [  2   2]]

dhole :
[[199   0]
 [  0   1]]

dingo :
[[199   0]
 [  0   1]]

doberman :
[[197   0]
 [  1   2]]

english_foxhound :
[[197   0]
 [  1   2]]

english_setter :
[[199   0]
 [  1   0]]

english_springer :
[[200   0]
 [  0   0]]

entlebucher :
[[198   1]
 [  0   1]]

eskimo_dog :
[[199   0]
 [  1   0]]

flat-coated_retriever :
[[198   0]
 [  1   1]]

french_bulldog :
[[200   0]
 [  0   0]]
```

```
german_shepherd :
[[199   0]
 [  0   1]]

german_short-haired_pointer :
[[200   0]
 [  0   0]]

giant_schnauzer :
[[198   0]
 [  1   1]]

golden_retriever :
[[198   1]
 [  0   1]]

gordon_setter :
[[200   0]
 [  0   0]]

great_dane :
[[197   0]
 [  3   0]]

great_pyrenees :
[[198   2]
 [  0   0]]

greater_swiss_mountain_dog :
[[200   0]
 [  0   0]]

groenendael :
[[198   0]
 [  0   2]]

ibizan_hound :
[[200   0]
 [  0   0]]

irish_setter :
[[197   0]
 [  1   2]]

irish_terrier :
[[198   0]
 [  0   2]]
```

```
irish_water_spaniel :
[[198   0]
 [  0   2]]

irish_wolfhound :
[[198   2]
 [  0   0]]

italian_greyhound :
[[197   1]
 [  1   1]]

japanese_spaniel :
[[199   0]
 [  0   1]]

keeshond :
[[199   0]
 [  0   1]]

kelpie :
[[197   0]
 [  3   0]]

kerry_blue_terrier :
[[198   0]
 [  0   2]]

komondor :
[[199   0]
 [  0   1]]

kuvasz :
[[199   0]
 [  0   1]]

labrador_retriever :
[[197   1]
 [  1   1]]

lakeland_terrier :
[[197   3]
 [  0   0]]

leonberg :
[[200   0]
 [  0   0]]
```

```
lhasa :
[[198   1]
 [  1   0]]

malamute :
[[198   1]
 [  0   1]]

malinois :
[[196   2]
 [  0   2]]

maltese_dog :
[[198   1]
 [  1   0]]

mexican_hairless :
[[196   1]
 [  2   1]]

miniature_pinscher :
[[198   1]
 [  0   1]]

miniature_poodle :
[[195   3]
 [  0   2]]

miniature_schnauzer :
[[198   2]
 [  0   0]]

newfoundland :
[[198   1]
 [  1   0]]

norfolk_terrier :
[[197   0]
 [  1   2]]

norwegian_elkhound :
[[200   0]
 [  0   0]]

norwich_terrier :
[[197   0]
 [  3   0]]
```

```
old_english_sheepdog :
[[199   0]
 [  0   1]]

otterhound :
[[199   0]
 [  0   1]]

papillon :
[[199   0]
 [  0   1]]

pekinese :
[[198   0]
 [  0   2]]

pembroke :
[[199   0]
 [  1   0]]

pomeranian :
[[196   0]
 [  1   3]]

pug :
[[198   0]
 [  1   1]]

redbone :
[[200   0]
 [  0   0]]

rhodesian_ridgeback :
[[196   3]
 [  0   1]]

rottweiler :
[[199   0]
 [  1   0]]

saint_bernard :
[[197   2]
 [  0   1]]

saluki :
[[197   0]
 [  0   3]]
```

```
samoyed :
[[195   0]
 [  0   5]]

schipperke :
[[199   1]
 [  0   0]]

scotch_terrier :
[[199   0]
 [  0   1]]

scottish_deerhound :
[[197   0]
 [  0   3]]

sealyham_terrier :
[[198   0]
 [  0   2]]

shetland_sheepdog :
[[198   0]
 [  1   1]]

shih-tzu :
[[198   2]
 [  0   0]]

siberian_husky :
[[198   0]
 [  0   2]]

