Open in Colab

🐶 Using Transfer Learning and TensorFlow 2.0 to Classify Different Dog Breeds

Who's that doggy in the window?

Dogs are incredible. But have you ever been sitting at a cafe, seen a dog and not known what breed it is? I have. And then someone says, "it's an English Terrier" and you think, how did they know that?

In this project we're going to be using machine learning to help us identify different breeds of dogs.

To do this, we'll be using data from the Kaggle dog breed identification competition. It consists of a collection of 10,000+ labelled images of 120 different dog breeds.

This kind of problem is called multi-class image classification. It's multi-class because we're trying to classify mutliple different breeds of dog. If we were only trying to classify dogs versus cats, it would be called binary classification (one thing versus another).

Multi-class image classification is an important problem because it's the same kind of technology Tesla uses in their self-driving cars or Airbnb uses in atuomatically adding information to their listings.

Since the most important step in a deep learng problem is getting the data ready (turning it into numbers), that's what we're going to start with.

We're going to go through the following TensorFlow/Deep Learning workflow:

1. Get data ready (download from Kaggle, store, import).
2. Prepare the data (preprocessing, the 3 sets, X & y).
3. Choose and fit/train a model (TensorFlow Hub, `tf.keras.applications`, TensorBoard, EarlyStopping).
4. Evaluating a model (making predictions, comparing them with the ground truth labels).
5. Improve the model through experimentation (start with 1000 images, make sure it works, increase the number of images).
6. Save, sharing and reloading your model (once you're happy with the results).

For preprocessing our data, we're going to use TensorFlow 2.x. The whole premise here is to get our data into Tensors (arrays of numbers which can be run on GPUs) and then allow a machine learning model to find patterns between them.

For our machine learning model, we're going to be using a pretrained deep learning model from TensorFlow Hub.

The process of using a pretrained model and adapting it to your own problem is called **transfer learning**. We do this because rather than train our own model from scratch (could be timely and expensive), we leverage the patterns of another model which has been trained to classify images.

Getting our workspace ready

Before we get started, since we'll be using TensorFlow 2.x and TensorFlow Hub (TensorFlow Hub), let's import them.

NOTE: Don't run the cell below if you're already using TF 2.x.

```
# # At time of recording, TF 2.x is not the default
# import tensorflow as tf
# print("TF version:", tf.__version__)
```

To import TensorFlow 2.x, we can run the cell bellow. But it'll tell us TensorFlow is already loaded and we have to restart the runtime (runtime -> restart runtime).

```
# Import TF 2.x
try:
  # %tensorflow_version only exists in Colab
  %tensorflow_version 2.x
except Exception:
  pass
```

```
    TensorFlow 2.x selected.
```

After restarting the runtime and rerunning the cell above, it tells us TensorFlow 2.x is selected.

NOTE: You may not need to do the above steps in the future when TensorFlow 2.x becomes the default in Colab.

Let's rerun some import statements. And check whether or not we're using a GPU.

```
import·tensorflow·as·tf
import·tensorflow_hub·as·hub

print("TF version:", tf.__version__)
print("Hub version:", hub.__version__)

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

```
    TF version: 2.1.0
    Hub version: 0.7.0
    GPU available (YESS!!!!)
```

You might be wondering what a GPU is or why we need one. The short story is, a GPU is a computer chip which is faster at doing numerical computing. And since machine learning is all about finding patterns in numbers, that's what we're after.

Running this for the first time in Colab will let us know there's no GPU available.

This is because by default Colab runs on a computer located on Google's servers which doesn't have a GPU attached to it.

But we can fix this going to runtime and then changing the runtime type:

1. Go to Runtime.
2. Click "Change runtime type".
3. Where it says "Hardware accelerator", choose "GPU" (don't worry about TPU for now but feel free to research them).
4. Click save.
5. The runtime will be restarted to activate the new hardware, so you'll have to rerun the above cells.
6. If the steps have worked you should see a print out saying "GPU available".

If you want an example of how much a GPU speeds up computing, Google Colab have a demonstration notebook available.

Getting data ready

Since much of machine learning is getting your data ready to be used with a machine learning model, we'll take extra care getting it setup.

There are a few ways we could do this. Many of them are detailed in the Google Colab notebook on I/O (input and output).

And because the data we're using is hosted on Kaggle, we could even use the Kaggle API.

This is great but what if the data you want to use wasn't on Kaggle?

One method is to upload it to your Google Drive, mount your drive in this notebook and import the file.

```
# Running this cell will provide you with a token to link your drive to this notebook
from google.colab import drive
drive.mount('/content/drive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8q

Enter your authorization code:
··········
Mounted at /content/drive
```

Following the prompts from the cell above, if everything worked, you should see a "drive" folder available under the Files tab.

This means we'll be able to access files in our Google Drive right in this notebook.

For this project, I've downloaded the data from Kaggle and uploaded it to my Google Drive as a .zip file under the folder "Data".

To access it, we'll have to unzip it.

Note: Running the cell below for the first time could take a while (a couple of minutes is normal). After you've run it once and got the data in your Google Drive, you don't need to run it again.

Note 2: Wherever you see something like `drive/My Drive/Data/` you will need to change it to wherever you are storing your files for this project. The first place you'll have to change it is the cell below.

```
# Use the '-d' parameter as the destination for where the files should go
#!unzip "drive/My Drive/Data/dog-breed-identification.zip" -d "drive/My Drive/Data/"
```

Once the files have been unzipped to your Google Drive, you don't have to run the cell above anymore.

Accessing the data

Now the data files we're working with are available on our Google Drive, we can start to check it out.

Let's start with `labels.csv` which contains all of the image ID's and their assosciated dog breed (our data and labels).

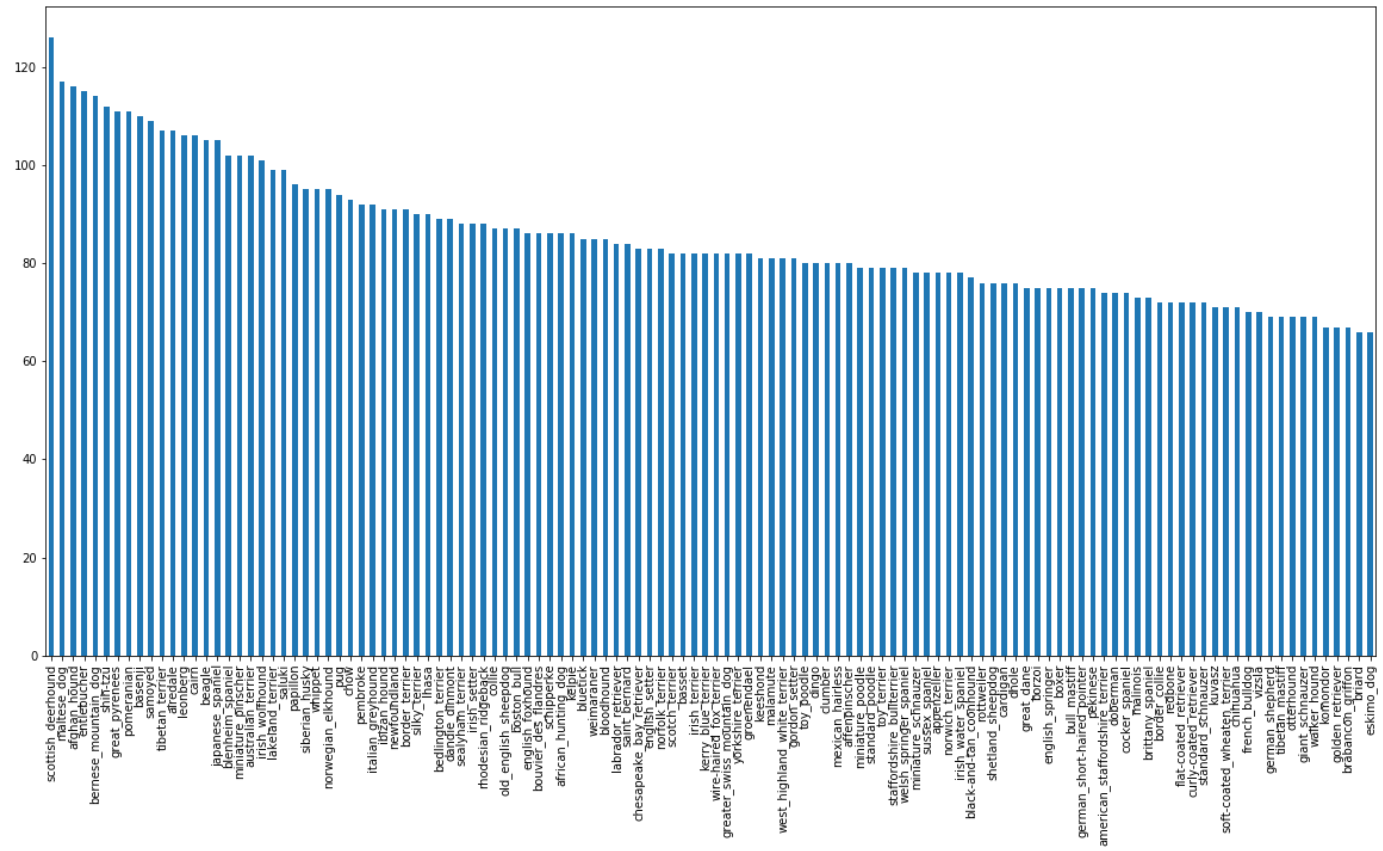
```
# Checkout the labels of our data
import pandas as pd
labels_csv = pd.read_csv("drive/My Drive/Data/labels.csv")
print(labels_csv.describe())
print(labels_csv.head())
```