silky_terrier :
[[198   1]
 [  1   0]]

soft-coated_wheaten_terrier :
[[199   0]
 [  1   0]]

staffordshire_bullterrier :
[[198   0]
 [  1   1]]

standard_poodle :
[[198   0]
 [  2   0]]
```

```
standard_schnauzer :
[[197   1]
 [  2   0]]

sussex_spaniel :
[[200   0]
 [  0   0]]

tibetan_mastiff :
[[199   0]
 [  1   0]]

tibetan_terrier :
[[200   0]
 [  0   0]]

toy_poodle :
[[195   2]
 [  1   2]]

toy_terrier :
[[199   0]
 [  1   0]]

vizsla :
[[197   0]
 [  3   0]]

walker_hound :
[[196   1]
 [  0   3]]

weimaraner :
[[198   1]
 [  0   1]]

welsh_springer_spaniel :
[[194   3]
 [  0   3]]

west_highland_white_terrier :
[[200   0]
 [  0   0]]

whippet :
[[196   1]
 [  1   2]]
```

```
wire-haired_fox_terrier :
[[197   0]
 [  2   1]]

yorkshire_terrier :
[[195   2]
 [  2   1]]
```

[58]:
```
val_images[val_labels == True]
```

[58]:
```
32
```

[70]:
```
[val_labels == True]
```

[70]:
```
[array([[False, False, False, …, False, False, False],
        [False, False, False, …, False, False, False],
        [False, False, False, …, False, False, False],
        …,
        [False, False, False, …, False, False, False],
        [False, False, False, …, False, False, False],
        [False, False, False, …, False, False, False]])]
```

1.13 Saving and reloading a trained model

[32]:
```
# Create a function to save a mocel
def save_model(model, suffix=None):
  """
  Saves a given model in a models directory and appends a suffix (string).
  """
  # Create a model directory pathname with current time
  modeldir = os.path.join("drive/My Drive/Dog Vision/models",
                          datetime.datetime.now().strftime("%Y%m%d-%H%M%s"))
  model_path = modeldir + "-" + suffix + ".h5" # save format of model
  print(f"Saving model to: {model_path}...")
  model.save(model_path)
  return model_path
```

[33]:
```
# Create a function to load a trained model
def load_model(model_path):
  """
  Load a saved model from a specified path.
  """
  print(f"Loading saved model from: {model_path}")
  model = tf.keras.models.load_model(model_path,
```

```
                                                custom_objects={"KerasLayer" : hub.
↪KerasLayer})
  return model
```

Now we've got funtions for saving and loading our trained model let's test them out

[ ]:
```
# Save our model trained on 1000 images
save_model(model, suffix="1000_images_mobilenetv2_Adam")
```

Saving model to: drive/My Drive/Dog Vision/models/20200704-07111593846702-1000_images_mobilenetv2_Adam.h5…

[ ]: 'drive/My Drive/Dog Vision/models/20200704-07111593846702-1000_images_mobilenetv2_Adam.h5'

[34]:
```
# Let's load our saved model
loaded_1000_images_model = load_model("drive/My Drive/Dog Vision/models/
↪20200704-07111593846702-1000_images_mobilenetv2_Adam.h5")
```

Loading saved model from: drive/My Drive/Dog Vision/models/20200704-07111593846702-1000_images_mobilenetv2_Adam.h5

[ ]:
```
# Evaluate the pre-saved model
model.evaluate(val_data)
```

7/7 [==============================] - 1s 101ms/step - loss: 1.2544 - accuracy: 0.6700

[ ]: [1.2543690204620361, 0.6700000166893005]

[ ]:
```
# Evaluate the loaded model
loaded_1000_images_model.evaluate(val_data)
```

7/7 [==============================] - 1s 98ms/step - loss: 1.2544 - accuracy: 0.6700

[ ]: [1.2543690204620361, 0.6700000166893005]

1.13.1 Training a big dog model (on the full data)

[ ]:
```
len(X), len(y)
```

[ ]: (10222, 10222)

[ ]:
```
# Create a data batch with the full data set
full_data = create_data_batches(X, y)
```

Creating training data batches…

```
[ ]: # Create a model for full dataset
full_model = create_model()
```

Building model with:
https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4

```
[ ]: # Create full model callbacks
full_model_tensorboard = create_tensorboard_callback()
# No validation set when training on all the data, so we can't monitor␣
↪validation accuracy
full_model_early_stopping = tf.keras.callbacks.EarlyStopping(monitor="accuracy",
                                                              patience=3)
```

Note: Running the cell below will take a little while (maybe upto 30 min for the first epoch) because the GPU we're using in the runtime has to load all of the images into memory.

```
[ ]: # Fit the full model to the full data
full_model.fit(x=full_data,
               epochs=NUM_EPOCHS,
               callbacks=[full_model_tensorboard, full_model_early_stopping])
```

```
Epoch 1/100
320/320 [==============================] - 5349s 17s/step - loss: 1.3429 -
accuracy: 0.6714
Epoch 2/100
320/320 [==============================] - 46s 143ms/step - loss: 0.3980 -
accuracy: 0.8829
Epoch 3/100
320/320 [==============================] - 46s 144ms/step - loss: 0.2360 -
accuracy: 0.9334
Epoch 4/100
320/320 [==============================] - 46s 143ms/step - loss: 0.1549 -
accuracy: 0.9643
Epoch 5/100
320/320 [==============================] - 46s 143ms/step - loss: 0.1072 -
accuracy: 0.9781
Epoch 6/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0789 -
accuracy: 0.9863
Epoch 7/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0584 -
accuracy: 0.9915
Epoch 8/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0464 -
accuracy: 0.9944
```

```
Epoch 9/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0403 -
accuracy: 0.9955
Epoch 10/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0316 -
accuracy: 0.9971
Epoch 11/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0257 -
accuracy: 0.9976
Epoch 12/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0229 -
accuracy: 0.9977
Epoch 13/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0205 -
accuracy: 0.9981
Epoch 14/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0174 -
accuracy: 0.9984
Epoch 15/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0165 -
accuracy: 0.9986
Epoch 16/100
320/320 [==============================] - 45s 141ms/step - loss: 0.0145 -
accuracy: 0.9987
Epoch 17/100
320/320 [==============================] - 46s 144ms/step - loss: 0.0145 -
accuracy: 0.9984
Epoch 18/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0121 -
accuracy: 0.9988
Epoch 19/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0123 -
accuracy: 0.9985
Epoch 20/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0098 -
accuracy: 0.9990
Epoch 21/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0101 -
accuracy: 0.9989
Epoch 22/100
320/320 [==============================] - 46s 143ms/step - loss: 0.0100 -
accuracy: 0.9989
Epoch 23/100
320/320 [==============================] - 45s 142ms/step - loss: 0.0141 -
accuracy: 0.9975
```

```
[ ]: <tensorflow.python.keras.callbacks.History at 0x7f36e1ed80f0>
```

[ ]:
```
save_model(full_model, suffix="full-image-set-mobilenetv2-Adam")
```

```
Saving model to: drive/My Drive/Dog Vision/models/20200704-09191593854382-full-
image-set-mobilenetv2-Adam.h5…
```

[ ]:
```
'drive/My Drive/Dog Vision/models/20200704-09191593854382-full-image-set-
mobilenetv2-Adam.h5'
```

[35]:
```
# Load the full model
loaded_full_model = load_model("drive/My Drive/Dog Vision/models/
 ↪20200704-09191593854382-full-image-set-mobilenetv2-Adam.h5")
```