```
                                  id               breed
count                          10222               10222
unique                         10222                 120
top     45fef015b1974e98da0173e8260b3482  scottish_deerhound
freq                               1                 126
                                 id             breed
0  000bec180eb18c7604dcecc8fe0dba07       boston_bull
1  001513dfcb2ffafc82cccf4d8bbaba97             dingo
2  001cdf01b096e06d78e9e5112d419397          pekinese
3  00214f311d5d2247d5dfe4fe24b2303d          bluetick
4  0021f9ceb3235effd7fcde7f7538ed62  golden_retriever
```

Looking at this, we can see there are 10222 different ID's (meaning 10222 different images) and 120 different breeds.

Let's figure out how many images there are of each breed.

```
# How many images are there of each breed?
labels_csv["breed"].value_counts().plot.bar(figsize=(20, 10));
```



Okay sweet. If we were to roughly draw a line across the middle of the graph, we'd see there's about 60+ images for each dog breed.

This is a good amount as for some of their vision products Google recommends a minimum of 10 images per class to get started. And as you might imagine, the more images per class available, the more chance a model has to figure out patterns between them.

Let's check out one of the images.

Note: Loading an image file for the first time may take a while as it gets loaded into the runtime memory. If you see a Google Drive timeout error, check out the Colab FAQ for more.

```
from IPython.display import display, Image
# Image("drive/My Drive/Data/train/000bec180eb18c7604dcecc8fe0dba07.jpg")
```

Getting images and their labels

Since we've got the image ID's and their labels in a DataFrame (`labels_csv`), we'll use it to create:

- A list a filepaths to training images
- An array of all labels
- An array of all unique labels

We'll only create a list of filepaths to images rather than importing them all to begin with. This is because working with filepaths (strings) is much efficient than working with images.

```
# Create pathnames from image ID's
filenames = ["drive/My Drive/Data/train/" + fname + ".jpg" for fname in labels_csv["id"]]

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

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

Now we've got a list of all the filenames from the ID column of `labels_csv`, we can compare it to the number of files in our training data directory to see if they line up.

If they do, great. If not, there may have been an issue when unzipping the data (what we did above), to fix this, you might have to unzip the data again. Be careful not to let your Colab notebook disconnect whilst unzipping.

```
# Check whether number of filenames matches number of actual image files
import os
if len(os.listdir("drive/My Drive/Data/train/")) == len(filenames):
  print("Filenames match actual amount of files!")
else:
  print("Filenames do not match actual amount of files, check the target directory.")
```

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

If everything worked, we should see a match up.

Let's do one more check. Visualizing directly from a filepath.

```
# Check an image directly from a filepath
Image(filenames[9000])
```

Woah! What a beast!

Now we've got our image filepaths together, let's get the labels.

We'll take them from `labels_csv` and turn them into a NumPy array.

```
import numpy as np
labels = labels_csv["breed"].to_numpy() # convert labels column to NumPy array
labels[:10]
```

```
array(['boston_bull', 'dingo', 'pekinese', 'bluetick', 'golden_retriever',
       'bedlington_terrier', 'bedlington_terrier', 'borzoi', 'basenji',
       'scottish_deerhound'], dtype=object)
```

Wonderful, now lets do the same thing as before, compare the amount of labels to number of filenames.

```
# 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!
```

If it all worked, we should have the same amount of images and labels.

Finally, since a machine learning model can't take strings as input (what `labels` currently is), we'll have to convert our labels to numbers.

To begin with, we'll find all of the unique dog breed names.

Then we'll go through the list of `labels` and compare them to unique breeds and create a list of booleans indicating which one is the real label (`True`) and which ones aren't (`False`).

```
# Find the unique label values
unique_breeds = np.unique(labels)
len(unique_breeds)
```

```
120
```

The length of `unique_breeds` should be 120, meaning we're working with images of 120 different breeds of dogs.

Now use `unique_breeds` to help turn our `labels` array into an array of booleans.

```
# Example: Turn one label into an array of booleans
print(labels[0])
labels[0] == unique_breeds # use comparison operator to create boolean array
```

```
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])
```

That's for one example, let's do the whole thing.

```
# Turn every label into a boolean array
boolean_labels = [label == np.array(unique_breeds) for label in labels]
boolean_labels[:2]
```

```
[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])]
```

Why do it like this?

Remember, an important concept in machine learning is converting your data to numbers before passing it to a machine learning model.

In this case, we've transformed a single dog breed name such as `boston_bull` into a one-hot array.

Let's see an example.

```
# Example: Turning a boolean array into integers
print(labels[0]) # original label
print(np.where(unique_breeds == labels[0])[0][0]) # index where label occurs
print(boolean_labels[0].argmax()) # index where label occurs in boolean array
print(boolean_labels[0].astype(int)) # there will be a 1 where the sample label occurs
```

```
boston_bull
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 0 0]
```

Wonderful! Now we've got our labels in a numeric format and our image filepaths easily accessible (they aren't numeric yet), let's split our data up.

Creating our own validation set

Since the dataset from Kaggle doesn't come with a validation set (a split of the data we can test our model on before making final predicitons on the test set), let's make one.

We could use Scikit-Learn's [train_test_split](#) function or we could simply make manual splits of the data.

For accessibility later, let's save our filenames variable to `X` (data) and our labels to `y`.

```
# Setup X & y variables
X = filenames
y = boolean_labels
```

Since we're working with 10,000+ images, it's a good idea to work with a portion of them to make sure things are working before training on them all.

This is because computing with 10,000+ images could take a fairly long time. And our goal when working through machine learning projects is to reduce the time between experiments.

Let's start experimenting with 1000 and increase it as we need.

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

NUM_IMAGES: 1000

```
1000
```

Now let's split our data into training and validation sets. We'll use and 80/20 split (80% training data, 20% validation data).

```
# Import train_test_split from Scikit-Learn
from sklearn.model_selection import train_test_split

# Split them into training and validation using 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(y_train), len(X_val), len(y_val)
```

```
(800, 800, 200, 200)
```

```
# Check out the training data (image file paths and labels)
X_train[:5], y_train[:2]
```

```
(['drive/My Drive/Data/train/00bee065dcec471f26394855c5c2f3de.jpg',
  'drive/My Drive/Data/train/0d2f9e12a2611d911d91a339074c8154.jpg',
  'drive/My Drive/Data/train/1108e48ce3e2d7d7fb527ae6e40ab486.jpg',
  'drive/My Drive/Data/train/0dc3196b4213a2733d7f4bdcd41699d3.jpg',
  'drive/My Drive/Data/train/146fbfac6b5b1f0de83a5d0c1b473377.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]),
  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])])
```

Preprocessing images (turning images into Tensors)

Our labels are in numeric format but our images are still just file paths.

Since we're using TensorFlow, our data has to be in the form of Tensors.

A Tensor is a way to represent information in numbers. If you're familar with NumPy arrays (you should be), a Tensor can be thought of as a combination of NumPy arrays, except with the special ability to be used on a GPU.

Because of how TensorFlow stores information (in Tensors), it allows machine learning and deep learning models to be run on GPUs (generally faster at numerical computing).

To preprocess our images into Tensors we're going to write a function which does a few things:

1. Takes an image filename as input.
2. Uses TensorFlow to read the file and save it to a variable, `image`.
3. Turn our `image` (a jpeg file) into Tensors.
4. Resize the `image` to be of shape (224, 224).
5. Return the modified `image`.

A good place to read about this type of function is the TensorFlow documentation on loading images.

You might be wondering why (224, 224), which is (heigh, width). It's because this is the size of input our model (we'll see this soon) takes, an image which is (224, 224, 3).

What? Where's the 3 from? We're getting ahead of ourselves but that's the number of colour channels per pixel, red, green and blue.

Let's make this a little more concrete.

```
# Convert image to NumPy array
from matplotlib.pyplot import imread
image = imread(filenames[42]) # read in an image
image.shape
```

```
(257, 350, 3)
```

Notice the shape of `image`. It's (257, 350, 3). This is height, width, colour channel value.