```
Loading saved model from: drive/My Drive/Dog
Vision/models/20200704-09191593854382-full-image-set-mobilenetv2-Adam.h5
```

[ ]:
```
len(X)
```

[ ]:
```
10222
```

1.14 Making predictions on the test dataset

Since our model has been trained on images in the form of Tensor batches, to make predictions on the test data, we'll have to get it into the same format.

Luckily we created create_data_batches() earlier which can take a list of filenames as input and convert them into Tensor batches.

To make predictions on the test data, we'll: * Get the test image filenames. * Convert the filenames into test data batches using create_data_batches() and setting the test_data parameter to True (since the test data doesn't have labels). * Make a predictions array by passing the test batches to the predict() method called on our model.

[36]:
```
# Load test image filenames
test_path = "drive/My Drive/Dog Vision/test/"
test_filenames = [test_path + fpath for fpath in os.listdir(test_path)]
# for filepath in os.listdir("drive/My Drive/Dog Vision/test/"):
#   test_filepaths.append(os.path.join("drive/My Drive/Dog Vision/test/"␣
 ↪,filepath))
len(test_filenames)
```

[36]:
```
10357
```

[ ]:
```
# Create test data batch
test_data = create_data_batches(test_filenames, test_data=True)
```

```
Creating test data batches…
```

[ ]:
```
test_data
```

[ ]: <BatchDataset shapes: (None, 224, 224, 3), types: tf.float32>

Note: Calling predict() on our full model and passing it the test data batch will take a long time to run(about an ~1hr).

[ ]:
```python
# Make predictions on test data batch using the loaded full model

test_predictions = loaded_full_model.predict(test_data,
                                             verbose=1)
```

```
73/324 [=====>…] - ETA: 1:16:35

 ␣
↪---------------------------------------------------------------------------

    KeyboardInterrupt                         Traceback (most recent call␣
↪last)

    <ipython-input-114-1081792ff13f> in <module>()
      1 # Make predictions on test data batch using the loaded full model
      2 test_predictions = loaded_full_model.predict(test_data,
  ----> 3                                              verbose=1)


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/
↪training.py in _method_wrapper(self, *args, **kwargs)
     86       raise ValueError('{} is not supported in multi-worker mode.'.
↪format(
     87           method.__name__))
  ---> 88     return method(self, *args, **kwargs)
     89
     90   return tf_decorator.make_decorator(


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/keras/engine/
↪training.py in predict(self, x, batch_size, verbose, steps, callbacks,␣
↪max_queue_size, workers, use_multiprocessing)
   1266           for step in data_handler.steps():
   1267             callbacks.on_predict_batch_begin(step)
  -> 1268             tmp_batch_outputs = predict_function(iterator)
   1269             # Catch OutOfRangeError for Datasets of unknown size.
   1270             # This blocks until the batch has finished executing.


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/
↪def_function.py in __call__(self, *args, **kwds)
    578         xla_context.Exit()
```

```
    579     else:
--> 580       result = self._call(*args, **kwds)
    581
    582     if tracing_count == self._get_tracing_count():


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/
↪def_function.py in _call(self, *args, **kwds)
    616       # In this case we have not created variables on the first call.
↪ So we can
    617       # run the first trace but we should fail if variables are␣
↪created.
--> 618       results = self._stateful_fn(*args, **kwds)
    619       if self._created_variables:
    620         raise ValueError("Creating variables on a non-first call to␣
↪a function"


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.
↪py in __call__(self, *args, **kwargs)
   2418     with self._lock:
   2419       graph_function, args, kwargs = self.
↪_maybe_define_function(args, kwargs)
-> 2420     return graph_function._filtered_call(args, kwargs)  # pylint:␣
↪disable=protected-access
   2421
   2422   @property


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.
↪py in _filtered_call(self, args, kwargs)
   1663          if isinstance(t, (ops.Tensor,
   1664                             resource_variable_ops.
↪BaseResourceVariable))),
-> 1665         self.captured_inputs)
   1666
   1667   def _call_flat(self, args, captured_inputs,␣
↪cancellation_manager=None):


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.
↪py in _call_flat(self, args, captured_inputs, cancellation_manager)
   1744       # No tape is watching; skip to running the function.
   1745       return self._build_call_outputs(self._inference_function.call(
-> 1746           ctx, args, cancellation_manager=cancellation_manager))
   1747     forward_backward = self._select_forward_and_backward_functions(
   1748         args,
```

```
    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/function.
↪py in call(self, ctx, args, cancellation_manager)
    596               inputs=args,
    597               attrs=attrs,
--> 598               ctx=ctx)
    599         else:
    600           outputs = execute.execute_with_cancellation(


    /usr/local/lib/python3.6/dist-packages/tensorflow/python/eager/execute.
↪py in quick_execute(op_name, num_outputs, inputs, attrs, ctx, name)
     58     ctx.ensure_initialized()
     59     tensors = pywrap_tfe.TFE_Py_Execute(ctx._handle, device_name,␣
↪op_name,
---> 60                                         inputs, attrs, num_outputs)
     61   except core._NotOkStatusException as e:
     62     if name is not None:


    KeyboardInterrupt:
```

[ ]:
```
# Save predictions (NumPy array) to csv file
np.savetxt("drive/My Drive/Dog Vision/preds_array.csv", delimeter=",")
```

[41]:
```
# Load predictions (NumPy array) from csv file
test_predictions = np.loadtxt("drive/My Drive/Dog Vision/preds_array.csv",␣
↪delimiter=",")
```

[ ]:
```
test_predictions[:10]
```

[ ]:
```
array([[1.61196489e-09, 3.44086413e-12, 2.32834394e-11, …,
        1.06917716e-13, 1.58530451e-08, 1.52161670e-06],
       [3.17894322e-10, 3.20088262e-14, 1.85374840e-10, …,
        7.00588814e-08, 1.88822238e-08, 2.56980937e-10],
       [4.27301083e-09, 1.84139528e-13, 1.11784948e-09, …,
        2.71949238e-12, 2.23927123e-06, 7.41860809e-11],
       …,
       [4.47232779e-10, 4.28004029e-07, 4.11986996e-08, …,
        4.65437893e-07, 8.21722967e-10, 8.86002116e-09],
       [3.50528079e-11, 1.94377336e-03, 1.44941642e-10, …,
        1.56135718e-06, 6.13228721e-08, 7.32120961e-12],
       [1.23221771e-08, 3.08354520e-09, 1.87174110e-10, …,
        8.16165635e-10, 9.98905063e-01, 6.73740752e-09]])
```

1.15 Preparing test dataset predictions for kaggle

Looking at the Kaggle sample submission, we find that it wants our models prediction probability outputs in a DataFrame with an ID and a column for each different dog breed.

To get the data in this format, we'll: * Create a pandas DataFrame with an ID column as well as a column for each dog breed * Add data to the ID column by extracting the test image ID's from their filepaths * Add data (the prediction probabilities) to each of the dog breed columns. * Export the DataFrame as a CSV to submit it to Kaggle.

[ ]:
```
#list(unique_breeds)
```

[37]:
```
# Create pandas dataframe with empty columns
preds_df = pd.DataFrame(columns=["id"] + list(unique_breeds))
```

[38]:
```
# Append test images ID's to predictions DataFrame
test_ids = [os.path.splitext(path)[0] for path in os.listdir(test_path)]
preds_df["id"] = test_ids
```

[39]:
```
preds_df
```

[39]:
```
                                     id  … yorkshire_terrier
0      f157256196b2c6e28a739d2947e956e5  …               NaN
1      f0d2e080797f5e1f54bfa26bda41887b  …               NaN
2      ea1039f3869357b53abf4ace351218a6  …               NaN
3      efb73fb00c85027773f3bef3dfc6c06b  …               NaN
4      f1230a99088c9bc88bc2989affee43d2  …               NaN
…                                   …   …                 …
10352  055cb4e6ba1540c275d6ccd2c0b52c27  …               NaN
10353  0551d6061247bb9cb351e94ae392d0ae  …               NaN
10354  0568885d3278881be3fa98b4b7e85efb  …               NaN
10355  056eca95930e26c627c11e14cd9e1b3a  …               NaN
10356  0518d41fa95b141f8267a3d2e2cef756  …               NaN