And you can easily convert it to a Tensor using `tf.constant()`.

```
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)>
```

Ok, now let's build that function we were talking about.

```
# Define image size
IMG_SIZE = 224

def process_image(image_path):
  """
  Takes an image file path and turns it into a Tensor.
  """
  # Read in image file
  image = tf.io.read_file(image_path)
  # Turn the jpeg image into numerical Tensor with 3 colour channels (Red, Green, Blue)
  image = tf.image.decode_jpeg(image, channels=3)
  # Convert the colour channel values from 0-225 values to 0-1 values
  image = tf.image.convert_image_dtype(image, tf.float32)
  # Resize the image to our desired size (224, 244)
  image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])
  return image
```

Creating data batches

Wonderful. Now we've got a function to convert our images into Tensors, we'll now build one to turn our data into batches (more specifically, a TensorFlow `BatchDataset`).

What's a batch?

A batch (also called mini-batch) is a small portion of your data, say 32 (32 is generally the default batch size) images and their labels. In deep learning, instead of finding patterns in an entire dataset at the same time, you often find them one batch at a time.

Let's say you're dealing with 10,000+ images (which we are). Together, these files may take up more memory than your GPU has. Trying to compute on them all would result in an error.

Instead, it's more efficient to create smaller batches of your data and compute on one batch at a time.

TensorFlow is very efficient when your data is in batches of (image, label) Tensors. So we'll build a function to do create those first. We'll take advantage of of `process_image` function at the same time.

```
# 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
```

Now we've got a simple function to turn our image file path names and their associated labels into tuples (we can turn these into Tensors next), we'll create a function to make data batches.

Because we'll be dealing with 3 different sets of data (training, validation and test), we'll make sure the function can accomodate for each set.

We'll set a default batch size of 32 because [according to Yann Lecun](#) (one of the OG's of deep learning), friends don't let friends train with batch sizes over 32.

```
# Define the batch size, 32 is a good default
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 it 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
    data_batch = data.map(process_image).batch(BATCH_SIZE)
    return data_batch

  # If the data if 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:
    # If the data is a training dataset, we shuffle it
    print("Creating training data batches...")
    # Turn filepaths and labels into Tensors
```

```
    data = tf.data.Dataset.from_tensor_slices((tf.constant(x), # filepaths
                                               tf.constant(y))) # labels

    # 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 data into batches
    data_batch = data.batch(BATCH_SIZE)
  return data_batch
```

```
# 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)))
```

Look at that! We've got our data in batches, more specifically, they're in Tensor pairs of (images, labels) ready for use on a GPU.

But having our data in batches can be a bit of a hard concept to understand. Let's build a function which helps us visualize what's going on under the hood.

Visualizing data batches

```
import matplotlib.pyplot as plt

# Create a function for viewing images in a data batch
def show_25_images(images, labels):
  """
  Displays 25 images from a 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 gird lines off
    plt.axis("off")
```

To make computation efficient, a batch is a tighly wound collection of Tensors.

So to view data in a batch, we've got to unwind it.

We can do so by calling the `as_numpy_iterator()` method on a data batch.

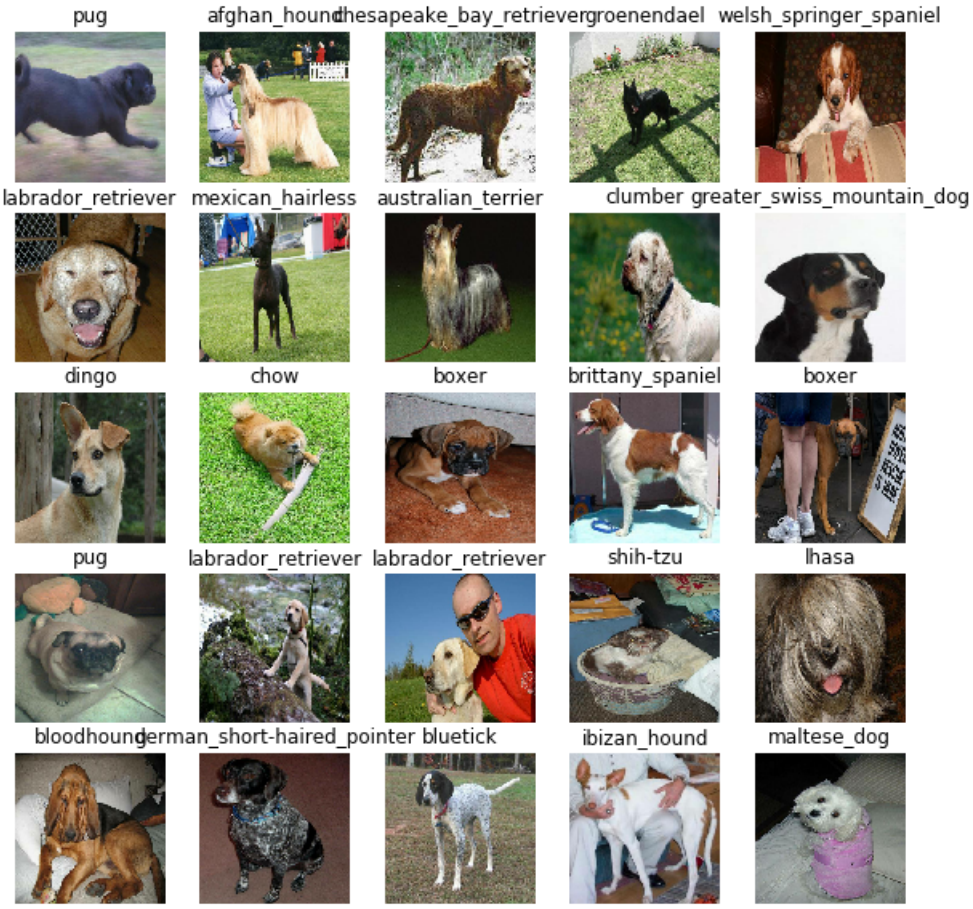
This will turn our a data batch into something which can be iterated over.

Passing an iterable to `next()` will return the next item in the iterator.

In our case, next will return a batch of 32 images and label pairs.

Note: Running the cell below and loading images may take a little while.

```
# Visualize training images from the training data batch
train_images, train_labels = next(train_data.as_numpy_iterator())
show_25_images(train_images, train_labels)
```



Look at all those beautiful dogs!

Question: Rerun the cell above, why do you think a different set of images is displayed each time you run it?

```
# Visualize validation images from the validation data batch
val_images, val_labels = next(val_data.as_numpy_iterator())
show_25_images(val_images, val_labels)
```

Even more dogs!

Question: Why does running the cell above and viewing validation images return the same dogs each time?

Creating and training a model

Now our data is ready, let's prepare it modelling. We'll use an existing model from TensorFlow Hub.

TensorFlow Hub is a resource where you can find pretrained machine learning models for the problem you're working on.

Using a pretrained machine learning model is often referred to as **transfer learning**.

Why use a pretrained model?

Building a machine learning model and training it on lots from scratch can be expensive and time consuming.

Transfer learning helps eliviate some of these by taking what another model has learned and using that information with your own problem.

How do we choose a model?

Since we know our problem is image classification (classifying different dog breeds), we can navigate the TensorFlow Hub page by our problem domain (image).

We start by choosing the image problem domain, and then can filter it down by subdomains, in our case, image classification.

Doing this gives a list of different pretrained models we can apply to our task.

Clicking on one gives us information about the model as well as instructions for using it.

For example, clicking on the mobilenet_v2_130_224 model, tells us this model takes an input of images in the shape 224, 224. It also says the model has been trained in the domain of image classification.

Let's try it out.

Building a model

Before we build a model, there are a few things we need to define:

- The input shape (images, in the form of Tensors) to our model.
- The output shape (image labels, in the form of Tensors) of our model.
- The URL of the model we want to use.

These things will be standard practice with whatever machine learning model you use. And because we're using TensorFlow, everything will be in the form of Tensors.

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

# Setup output shape of the model
OUTPUT_SHAPE = len(unique_breeds) # number of unique labels

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

Now we've got the inputs, outputs and model we're using ready to go. We can start to put them together

There are many ways of building a model in TensorFlow but one of the best ways to get started is to use the Keras API.

Defining a deep learning model in Keras can be as straightforward as saying, "here are the layers of the model, the input shape and the output shape, let's go!"

Knowing this, let's create a function which:

- Takes the input shape, output shape and the model we've chosen's URL as parameters.
- Defines the layers in a Keras model in a sequential fashion (do this first, then this, then that).
- Compiles the model (says how it should be evaluated and improved).
- Builds the model (tells it what kind of input shape it'll be getting).
- Returns the model.

We'll take a look at the code first, then dicuss each part.

```
# 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([
    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(), # Our model wants to reduce this (how wrong its guesse
      optimizer=tf.keras.optimizers.Adam(), # A friend telling our model how to improve its guesses
      metrics=["accuracy"] # We'd like this to go up
  )

  # Build the model
  model.build(INPUT_SHAPE) # Let the model know what kind of inputs it'll be getting

  return model
```

What's happening here?

Setting up the model layers

There are two ways to do this in Keras, the functional and sequential API. We've used the sequential.

Which one should you use?

The Keras documentation states the functional API is the way to go for defining complex models but the sequential API (a linear stack of layers) is perfectly fine for getting started, which is what we're doing.

The first layer we use is the model from TensorFlow Hub (`hub.KerasLayer(MODEL_URL)`. So our first layer is actually an entire model (many more layers). This **input layer** takes in our images and finds patterns in them based on the patterns `mobilenet_v2_130_224` has found.

The next layer (`tf.keras.layers.Dense()`) is the **output layer** of our model. It brings all of the information discovered in the input layer together and outputs it in the shape we're after, 120 (the number of unique labels we have).

The `activation="softmax"` parameter tells the output layer, we'd like to assign a probability value to each of the 120 labels somewhere between 0 & 1. The higher the value, the more the model believes the input image should have that label. If we were working on a binary classification problem, we'd use `activation="sigmoid"`.

For more on which activation function to use, see the article Which Loss and Activation Functions Should I Use?

Compiling the model

This one is best explained with a story.

Let's say you're at the international hill descending championships. Where your start standing on top of a hill and your goal is to get to the bottom of the hill. The catch is you're blindfolded.

Luckily, your friend Adam is standing at the bottom of the hill shouting instructions on how to get down.

At the bottom of the hill there's a judge evaluating how you're doing. They know where you need to end up so they compare how you're doing to where you're supposed to be. Their comparison is how you get scored.

Transferring this to `model.compile()` terminology:

- `loss` - The height of the hill is the loss function, the models goal is to minimize this, getting to 0 (the bottom of the hill) means the model is learning perfectly.
- `optimizer` - Your friend Adam is the optimizer, he's the one telling you how to navigate the hill (lower the loss function) based on what you've done so far. His name is Adam because the Adam optimizer is a great general which performs well on most models. Other optimizers include RMSprop and Stochastic Gradient Descent.
- `metrics` - This is the onlooker at the bottom of the hill rating how well your perfomance is. Or in our case, giving the accuracy of how well our model is predicting the correct image label.

Building the model

We use `model.build()` whenever we're using a layer from TensorFlow Hub to tell our model what input shape it can expect.

In this case, the input shape is `[None, IMG_SIZE, IMG_SIZE, 3]` or `[None, 224, 224, 3]` or `[batch_size, img_height, img_width, color_channels]`.

Batch size is left as `None` as this is inferred from the data we pass the model. In our case, it'll be 32 since that's what we've set up our data batches as.

Now we've gone through each section of the function, let's use it to create a model.

We can call `summary()` on our model to get idea of what our model looks like.

```
# Create a model and check its details
model = create_model()
model.summary()
```

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
keras_layer_1 (KerasLayer)   multiple                  5432713
_________________________________________________________________
dense_1 (Dense)              multiple                  120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

The non-trainable parameters are the patterns learned by `mobilenet_v2_130_224` and the trainable parameters are the ones in the dense layer we added.

This means the main bulk of the information in our model has already been learned and we're going to take that and adapt it to our own problem.

Creating callbacks

We've got a model ready to go but before we train it we'll make some callbacks.

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

The two callbacks we're going to add are a TensorBoard callback and an Early Stopping callback.

TensorBoard Callback

TensorBoard helps provide a visual way to monitor the progress of your model during and after training.

It can be used directly in a notebook to track the performance measures of a model such as loss and accuracy.

To set up a TensorBoard callback and view TensorBoard in a notebook, we need to do three 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 model's `fit()` function.
3. Visualize the our models training logs using the `%tensorboard` magic function (we'll do this later on).

```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

```
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/Data/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)
```

Early Stopping Callback

Early stopping helps prevent overfitting by stopping a model when a certain evaluation metric stops improving. If a model trains for too long, it can do so well at finding patterns in a certain dataset that it's not able to use those patterns on another dataset it hasn't seen before (doesn't generalize).

It's basically like saying to our model, "keep finding patterns until the quality of those patterns starts to go down."

```
# Create early stopping (once our model stops improving, stop training)
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_accuracy",
```

```
                                                patience=3) # stops after 3 rounds of no improvements
```

Training a model (on a subset of data)

Our first model is only going to be trained on 1000 images. Or trained on 800 images and then validated on 200 images, meaning 1000 images total or about 10% of the total data.

We do this to make sure everything is working. And if it is, we can step it up later and train on the entire training dataset.

The final parameter we'll define before training is `NUM_EPOCHS` (also known as **number of epochs**).

`NUM_EPOCHS` defines how many passes of the data we'd like our model to do. A pass is equivalent to our model trying to find patterns in each dog image and see which patterns relate to each label.

If `NUM_EPOCHS=1`, the model will only look at the data once and will probably score badly because it hasn't a chance to correct itself. It would be like you competing in the international hill descent championships and your friend Adam only being able to give you 1 single instruction to get down the hill.

What's a good value for `NUM_EPOCHS`?

This one is hard to say. 10 could be a good start but so could 100. This is one of the reasons we created an early stopping callback. Having early stopping setup means if we set `NUM_EPOCHS` to 100 but our model stops improving after 22 epochs, it'll stop training.

Along with this, let's quickly check if we're still using a GPU.

```
# Check again if GPU is available (otherwise computing will take a looooonnnnggggg time)
print("GPU", "available (YESS!!!!)" if tf.config.list_physical_devices("GPU") else "not available :(")
```

```
GPU available (YESS!!!!)
```

```
# How many rounds should we get the model to look through the data?
NUM_EPOCHS = 100 #@param {type:"slider", min:10, max:100, step:10}
```

NUM_EPOCHS: 100

Boom! We've got a GPU running and `NUM_EPOCHS` setup. Let's create a simple function which trains a model. The function will:

- Create a model using `create_model()`.
- Setup a TensorBoard callback using `create_tensorboard_callback()` (we do this here so it creates a log directory of the current date and time).
- Call the `fit()` function on our model passing it the training data, validatation data, number of epochs to train for and the callbacks we'd like to use.
- Return the fitted model.

```
# 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 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,
            epochs=NUM_EPOCHS,
            validation_data=val_data,
```

```
            validation_freq=1, # check validation metrics every epoch
            callbacks=[tensorboard, early_stopping])

  return model
```

Note: When training a model for the first time, the first epoch will take a while to load compared to the rest. This is because the model is getting ready and the data is being initialised. Using more data will generally take longer, which is why we've started with ~1000 images. After the first epoch, subsequent epochs should take a few seconds.

```
# Fit the model to the data
model = train_model()
```

```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Train for 25 steps, validate for 7 steps
Epoch 1/100
25/25 [==============================] - 164s 7s/step - loss: 4.5284 - accuracy: 0.1112 - val_loss: 3.4
Epoch 2/100
25/25 [==============================] - 5s 206ms/step - loss: 1.5946 - accuracy: 0.7000 - val_loss: 2.
Epoch 3/100
25/25 [==============================] - 5s 210ms/step - loss: 0.5552 - accuracy: 0.9337 - val_loss: 1.
Epoch 4/100
25/25 [==============================] - 5s 215ms/step - loss: 0.2469 - accuracy: 0.9900 - val_loss: 1.
Epoch 5/100
25/25 [==============================] - 6s 220ms/step - loss: 0.1447 - accuracy: 0.9962 - val_loss: 1.
Epoch 6/100
25/25 [==============================] - 5s 200ms/step - loss: 0.0986 - accuracy: 1.0000 - val_loss: 1.
Epoch 7/100
25/25 [==============================] - 5s 210ms/step - loss: 0.0744 - accuracy: 1.0000 - val_loss: 1.
Epoch 8/100
25/25 [==============================] - 5s 207ms/step - loss: 0.0592 - accuracy: 1.0000 - val_loss: 1.
Epoch 9/100
25/25 [==============================] - 5s 211ms/step - loss: 0.0486 - accuracy: 1.0000 - val_loss: 1.
Epoch 10/100
25/25 [==============================] - 5s 217ms/step - loss: 0.0409 - accuracy: 1.0000 - val_loss: 1.
Epoch 11/100
25/25 [==============================] - 5s 214ms/step - loss: 0.0354 - accuracy: 1.0000 - val_loss: 1.
Epoch 12/100
25/25 [==============================] - 5s 215ms/step - loss: 0.0309 - accuracy: 1.0000 - val_loss: 1.
Epoch 13/100
25/25 [==============================] - 5s 214ms/step - loss: 0.0270 - accuracy: 1.0000 - val_loss: 1.
```

Question: It looks like our model might be overfitting (getting far better results on the training set than the validation set), what are some ways to prevent model overfitting? Hint: this may involve searching something like "ways to prevent overfitting in a deep learning model?".

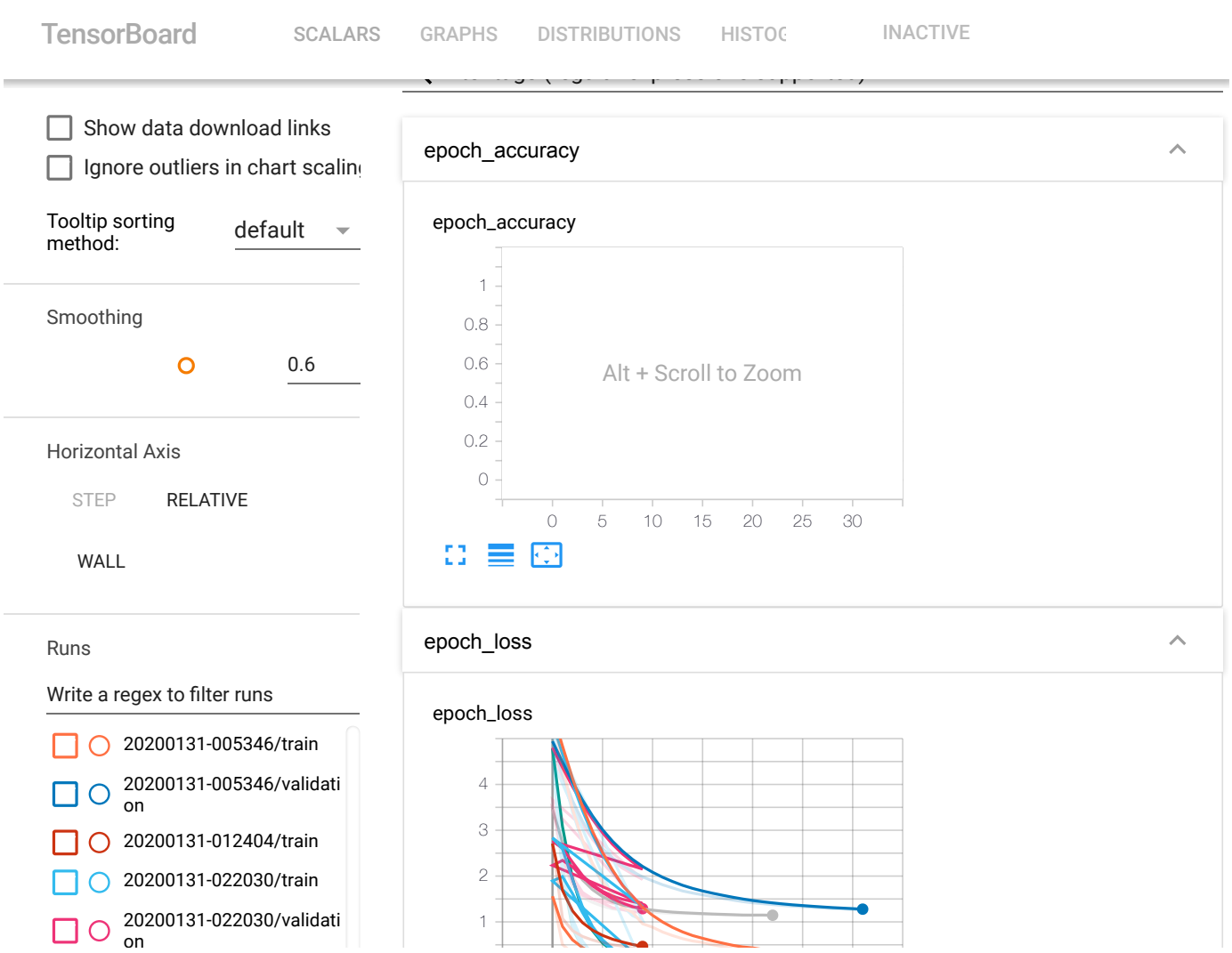
Note: Overfitting to begin with is a good thing. It means our model is learning something.

Checking the TensorBoard logs

Now our model has been trained, we can make its performance visual by checking the TensorBoard logs.

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

```
%tensorboard --logdir drive/My\ Drive/Data/logs
```



Thanks to our `early_stopping` callback, the model stopped training after 26 or so epochs (in my case, yours might be slightly different). This is because the validation accuracy failed to improve for 3 epochs.

But the good new is, we can definitely see our model is learning something. The validation accuracy got to 65% in only a few minutes.

This means, if we were to scale up the number of images, hopefully we'd see the accuracy increase.

Making and evaluating predictions using a trained model

Before we scale up and train on more data, let's see some other ways we can evaluate our model. Because although accuracy is a pretty good indicator of how our model is doing, it would be even better if we could could see it in action.

Making predictions with a trained model is as calling `predict()` on it and passing it data in the same format the model was trained on.

```
# Make predictions on the validation data (not used to train on)
predictions = model.predict(val_data, verbose=1) # verbose shows us how long there is to go
predictions
```

```
7/7 [==============================] - 2s 245ms/step
array([[6.84443337e-04, 3.89031629e-05, 1.09281263e-03, ...,
        7.02347374e-04, 2.43487393e-05, 2.55019218e-03],
       [5.12312958e-03, 1.39544054e-03, 6.85423985e-03, ...,
        4.46801248e-04, 1.17600709e-03, 1.40880875e-04],
```

```
       [1.08530430e-05, 7.95329834e-06, 7.72561236e-07, ...,
        2.33805578e-04, 1.15687035e-05, 1.69692139e-04],
       ...,
       [2.49893637e-05, 1.02584760e-04, 7.39373490e-06, ...,
        1.44409234e-04, 4.53142362e-04, 1.40150281e-04],
       [2.12867167e-02, 4.90643026e-04, 4.24099126e-04, ...,
        5.92991346e-05, 1.07101347e-04, 1.37331029e-02],
       [2.32493461e-04, 1.73809931e-05, 6.52848138e-03, ...,
        2.76323059e-03, 5.98608633e-04, 2.99575768e-04]], dtype=float32)
```

```
# Check the shape of predictions
predictions.shape
```

```
(200, 120)
```

Making predictions with our model returns an array with a different value for each label.

In this case, making predictions on the validation data (200 images) returns an array (`predictions`) of arrays, each containing 120 different values (one for each unique dog breed).

These different values are the probabilities or the likelihood the model has predicted a certain image being a certain breed of dog. The higher the value, the more likely the model thinks a given image is a specific breed of dog.

Let's see how we'd convert an array of probabilities into an actual label.

```
# First prediction
print(predictions[0])
print(f"Max value (probability of prediction): {np.max(predictions[0])}") # the max probability value predic
print(f"Sum: {np.sum(predictions[0])}") # because we used softmax activation in our model, this will be clos
print(f"Max index: {np.argmax(predictions[0])}") # the index of where the max value in predictions[0] occurs
print(f"Predicted label: {unique_breeds[np.argmax(predictions[0])]}") # the predicted label
```

```
[6.84443337e-04 3.89031629e-05 1.09281263e-03 2.91300385e-05
 8.03185161e-04 3.14614190e-05 2.33187284e-02 5.58101106e-04
 4.16126044e-04 3.28170019e-03 1.35926093e-04 9.96642120e-05
 4.73195134e-04 2.47105840e-04 1.29635184e-04 7.56074558e-04
 4.81268944e-05 1.00964129e-01 9.26381254e-05 4.70037921e-05
 1.43714307e-03 6.96299714e-04 8.81163578e-06 5.72976307e-04
 1.18002768e-04 9.83200734e-05 5.02589107e-01 8.00870985e-05
 2.11806037e-03 1.36482689e-04 9.34780255e-05 1.24260201e-03
 4.07848653e-04 2.67286778e-05 1.97614994e-04 6.04105648e-03
 6.67608629e-06 8.51864461e-05 5.20519789e-05 4.85751196e-04
 2.61472975e-04 3.68152541e-05 2.29220561e-04 3.62329069e-04
 3.05753958e-04 1.44591046e-04 5.85637026e-05 1.03238519e-04
 5.93333913e-04 3.17924394e-04 2.05869379e-04 5.33956263e-05
 4.26513521e-04 2.83966310e-05 8.56074912e-05 4.43779099e-05
 3.80086247e-04 1.38787611e-03 7.50362815e-04 1.20276995e-01
 1.14553084e-04 5.97622966e-05 2.14516232e-03 3.66680833e-05
 2.15207366e-03 1.37935337e-02 1.26239102e-04 3.54231859e-04
 7.54820276e-03 1.65769859e-04 1.95510630e-02 6.39963080e-04
 2.68232485e-04 7.74671184e-03 6.38541707e-04 4.04078193e-04
 1.09590753e-03 5.07419137e-03 4.77885478e-04 6.61956379e-03
 2.33302912e-04 3.62189161e-03 3.46420507e-04 9.73619334e-03
 8.39721179e-05 2.41281843e-04 3.25776462e-04 2.41334690e-03
 4.55477311e-05 4.47568425e-04 4.02137754e-04 3.72363429e-04
 5.85472526e-06 3.91939480e-04 1.26994870e-04 1.23371472e-04
 3.37114820e-04 1.62539014e-03 7.63085845e-04 1.14508301e-04
 1.48690874e-02 1.13483999e-04 2.96788476e-02 1.56830736e-02
 4.94542124e-04 1.95087283e-04 3.69213894e-02 1.60986834e-04
 6.81152378e-05 2.63732225e-02 1.41710660e-03 2.77160842e-04
 8.88329578e-06 1.13954666e-04 4.66341386e-04 1.78232149e-05
 3.29842418e-03 7.02347374e-04 2.43487393e-05 2.55019218e-03]
Max value (probability of prediction): 0.5025891065597534
Sum: 0.9999998807907104
Max index: 26
Predicted label: cairn
```

Having this information is great but it would be even better if we could compare a prediction to its true label and original image.

To help us, let's first build a little function to convert prediction probabilities into predicted labels.

Note: Prediction probabilities are also known as confidence levels.

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

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

```
'cairn'
```

Wonderful! Now we've got a list of all different predictions our model has made, we'll do the same for the validation images and validation labels.

Remember, the model hasn't trained on the validation data, during the `fit()` function, it only used the validation data to evaluate itself. So we can use the validation images to visually compare our models predictions with the validation labels.

Since our validation data (`val_data`) is in batch form, to get a list of validation images and labels, we'll have to unbatch it (using `unbatch()`) and then turn it into an iterator using `as_numpy_iterator()`.

Let's make a small function to do so.

```
# Create a function to unbatch a batched 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(unique_breeds[np.argmax(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')
```

Nailed it!

Now we've got ways to get:

- Prediction labels
- Validation labels (truth labels)
- Validation images

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

More specifically, we want to be able to view an image, its predicted label and its actual label (true label).

The first function we'll create will:

- Take an array of prediction probabilities, an array of truth labels, 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 target image on a single plot.

```
def plot_pred(prediction_probabilities, labels, images, n=1):
  """
  View the prediction, ground truth label 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 color of the title depending on if the prediction is right or wrong
  if pred_label == true_label:
    color = "green"
  else:
    color = "red"

  plt.title("{} {:2.0f}% ({})".format(pred_label,
                                      np.max(pred_prob)*100,
                                      true_label),
                                      color=color)
```

```
# View an example prediction, original image and truth label
plot_pred(prediction_probabilities=predictions,
          labels=val_labels,
          images=val_images)
```

Nice! Making functions to help visual your models results are really helpful in understanding how your model is doing.

Since we're working with a multi-class problem (120 different dog breeds), it would also be good to see what other guesses our model is making. More specifically, if our model predicts a certain label with 24% probability, what else did it predict?

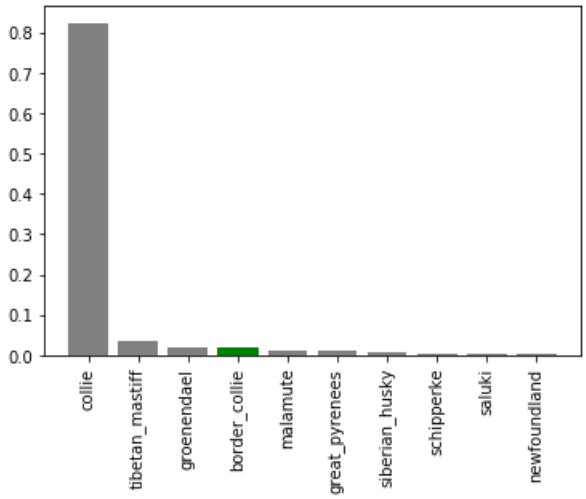
Let's build a function to demonstrate. The function will:

- Take an input of a prediction probabilities array, a ground truth labels array and an integer.
- Find the predicted label 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 green.

```
def plot_pred_conf(prediction_probabilities, labels, n=1):
  """
  Plots 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
```

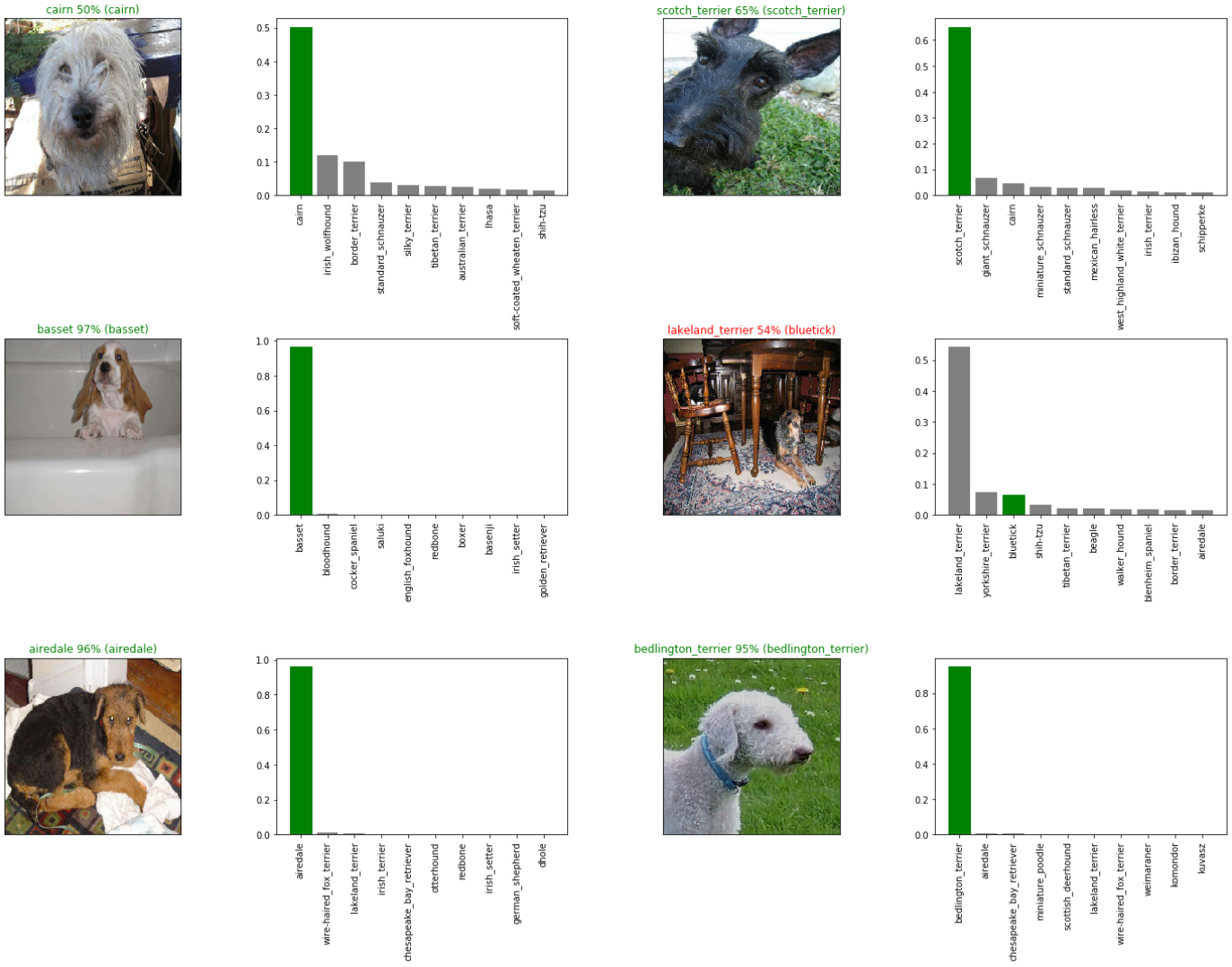
```
plot_pred_conf(prediction_probabilities=predictions,
               labels=val_labels,
               n=9)
```



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

```
# Let's check a few predictions and their different values
i_multiplier = 0
num_rows = 3
num_cols = 2
num_images = num_rows*num_cols
plt.figure(figsize=(5*2*num_cols, 5*num_rows))
for i in range(num_images):
  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)
  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()
```



Saving and reloading a model

After training a model, it's a good idea to save it. Saving it means you can share it with colleagues, put it in an application and more importantly, won't have to go through the potentially expensive step of retraining it.

The format of an entire saved Keras model is h5. So we'll make a function which can take a model as input and utilise the `save()` method to save it as a h5 file to a specified directory.

```
def save_model(model, suffix=None):
  """
  Saves a given model in a models directory and appends a suffix (str)
  for clarity and reuse
```

```
  for clarity and reuse.
  """
  # Create model directory with current time
  modeldir = os.path.join("drive/My Drive/Data/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
```

If we've got a saved model, we'd like to load it, let's create a function which can take a model path and use the `tf.keras.models.load_model()` function to load it into the notebook.

Because we're using a component from TensorFlow Hub (`hub.KerasLayer`) we'll have to pass this as a parameter to the `custom_objects` parameter.

```
def load_model(model_path):
  """
  Loads 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
```

```
# Save our model trained on 1000 images
save_model(model, suffix="1000-images-Adam")
```

```
Saving model to: drive/My Drive/Data/models/20200202-06371580625449-1000-images-Adam.h5...
'drive/My Drive/Data/models/20200202-06371580625449-1000-images-Adam.h5'
```

```
# Load our model trained on 1000 images
model_1000_images = load_model('drive/My Drive/Data/models/20200131-02551580439347-1000-images-Adam.h5')
```

```
Loading saved model from: drive/My Drive/Data/models/20200131-02551580439347-1000-images-Adam.h5
WARNING:tensorflow:Sequential models without an `input_shape` passed to the first layer cannot reload t
WARNING:tensorflow:Sequential models without an `input_shape` passed to the first layer cannot reload t
```

Compare the two models (the original one and loaded one). We can do so easily using the `evaluate()` method.

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

```
7/7 [==============================] - 1s 128ms/step - loss: 1.1389 - accuracy: 0.7000
[1.1388890828405107, 0.7]
```

```
# Evaluate the loaded model
model_1000_images.evaluate(val_data)
```

```
7/7 [==============================] - 1s 198ms/step - loss: 1.1389 - accuracy: 0.7000
[1.1388890828405107, 0.7]
```

Training a model (on the full data)

Now we know our model works on a subset of the data, we can start to move forward with training one on the full data.

Above, we saved all of the training filepaths to `X` and all of the training labels to `y`. Let's check them out.

```
# Remind ourselves of the size of the full dataset
len(X), len(y)
```

```
(10222, 10222)
```

There we go! We've got over 10,000 images and labels in our training set.

Before we can train a model on these, we'll have to turn them into a data batch.

The beautiful thing is, we can use our `create_data_batches()` function from above which also preprocesses our images for us (thank you past us for writing a helpful function).

```
# Turn full training data in a data batch
full_data = create_data_batches(X, y)
```

```
Creating training data batches...
```

Our data is in a data batch, all we need now is a model.

And surprise, we've got a function for that too! Let's use `create_model()` to instantiate another model.

```
# Instantiate a new model for training on the full dataset
full_model = create_model()
```

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

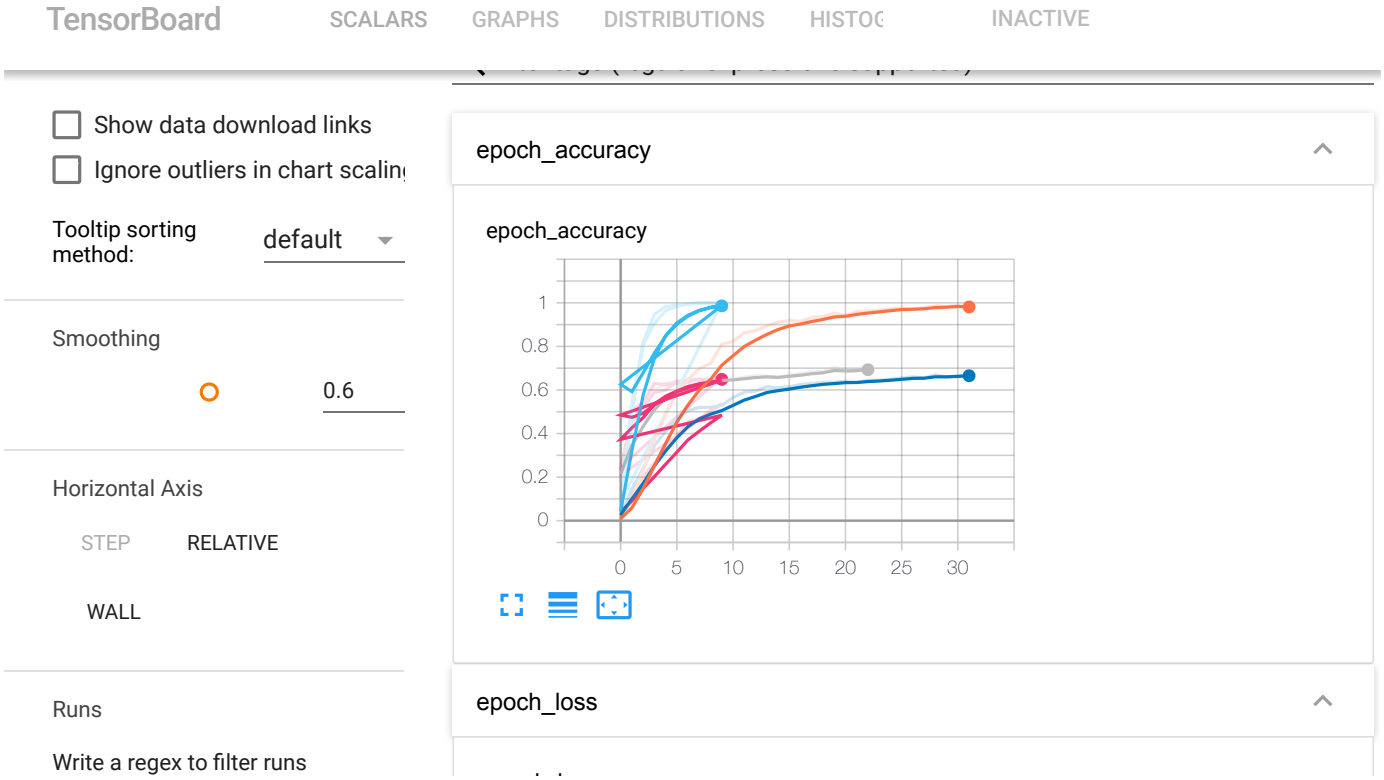
Since we've made a new model instance, `full_model`, we'll need some callbacks too.

```
# Create full model callbacks

# TensorBoard callback
full_model_tensorboard = create_tensorboard_callback()

# Early stopping callback
# Note: No validation set when training on all the data, therefore can't monitor validation accruacy
full_model_early_stopping = tf.keras.callbacks.EarlyStopping(monitor="accuracy",
                                                             patience=3)
```

To monitor the model whilst it trains, we'll load TensorBoard (it should update every 30-seconds or so whilst the model trains).

```
%tensorboard --logdir drive/My\ Drive/Data/logs
```

```
Reusing TensorBoard on port 6006 (pid 1855), started 1:55:46 ago. (Use '!kill 1855' to kill it.)
```



Note: Since running the cell below will cause the model to train on all of the data (10,000+) images, it may take a fairly long time to get started and finish. However, thanks to our `full_model_early_stopping` callback, it'll stop before it starts going too long.

Remember, the first epoch is always the longest as data gets loaded into memory. After it's there, it'll speed up.

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

```
Train for 320 steps
Epoch 1/100
320/320 [==============================] - 55s 171ms/step - loss: 1.5578 - accuracy: 0.6161
Epoch 2/100
320/320 [==============================] - 53s 165ms/step - loss: 0.5033 - accuracy: 0.8446
Epoch 3/100
320/320 [==============================] - 52s 163ms/step - loss: 0.3268 - accuracy: 0.8990
Epoch 4/100
320/320 [==============================] - 53s 165ms/step - loss: 0.2429 - accuracy: 0.9254
Epoch 5/100
320/320 [==============================] - 51s 160ms/step - loss: 0.1810 - accuracy: 0.9477
Epoch 6/100
320/320 [==============================] - 51s 158ms/step - loss: 0.1447 - accuracy: 0.9594
Epoch 7/100
320/320 [==============================] - 53s 165ms/step - loss: 0.1209 - accuracy: 0.9652
Epoch 8/100
320/320 [==============================] - 55s 172ms/step - loss: 0.0972 - accuracy: 0.9731
Epoch 9/100
320/320 [==============================] - 52s 162ms/step - loss: 0.0813 - accuracy: 0.9804
Epoch 10/100
320/320 [==============================] - 52s 162ms/step - loss: 0.0698 - accuracy: 0.9838
Epoch 11/100
320/320 [==============================] - 51s 159ms/step - loss: 0.0652 - accuracy: 0.9830
Epoch 12/100
320/320 [==============================] - 51s 161ms/step - loss: 0.0602 - accuracy: 0.9839
```

```
Epoch 13/100
320/320 [==============================] - 51s 159ms/step - loss: 0.0553 - accuracy: 0.9867
Epoch 14/100
320/320 [==============================] - 52s 161ms/step - loss: 0.0500 - accuracy: 0.9885
Epoch 15/100
320/320 [==============================] - 52s 163ms/step - loss: 0.0516 - accuracy: 0.9868
Epoch 16/100
320/320 [==============================] - 52s 162ms/step - loss: 0.0467 - accuracy: 0.9882
Epoch 17/100
320/320 [==============================] - 51s 160ms/step - loss: 0.0482 - accuracy: 0.9872
<tensorflow.python.keras.callbacks.History at 0x7f2e4f1bcd30>
```

Saving and reloading the full model

Even on a GPU, our full model took a while to train. So it's a good idea to save it.

We can do so using our `save_model()` function.

Challenge: It may be a good idea to incorporate the `save_model()` function into a `train_model()` function. Or look into setting up a checkpoint callback.

```
# Save model to file
save_model(full_model, suffix="all-images-Adam")
```

```
Saving model to: drive/My Drive/Data/models/20200131-03111580440309-all-images-Adam.h5...
'drive/My Drive/Data/models/20200131-03111580440309-all-images-Adam.h5'
```

```
# Load in the full model
loaded_full_model = load_model('drive/My Drive/Data/models/20200131-03111580440309-all-images-Adam.h5')
```

```
Loading saved model from: drive/My Drive/Data/models/20200131-03111580440309-all-images-Adam.h5
WARNING:tensorflow:Sequential models without an `input_shape` passed to the first layer cannot reload t
WARNING:tensorflow:Sequential models without an `input_shape` passed to the first layer cannot reload t
```

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 there are no labels with the test images).
- Make a predictions array by passing the test data batches to the `predict()` function.

```
# Load test image filenames (since we're using os.listdir(), these already have .jpg)
test_path = "drive/My Drive/Data/test/"
test_filenames = [test_path + fname for fname in os.listdir(test_path)]

test_filenames[:10]
```

```
['drive/My Drive/Data/test/dd39ff72939c0445a76f0e7db9985f56.jpg',
 'drive/My Drive/Data/test/e0f9b92adbba451d296678f466732969.jpg',
 'drive/My Drive/Data/test/dcd3cfe0cd6d363a3ed21639c434c8d3.jpg',
 'drive/My Drive/Data/test/de15c21ba0a1f139d26223d5e2b09703.jpg',
 'drive/My Drive/Data/test/def88eeacc633cbc3d46d5f5fb495379.jpg',
 'drive/My Drive/Data/test/e57a2bee790b512b2ba824b26f3f93fd.jpg',
```

```
 'drive/My Drive/Data/test/e13f3871a8b4a745717ba6903f0dfe05.jpg',
 'drive/My Drive/Data/test/e0d85c8c8730c81e80ff2ca74bb61c87.jpg',
 'drive/My Drive/Data/test/de5496c6b58f66ab890ab24087d9e220.jpg',
 'drive/My Drive/Data/test/e427b9e1ab1b7f09cfb02ac073f56f2d.jpg']
```

```
# How many test images are there?
len(test_filenames)
```

```
10357
```

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

```
Creating test data batches...
```

Note: Since there are 10,000+ test images, making predictions could take a while, even on a GPU. So beware running the cell below may take up to an hour.

```
# Make predictions on test data batch using the loaded full model
test_predictions = loaded_full_model.predict(test_data,
                                             verbose=1)
```

```
324/324 [==============================] - 2934s 9s/step
```

```
# Check out the test predictions
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]], dtype=float32)
```

Preparing test dataset predictions for Kaggle

Looking at the Kaggle sample submission, it looks like they want the models output probabilities each for label along with the image ID's.

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 using the `unique_breeds` list and the `test_predictions` list.
- Export the DataFrame as a CSV to submit it to Kaggle.

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

id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffordshire_terrier	appen

0 rows × 121 columns

```
# Append test image ID's to predictions DataFrame
test_path = "drive/My Drive/Data/test/"
preds_df["id"] = [os.path.splitext(path)[0] for path in os.listdir(test_path)]
preds_df.head()
```

	id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_
0	dd39ff72939c0445a76f0e7db9985f56	NaN	NaN	NaN	NaN	
1	e0f9b92adbba451d296678f466732969	NaN	NaN	NaN	NaN	
2	dcd3cfe0cd6d363a3ed21639c434c8d3	NaN	NaN	NaN	NaN	
3	de15c21ba0a1f139d26223d5e2b09703	NaN	NaN	NaN	NaN	
4	def88eeacc633cbc3d46d5f5fb495379	NaN	NaN	NaN	NaN	

5 rows × 121 columns

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

	id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_
0	dd39ff72939c0445a76f0e7db9985f56	1.61196e-09	3.44086e-12	2.32834e-11	1.40799e-12	
1	e0f9b92adbba451d296678f466732969	3.17894e-10	3.20088e-14	1.85375e-10	1.95466e-13	
2	dcd3cfe0cd6d363a3ed21639c434c8d3	4.27301e-09	1.8414e-13	1.11785e-09	1.10664e-10	
3	de15c21ba0a1f139d26223d5e2b09703	5.22772e-14	3.37155e-14	1.63818e-11	5.28355e-12	
4	def88eeacc633cbc3d46d5f5fb495379	9.61375e-11	1.30428e-08	1.24693e-08	1.47886e-07	

5 rows × 121 columns

Boom! Let's now export our predictions DataFrame to CSV so we can submit it to Kaggle.

```
preds_df.to_csv("drive/My Drive/Data/full_submission_1_mobilienetV2_adam.csv",
                index=False)
```

Making predictions on custom images

It's great being able to make predictions on a test dataset already provided for us.

But how could we use our model on our own images?

The premise remains, if we want to make predictions on our own custom images, we have to pass them to the model in the same format the model was trained on.

To do so, we'll:

- Get the filepaths of our own images.
- Turn the filepaths into data batches using `create_data_batches()`. And since our custom images won't have labels, we set the `test_data` parameter to `True`.
- Pass the custom image data batch to our model's `predict()` method.
- Convert the prediction output probabilities to prediction labels.
- Compare the predicted labels to the custom images.

Note: To make predictions on custom images, I've uploaded pictures of my own to a directory located at `drive/My Drive/Data/dogs/` (as seen in the cell below). In order to make predictions on your own images, you will have to do

```
# Get custom image filepaths
custom_path = "drive/My Drive/Data/dogs/"
custom_image_paths = [custom_path + fname for fname in os.listdir(custom_path)]
```

```
# Turn custom image into batch (set to test data because there are no labels)
custom_data = create_data_batches(custom_image_paths, test_data=True)
```

```
Creating test data batches...
```

```
# Make predictions on the custom data
custom_preds = loaded_full_model.predict(custom_data)
```

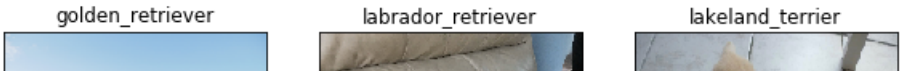
Now we've got some predictions arrays, let's convert them to labels and compare them with each image.

```
# Get custom image prediction labels
custom_pred_labels = [get_pred_label(custom_preds[i]) for i in range(len(custom_preds))]
custom_pred_labels
```

```
['golden_retriever', 'labrador_retriever', 'lakeland_terrier']
```

```
# Get custom images (our unbatchify() function won't work since there aren't labels)
custom_images = []
# Loop through unbatched data
for image in custom_data.unbatch().as_numpy_iterator():
  custom_images.append(image)
```

```
# Check custom image predictions
plt.figure(figsize=(10, 10))
for i, image in enumerate(custom_images):
  plt.subplot(1, 3, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.title(custom_pred_labels[i])
  plt.imshow(image)
```



What's next?

Woah! What an effort. If you've made it this far, you've just gone end-to-end on a multi-class image classification problem.

This is the same style of problem self-driving cars have, except with different data.

If you're looking on where to go next, you've got plenty of options.

You could try to improve the full model we trained in this notebook in a few ways (there are a fair few options). Since our early experiment (using only 1000 images) hinted at our model overfitting (the results on the training set far outperformed the results on the validation set), one goal going forward would be to try and prevent it.

1. Trying another model from TensorFlow Hub - Perhaps a different model would perform better on our dataset. One option would be to experiment with a different pretrained model from TensorFlow Hub or look into the `tf.keras.applications` module.
2. Data augmentation - Take the training images and manipulate (crop, resize) or distort them (flip, rotate) to create even more training data for the model to learn from. Check out the TensorFlow images documentation for a whole bunch of functions you can use on images. A great idea would be to try and replicate the techniques in this example cat vs. dog image classification notebook for our dog breeds problem.
3. Fine-tuning - The model we used in this notebook was directly from TensorFlow Hub, we took what it had already learned from another dataset (ImageNet) and applied it to our own. Another option is to use what the model already knows and fine-tune this knowledge to our own dataset (pictures of dogs). This would mean all of the patterns within the model would be updated to be more specific to pictures of dogs rather than general images.

If you're ever after more, one of the best ways to find out something is to search for something like:

- "How to improve a TensorFlow 2.x image classification model?"
- "TensorFlow 2.x image classification best practices"
- "Transfer learning for image classification with TensorFlow 2.x"

And when you see an example you think might be beyond your reach (because it looks too complicated), remember, if in doubt, run the code. Try and reproduce what you see. This is the best way to get hands-on and build your own knowledge.

No one starts out knowing how to do everything single thing. They just get better are knowing what to look for.

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.