[10357 rows x 121 columns]
```

[42]:
```
# Add the prediction probabilities to each dog breed column
preds_df[list(unique_breeds)] = test_predictions
preds_df
```

[42]:
```
                                     id  … yorkshire_terrier
0      f157256196b2c6e28a739d2947e956e5  …       1.52162e-06
1      f0d2e080797f5e1f54bfa26bda41887b  …       2.56981e-10
2      ea1039f3869357b53abf4ace351218a6  …       7.41861e-11
3      efb73fb00c85027773f3bef3dfc6c06b  …       6.13453e-11
4      f1230a99088c9bc88bc2989affee43d2  …       1.31887e-06
```

```
…                                         …   …               …
10352  055cb4e6ba1540c275d6ccd2c0b52c27  …         0.0680157
10353  0551d6061247bb9cb351e94ae392d0ae  …       4.74548e-07
10354  0568885d3278881be3fa98b4b7e85efb  …       6.54013e-09
10355  056eca95930e26c627c11e14cd9e1b3a  …       2.46965e-13
10356  0518d41fa95b141f8267a3d2e2cef756  …       3.84285e-09

[10357 rows x 121 columns]
```

[ ]:
```python
# Save our predictions dataframe to CSV for kaggle submission
preds_df.to_csv("drive/My Drive/Dog Vision/Dog_breeds_submission_mobilenetv2.
 ↪csv", index=False)
```

1.16 Making predictions on custom images

To make predictions on custom images, we'll: * Get the filepaths of our own images * Turn the filespaths into data batches using create_data_batches() function. And since our data batches won't have labels we will be using test_data parameter to True. * Pass the custom image batch to our models predict() method. * Convert the prediction probabilities to labels. * And compare the predicted outputs to true labels.

[66]:
```python
custom_path = "drive/My Drive/Dog Vision/my-dog-images/"
custom_images_paths = [custom_path + fname for fname in os.listdir(custom_path)]
```

[67]:
```python
custom_images_paths
```

[67]:
```
['drive/My Drive/Dog Vision/my-dog-images/chihuahua.jpg',
 'drive/My Drive/Dog Vision/my-dog-images/dog-photo-2.jpeg',
 'drive/My Drive/Dog Vision/my-dog-images/dog-photo-1.jpeg',
 'drive/My Drive/Dog Vision/my-dog-images/dog-photo-3.jpeg',
 'drive/My Drive/Dog Vision/my-dog-images/siberian_husky.jpg',
 'drive/My Drive/Dog Vision/my-dog-images/kuvasz.jpg',
 'drive/My Drive/Dog Vision/my-dog-images/miniature_pinscher.jpg',
 'drive/My Drive/Dog Vision/my-dog-images/labrador.jpg']
```

[68]:
```python
# Turn custom images into data batches
custom_data = create_data_batches(custom_images_paths, test_data=True)
custom_data
```

Creating test data batches…

[68]:
```
<BatchDataset shapes: (None, 224, 224, 3), types: tf.float32>
```

[69]:
```python
# Make predictions on the custom data
custom_preds = loaded_full_model.predict(custom_data)
custom_preds.shape
```

[69]: (8, 120)

[70]:
```
# Get custom images prediction labels
custom_labels = [get_pred_label(custom_preds[i]) for i in␣
↪range(len(custom_preds))]
custom_labels
```

[70]:
```
['miniature_pinscher',
 'lakeland_terrier',
 'golden_retriever',
 'kuvasz',
 'siberian_husky',
 'great_pyrenees',
 'doberman',
 'labrador_retriever']
```

[71]:
```
# Get custom images
custom_images = []
# Loop through unbatched data
for image in custom_data.unbatch().as_numpy_iterator():
  custom_images.append(image)
```

[84]:
```
# Check custom image predictions
plt.figure(figsize=(10, 10))
for i, image in enumerate(custom_images):
  plt.subplot(2, 4, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.title(custom_labels[i])
  plt.imshow(image)
  #plt.axis('off')
plt.savefig("drive/My Drive/Dog Vision/Custom_dog_predictions.jpg")
```

[78]:
```python
'labrador' in unique_breeds
```

[78]: False

[ ]: