End-to-End-Multiclass_Dog-Breed-Classification

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

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.

2. Data

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

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

3. Evaluation

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

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

4. Features

Some imformation 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 there are 120 different classes).
- There are around 10000+ images in the training set (these images have labels).
- There are around 10000+ 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/"
```

Get our workspace ready

- Import TensorFlow 2.x
- Import TensorFLow Hub
- Make sure we're using a GPU

```
# Import TensorFlow, TensorFlow Hub into Colab
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 (YESSS!!!!!)" if tf.config.list_physical_devices("GPU") else "not available :(")
```

```
TF Version: 2.3.0
TF Hub Version: 0.9.0
GPU available (YESSS!!!!!)
```

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).

Let's start by accessing our data and checking out the labels.

```
# Checkout 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     b38b639670034f8ef9bd87c17ce29b56  scottish_deerhound
freq                                   1                 126
                                 id          breed
0  000bec180eb18c7604dcecc8fe0dba07    boston_bull
1  001513dfcb2ffafc82cccf4d8bbaba97          dingo
2  001cdf01b096e06d78e9e5112d419397       pekinese
3  00214f311d5d2247d5dfe4fe24b2303d       bluetick
```

```
labels_csv.head()
```

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

```
# How many images are there of each breed?
labels_csv["breed"].value_counts()
```

```
    scottish_deerhound      126
```

```
labels_csv.breed.value_counts().plot.bar(figsize=(20,10))
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f50214689b0>
```


```python
labels_csv["breed"].value_counts().median()
```

```
82.0
```

```python
# Let's view an image
from IPython.display import Image
Image("drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg")
```

Getting Images and their labels

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

```
labels_csv.head()
```

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

```
# Create pathnames from Image ID's
# List Comprehensions
filenames = ["drive/My Drive/Dog Vision/train/" + fname + ".jpg" for fname in labels_csv["id"]]
# Check the first 10
filenames[:10]
```

```
['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']
```

```
import os
print(len(os.listdir("drive/My Drive/Dog Vision/train/")))
# print(len(os.listdir("drive/My Drive/Dog Vision/test/")))
```

```
10222
```

```
# Check whether no.of filename matches no. of actual image files
if len(os.listdir("drive/My Drive/Dog Vision/train/")) == len(filenames):
  print("Filenames match actual amount of files!!! Proceed")
else:
  print("Filenames do not match actual amount of files, check the target directory.")
```

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

```
# 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.

```
import numpy as np
labels = labels_csv["breed"].to_numpy()
# labels = np.array(labels) # does same thing as above
labels
```

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

Here, labels are numpy arrays storing the value of breeds. Unique Breeds contains 120 data.

```
len(labels)
```

```
10222
```

```
# See if number of  labels matches the number of filenames
if len(labels) == len(filenames):
  print("No. of labels matches no. of filenames")
else:
  print("No. of labels does not match no. of filenames, check the directories!")
```

```
  print("No. of labels does not match no. of filenames, check the directories!")
```

```
No. of labels matches no. of filenames
```

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

```
array(['affenpinscher', 'afghan_hound', 'african_hunting_dog', 'airedale',
       'american_staffordshire_terrier', 'appenzeller',
       'australian_terrier', 'basenji', 'basset', 'beagle',
       'bedlington_terrier', 'bernese_mountain_dog',
```

```
len(unique_breeds)
```

```
120
       'cairn', 'cardigan', 'chesapeake_bay_retriever', 'chihuahua',
```

```
# 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])
       'shetland_sheepdog', 'shih-tzu', 'siberian_husky', 'silky_terrier',
```

```
# Turn every label into a boolean array
# List Comprehension
boolean_labels = [label == 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,
```

```
len(boolean_labels)
```

```
10222
```

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

```
boston_bull
(array([19]),)
```

```
print(labels[2])
print(np.where(unique_breeds == labels[2]))
print(boolean_labels[2].argmax())
print(boolean_labels[2].astype(int))
```

```
pekinese
(array([85]),)
85
[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 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]
```

Boolean Labels are in the form of True(1) and False(0). At the point, where unique_breed matches the labels(numpy_array of breeds), it shows 1.

Filepaths corresponds to the train data contains images filepaths.

```
filenames[:10]
```

```
['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']
```

```
len(filenames)
```

```
10222
```

Creating our own validation set

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

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

X contains filepaths of all 10222 images in the train folder. y boolean labels in the form of True and False of all 10222 items corresponding to the result out by comparing unique_breeds == labels[:]

```
len(X), len(y)
```

```
(10222, 10222)
```

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

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

NUM_IMAGES: 1000

```
# Let's split our data into train & validation sets
from sklearn.model_selection import train_test_split

# np.random.seed(42)

# Split them into training & 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(y_train), len(X_val), len(y_val)
```

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

```
# Let's have a geez at the training data
X_train[:5], 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'],
 [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])])
```

Preprocessing Images (turning images into tensors)

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

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 0-255 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.

```
# Convert image to Numpy Array
from matplotlib.pyplot import imread
image = imread(filenames[42])
image.shape
```

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

```
image.max(), image.min()
```

```
(255, 0)
```

```
image[:2] # converted into numpy array in the range (0-255)
```

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

```
# 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 0-255 to 0-1).
5. Resize the `image` to be a shape of (224,224).
6. Return the modified `image`.

```
# Define image size
IMG_SIZE = 224

# Create a function for preprocessing images
def process_image(image_path, img_size=IMG_SIZE):
  """
  Takes an image file path and turns the image into a Tensor.
  """
  # Read in an 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 to 255 to 0-1 values (Normalization)
  image = tf.image.convert_image_dtype(image, tf.float32)

  # Resize the image to our desired value (224,224)
  image = tf.image.resize(image, size=[IMG_SIZE,IMG_SIZE])

  return image


# tensor = tf.io.read_file(filenames[26])
# tensor
# tensor = tf.image.decode_jpeg(tensor, channels=3)
# tensor
# tf.image.convert_image_dtype(tensor, tf.float32)
```

Turning our data into batches

Why turn our data into batches?

Let's say you're trying to process 10000+ 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 needed).

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

```
# 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 above
process_image(X[42]), tf.constant(y[42])
```

```
(<tf.Tensor: shape=(224, 224, 3), dtype=float32, numpy=
 array([[[0.3264178 , 0.5222886 , 0.3232816 ],
         [0.2537167 , 0.44366494, 0.24117757],
         [0.25699762, 0.4467087 , 0.23893751],
         ...,
         [0.29325107, 0.5189916 , 0.3215547 ],
         [0.29721776, 0.52466875, 0.33030328],
         [0.2948505 , 0.5223015 , 0.33406618]],

        [[0.25903144, 0.4537807 , 0.27294815],
         [0.24375686, 0.4407019 , 0.2554778 ],
         [0.2838985 , 0.47213382, 0.28298813],
         ...,
         [0.2785345 , 0.5027992 , 0.31004712],
         [0.28428748, 0.5108719 , 0.32523635],
         [0.28821915, 0.5148036 , 0.32916805]],

        [[0.20941195, 0.40692952, 0.25792548],
         [0.24045378, 0.43900946, 0.2868911 ],
         [0.29001117, 0.47937486, 0.32247734],
         ...,
         [0.26074055, 0.48414773, 0.30125174],
         [0.27101526, 0.49454468, 0.32096273],
         [0.27939945, 0.5029289 , 0.32934693]],

        ...,

        [[0.00634795, 0.03442048, 0.0258106 ],
         [0.01408936, 0.04459917, 0.0301715 ],
         [0.01385712, 0.04856448, 0.02839671],
         ...,
         [0.4220516 , 0.39761978, 0.21622123],
         [0.47932503, 0.45370543, 0.2696505 ],
         [0.48181024, 0.45828083, 0.27004552]],

        [[0.00222061, 0.02262166, 0.03176915],
         [0.01008397, 0.03669046, 0.02473482],
         [0.00608852, 0.03890046, 0.01207283],
         ...,
         [0.36070833, 0.33803678, 0.16216145],
         [0.42499566, 0.3976801 , 0.21701711],
         [0.4405433 , 0.4139589 , 0.23183356]],
```

```
       [[0.05608025, 0.06760229, 0.10401428],
```

```
get_image_label(X[42],y[42])
```

```
(<tf.Tensor: shape=(224, 224, 3), dtype=float32, numpy=
 array([[[0.3264178 , 0.5222886 , 0.3232816 ],
         [0.2537167 , 0.44366494, 0.24117757],
         [0.25699762, 0.4467087 , 0.23893751],
         ...,
         [0.29325107, 0.5189916 , 0.3215547 ],
         [0.29721776, 0.52466875, 0.33030328],
         [0.2948505 , 0.5223015 , 0.33406618]],

        [[0.25903144, 0.4537807 , 0.27294815],
         [0.24375686, 0.4407019 , 0.2554778 ],
         [0.2838985 , 0.47213382, 0.28298813],
         ...,
         [0.2785345 , 0.5027992 , 0.31004712],
         [0.28428748, 0.5108719 , 0.32523635],
         [0.28821915, 0.5148036 , 0.32916805]],

        [[0.20941195, 0.40692952, 0.25792548],
         [0.24045378, 0.43900946, 0.2868911 ],
         [0.29001117, 0.47937486, 0.32247734],
         ...,
         [0.26074055, 0.48414773, 0.30125174],
         [0.27101526, 0.49454468, 0.32096273],
         [0.27939945, 0.5029289 , 0.32934693]],

        ...,

        [[0.00634795, 0.03442048, 0.0258106 ],
```

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

```
         [0.47932503, 0.45370543, 0.2696505 ],
```

```
tf.constant(X)
```

```
<tf.Tensor: shape=(10222,), dtype=string, numpy=
array([b'drive/My Drive/Dog Vision/train/000bec180eb18c7604dcecc8fe0dba07.jpg',
       b'drive/My Drive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg',
       b'drive/My Drive/Dog Vision/train/001cdf01b096e06d78e9e5112d419397.jpg',
```

```
tf.constant(y)
```

```
<tf.Tensor: shape=(10222, 120), dtype=bool, numpy=
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, False, False, False, False, False, False, False, False,
```

```
# 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) & 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 valid data batches...")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), #filepaths
                                               tf.constant(y))) #labels
```

```
                                          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
```

```
# 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 valid 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)))
```

Visualizing Data Batches

Our data is now in batches. However, these can be a little hard to understand/comprehend. Let's visualize them.

```
import matplotlib.pyplot as plt

# Create a function for viewing images in a data batch

def show_25_images(images, labels):
  """
  Displays a plot of 25 images and their labels from a data batch.
  """
  # Setup the figure canvas
  plt.figure(figsize=(10,10))
  # Loop through 25 (for displaying 25 images)
  for i in range(25):
    # Create subplots (5 rows, 5 cols)
    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 gridlines off
    plt.axis("off")
```

```
unique_breeds[y[1].argmax()]
```

```
'dingo'
```

```
# Now let's visualize the data in a training batch
train_images, train_labels = next(train_data.as_numpy_iterator())
train_images, train_labels
```

```
(array([[[[0.12217762, 0.14543442, 0.12217762],
          [0.2789791 , 0.28736433, 0.2675758 ],
          [0.2817952 , 0.28464758, 0.26623273],
          ...,
          [0.32760072, 0.28352737, 0.29814452],
          [0.2628065 , 0.25383738, 0.25484714],
          [0.13860525, 0.13860525, 0.13860525]],

         [[0.5144871 , 0.54193807, 0.5144871 ],
          [0.91294146, 0.9311163 , 0.90896803],
          [0.98232603, 0.9829069 , 0.9666117 ],
          ...,
          [0.98968965, 0.9826944 , 0.98732305],
          [0.98462504, 0.98100185, 0.9843281 ],
          [0.5154031 , 0.5154031 , 0.5154031 ]],

         [[0.48133823, 0.5088592 , 0.4838405 ],
          [0.92955625, 0.9549764 , 0.9336179 ],
          [0.78012025, 0.80034584, 0.778252  ],
          ...,
          [0.7501372 , 0.77581364, 0.77171767],
          [0.93690974, 0.9403548 , 0.9386323 ],
          [0.52899915, 0.52899915, 0.52899915]],

         ...,

         [[0.52427566, 0.52427566, 0.52427566],
          [0.96441215, 0.96441215, 0.96441215],
          [0.9673422 , 0.9673422 , 0.9673422 ],
          ...,
          [0.97467244, 0.97467244, 0.97467244],
          [0.9319122 , 0.9319122 , 0.9319122 ],
          [0.52390796, 0.52390796, 0.52390796]],

         [[0.49462616, 0.49462616, 0.49462616],
          [0.96435046, 0.96435046, 0.96435046],
          [0.98627913, 0.98627913, 0.98627913],
          ...,
          [0.96641815, 0.96641815, 0.96641815],
          [0.9815315 , 0.9815315 , 0.9815315 ],
          [0.53548616, 0.53548616, 0.53548616]],
```

```
 [[0.17167036, 0.17167036, 0.17167036],
  [0.2740346 , 0.2740346 , 0.2740346 ],
  [0.2569238 , 0.2569238 , 0.2569238 ],
  ...,
  [0.25032952, 0.25032952, 0.25032952],
  [0.24575163, 0.24575163, 0.24575163],
  [0.1320957 , 0.1320957 , 0.1320957 ]]],


[[[0.45882356, 0.00784314, 0.        ],
  [0.45882356, 0.00784314, 0.        ],
  [0.46698183, 0.00392157, 0.        ],
  ...,
  [0.71813107, 0.48336446, 0.19603841],
  [0.61185217, 0.37616345, 0.10283753],
  [0.5894692 , 0.35076097, 0.08649342]],

 [[0.45882356, 0.00784314, 0.        ],
  [0.4666667 , 0.00392157, 0.        ],
  [0.46698183, 0.00392157, 0.        ],
  ...,
  [0.71635175, 0.48962206, 0.19843686],
  [0.59263414, 0.35741425, 0.07498042],
  [0.6000774 , 0.36478326, 0.08719142]],

 [[0.4641982 , 0.00515581, 0.        ],
  [0.4666667 , 0.00392157, 0.        ],
  [0.4698851 , 0.00370562, 0.        ],
  ...,
  [0.62894195, 0.41790953, 0.12080703],
  [0.582178  , 0.3539444 , 0.06311981],
  [0.6116838 , 0.37742144, 0.08881194]],

 ...,

 [[0.48230022, 0.00365031, 0.01418067],
  [0.4854481 , 0.01468316, 0.02293418],
  [0.4788483 , 0.01547034, 0.021764  ],
  ...,
  [0.15032706, 0.17354135, 0.135271  ],
  [0.14735222, 0.16696008, 0.14343065],
  [0.17657036, 0.1961782 , 0.17264879]],
```

```
       [[0.4721477 , 0.01260448, 0.01805856],
        [0.4740086 , 0.01896892, 0.02293418],
        [0.4667174 , 0.01894093, 0.0189811 ],
        ...,
        [0.1639848 , 0.18719912, 0.14892875],
        [0.166347  , 0.18595485, 0.16242543],
        [0.21845603, 0.23806387, 0.21453445]],

       [[0.46717414, 0.0223038 , 0.02120954],
        [0.46245685, 0.02725037, 0.02352941],
        [0.45463043, 0.02082334, 0.01637773],
        ...,
        [0.21611968, 0.23933399, 0.20106362],
        [0.2060512 , 0.22565904, 0.20212963],
        [0.18236834, 0.20197618, 0.17844677]]],


      [[[0.6862745 , 0.6392157 , 0.58431375],
        [0.6901961 , 0.6431373 , 0.5882353 ],
        [0.6942239 , 0.64716506, 0.5922631 ],
        ...,
        [0.6263603 , 0.5910662 , 0.57145834],
        [0.637098  , 0.6018039 , 0.58219606],
        [0.6231179 , 0.5878238 , 0.56821597]],

       [[0.6901961 , 0.6431373 , 0.5882353 ],
        [0.69352907, 0.64647025, 0.5915683 ],
        [0.69803923, 0.6509804 , 0.59607846],
        ...,
        [0.628209  , 0.5929149 , 0.57330704],
        [0.6538319 , 0.6185378 , 0.59892994],
        [0.6451323 , 0.6098382 , 0.59023035]],

       [[0.69288343, 0.6458246 , 0.59092265],
        [0.6963971 , 0.6493383 , 0.59443635],
        [0.6982552 , 0.65119636, 0.5962944 ],
        ...,
        [0.62834966, 0.59305555, 0.5734477 ],
        [0.6256093 , 0.59031516, 0.5707073 ],
        [0.62665445, 0.59136033, 0.5717525 ]],
```

```
 ...,

 [[0.6422306 , 0.610858  , 0.65700656],
  [0.5653963 , 0.5216638 , 0.5554173 ],
  [0.609317  , 0.5498632 , 0.5697862 ],
  ...,
  [0.49010026, 0.33089146, 0.24398674],
  [0.4717082 , 0.31663126, 0.2174015 ],
  [0.45210922, 0.30701116, 0.20112881]],

 [[0.5459699 , 0.51459736, 0.56074584],
  [0.6159409 , 0.57220846, 0.605962  ],
  [0.73712665, 0.67767286, 0.69759583],
  ...,
  [0.5143739 , 0.35516512, 0.26826042],
  [0.47259298, 0.31751603, 0.21828628],
  [0.45865896, 0.3135609 , 0.20767857]],

 [[0.6225194 , 0.5911468 , 0.63729525],
  [0.638391  , 0.59465855, 0.628412  ],
  [0.6227494 , 0.5632956 , 0.5832186 ],
  ...,
  [0.49978673, 0.34057796, 0.25367323],
  [0.43982866, 0.2847517 , 0.18552195],
  [0.43587202, 0.29077396, 0.1848916 ]]],


...,


[[[0.32753947, 0.44851288, 0.11973134],
  [0.33410004, 0.48050752, 0.15104683],
  [0.3791478 , 0.5568789 , 0.2394766 ],
  ...,
  [0.40766844, 0.30570763, 0.20829864],
  [0.45871168, 0.3528293 , 0.27047637],
  [0.4270133 , 0.32113093, 0.23877798]],

 [[0.2679128 , 0.39416298, 0.08101162],
  [0.24399512, 0.39585087, 0.0862961 ],
  [0.2841259 , 0.46970016, 0.15758528],
  ...,
```

```
         [0.5176301 , 0.41566935, 0.32547325],
         [0.4865171 , 0.38063475, 0.2982818 ],
         [0.51299745, 0.40711507, 0.32476214]],

        [[0.12415747, 0.2615871 , 0.00220275],
         [0.2771018 , 0.43919915, 0.14129996],
         [0.28419244, 0.47346064, 0.18207565],
         ...,
         [0.44235033, 0.33669555, 0.25388744],
         [0.4561977 , 0.35031533, 0.2679624 ],
         [0.5066178 , 0.40073544, 0.31838247]],

        ...,

        [[0.44102108, 0.3318125 , 0.25835302],
         [0.37510908, 0.28307417, 0.2115231 ],
         [0.49756822, 0.42740095, 0.37281412],
         ...,
         [0.15410265, 0.05658789, 0.02888923],
         [0.47150543, 0.3603197 , 0.30174   ],
         [0.50879014, 0.39369893, 0.32703197]],

        [[0.3627766 , 0.24240504, 0.16030076],
         [0.60410166, 0.50382566, 0.42664382],
         [0.12501295, 0.06296554, 0.05026679],
         ...,
         [0.41484842, 0.32400462, 0.26385027],
         [0.23493192, 0.10770832, 0.01876171],
         [0.33629707, 0.19576828, 0.10374403]],

        [[0.4000932 , 0.27943492, 0.18202598],
         [0.4679393 , 0.36036953, 0.2814259 ],
         [0.41619778, 0.33086863, 0.27596667],
         ...,
         [0.33808815, 0.25181365, 0.16466382],
         [0.4662379 , 0.3383666 , 0.23973212],
         [0.4228114 , 0.2816349 , 0.1796741 ]]],


       [[[0.5339536 , 0.59277713, 0.4712085 ],
         [0.52659065, 0.5854142 , 0.46384558],
         [0.5659464 , 0.6247699 , 0.5032013 ],
```

```
         ...,
         [0.17492993, 0.2533613 , 0.24551816],
         [0.17451985, 0.25295123, 0.2451081 ],
         [0.2056018 , 0.29187632, 0.2801116 ]],

        [[0.51972795, 0.57498   , 0.44626853],
         [0.46563628, 0.5208884 , 0.39217693],
         [0.5028012 , 0.55805326, 0.4293418 ],
         ...,
         [0.15089034, 0.22575028, 0.21790715],
         [0.13931584, 0.21774723, 0.20990409],
         [0.16793676, 0.24706842, 0.23887515]],

        [[0.5022209 , 0.54772913, 0.41709688],
         [0.45174074, 0.49834937, 0.367447  ],
         [0.4719288 , 0.5196379 , 0.38846543],
         ...,
         [0.16569626, 0.24020608, 0.23236294],
         [0.1517508 , 0.22815138, 0.22030824],
         [0.17523967, 0.25367105, 0.24582791]],

        ...,

        [[0.27864137, 0.25511196, 0.12962176],
         [0.27507004, 0.23977593, 0.11820729],
         [0.2879552 , 0.23305324, 0.11932775],
         ...,
         [0.2124652 , 0.16540636, 0.11834754],
         [0.2452883 , 0.17862162, 0.13940594],
         [0.26477602, 0.19418779, 0.15497209]],

        [[0.28242254, 0.25889313, 0.13340294],
         [0.27657026, 0.24127616, 0.11970752],
         [0.28543386, 0.23053189, 0.1168064 ],
         ...,
         [0.21425499, 0.16719615, 0.12013733],
         [0.22784083, 0.16117415, 0.12195846],
         [0.2601135 , 0.18952526, 0.15030956]],

        [[0.31666628, 0.29313686, 0.16764665],
         [0.31021363, 0.2749195 , 0.15335089],
         [0.3164962 , 0.26159424, 0.14786874],
```

```
          ...,
          [0.2722983 , 0.2252395 , 0.17818066],
          [0.25509158, 0.18842492, 0.14920923],
          [0.29888868, 0.22830047, 0.18908478]]],


        [[[0.64705884, 0.627451  , 0.36862746],
          [0.60085785, 0.58125   , 0.31531864],
          [0.57953435, 0.55330884, 0.28664216],
          ...,
          [0.6229167 , 0.62806374, 0.32156864],
          [0.6192402 , 0.6317402 , 0.32083336],
          [0.6156863 , 0.63529414, 0.3137255 ]],

         [[0.6435049 , 0.6238971 , 0.36507353],
          [0.6069662 , 0.58735836, 0.32142693],
          [0.59563804, 0.5694126 , 0.29674864],
          ...,
          [0.64312965, 0.6482767 , 0.34777883],
          [0.63345593, 0.6459559 , 0.33504903],
          [0.629902  , 0.64950985, 0.32794118]],

         [[0.629902  , 0.61029416, 0.34485295],
          [0.60935587, 0.589748  , 0.32319623],
          [0.61178005, 0.58555454, 0.31227025],
          ...,
          [0.6569049 , 0.6686696 , 0.3654833 ],
          [0.6385455 , 0.65704274, 0.34375766],
          [0.63799024, 0.6575981 , 0.34264708]],

         ...,

         [[0.67242646, 0.66102946, 0.41151965],
          [0.48551244, 0.46534163, 0.23648898],
          [0.44415215, 0.4084942 , 0.18963695],
          ...,
          [0.43297338, 0.45598578, 0.20293735],
          [0.3611558 , 0.39540443, 0.13393459],
          [0.72843146, 0.7703432 , 0.5034314 ]],

         [[0.69705886, 0.7102941 , 0.44644612],
          [0.28358227, 0.2783816 , 0.08483456],
```

```
          [0.40800783, 0.35902268, 0.17595358],
          ...,
          [0.40897292, 0.41884196, 0.16441484],
          [0.3651425 , 0.3889706 , 0.13952973],
          [0.4901961 , 0.52401966, 0.27879903]],

         [[0.83921576, 0.86666673, 0.5921569 ],
          [0.7574755 , 0.76004905, 0.5139706 ],
          [0.61335784, 0.5504902 , 0.37352943],
          ...,
          [0.37487748, 0.3806373 , 0.12487746],
          [0.358701  , 0.3747549 , 0.13664216],
          [0.4901961 , 0.50980395, 0.28235295]]]], dtype=float32),
 array([[False, False, False, ..., False, False, False],
        [False, False, False, ..., False, False, False],
        [False, False, False, ..., False, False, False],
        ...,
```

```
train_images, train_labels = next(train_data.as_numpy_iterator())
show_25_images(train_images, train_labels)
```

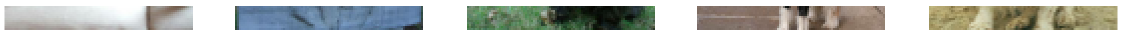

```
# Now let's visualize our validation set
val_images, val_labels = next(val_data.as_numpy_iterator())
show_25_images(val_images, val_labels)
```

Building a model

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

- The input shape (our images shape, 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 from TensorFlow Hub - https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4

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

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

# Setup 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 a Keras deep learning model!

Knowing this, let's create a function which:

- 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 these steps can be found here: https://www.tensorflow.org/guide/keras/overview

```
# 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(),
      optimizer=tf.keras.optimizers.Adam(),
      metrics=["accuracy"]
  )

  # Build the model
  model.build(INPUT_SHAPE)

  return model
```

- Read about TensorFLowHub.
- Pytorch Hub
- Modelzoo
- Paperswithcode

```
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)     (None, 1001)              5432713
_________________________________________________________________
dense (Dense)                (None, 120)               120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

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 early training 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 (before overfitting).

TensorBoard Callback

To setup a Tensorboard callback, we need to do 3 things:

1. Load the TensorBoard Notebook extention.
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 our models training logs with the `%tensorboard` magic function (we'll do this after model training).

```
# Load TensorBoard Notebook extention
%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/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)
```

Early Stopping Callback

Early Stopping helps stop our model from overfitting by stopping training if a certain evalaution metric stops improving.

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/EarlyStopping

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

Training a model (on subset of data)

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

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

NUM_EPOCHS: 100

```
# Check to make sure we're still running on a GPU
print("GPU", "Available (Yess!!!!!)" if tf.config.list_physical_devices("GPU") else "Not Available :(")
```

```
GPU Available (Yess!!!!!)
```

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

```
# 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,
            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
 1/25 [>.............................] - ETA: 0s - loss: 5.8843 - accuracy: 0.0000e+00WARNING:tensorflow:From /usr/local/lib/py
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/summary_ops_v2.py:1277: stop (from tensorf
Instructions for updating:
use `tf.profiler.experimental.stop` instead.
25/25 [==============================] - 484s 19s/step - loss: 4.5522 - accuracy: 0.0925 - val_loss: 3.3863 - val_accuracy: 0.2
Epoch 2/100
25/25 [==============================] - 4s 144ms/step - loss: 1.6168 - accuracy: 0.6975 - val_loss: 2.1140 - val_accuracy: 0.5
Epoch 3/100
25/25 [==============================] - 4s 146ms/step - loss: 0.5745 - accuracy: 0.9312 - val_loss: 1.6566 - val_accuracy: 0.6
Epoch 4/100
25/25 [==============================] - 4s 147ms/step - loss: 0.2626 - accuracy: 0.9800 - val_loss: 1.4671 - val_accuracy: 0.6
Epoch 5/100
25/25 [==============================] - 4s 145ms/step - loss: 0.1505 - accuracy: 0.9950 - val_loss: 1.3825 - val_accuracy: 0.6
Epoch 6/100
25/25 [==============================] - 4s 145ms/step - loss: 0.1014 - accuracy: 1.0000 - val_loss: 1.3429 - val_accuracy: 0.6
Epoch 7/100
25/25 [==============================] - 4s 146ms/step - loss: 0.0766 - accuracy: 1.0000 - val_loss: 1.3155 - val_accuracy: 0.6
Epoch 8/100
25/25 [==============================] - 4s 145ms/step - loss: 0.0603 - accuracy: 1.0000 - val_loss: 1.3009 - val_accuracy: 0.6
Epoch 9/100
25/25 [==============================] - 4s 145ms/step - loss: 0.0496 - accuracy: 1.0000 - val_loss: 1.2743 - val_accuracy: 0.6
Epoch 10/100
25/25 [==============================] - 4s 147ms/step - loss: 0.0419 - accuracy: 1.0000 - val_loss: 1.2654 - val_accuracy: 0.6
```

Question: It looks like our model is overfitting because it's performing far better on the training dataset than the validation 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
```

TensorBoard SCALARS GRAPHS INACTIVE

Filter tags (regular expressions supported)

- [ ] Show data download links
- [ ] Ignore outliers in chart scalin

Tooltip sorting method: default

Smoothing

0.6

Horizontal Axis

STEP RELATIVE

WALL

Runs

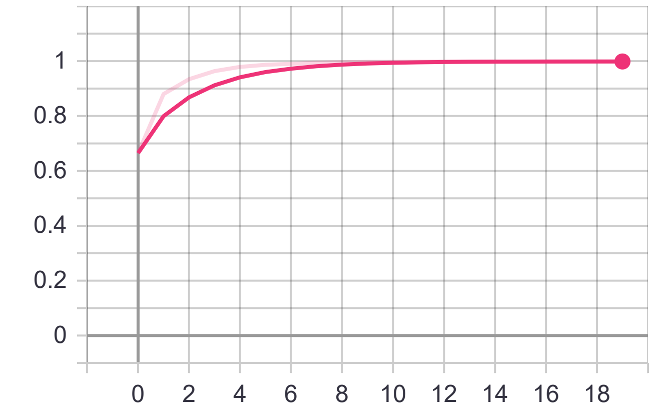
Write a regex to filter runs

- 20200904-191444/train
- 20200904-191444/validation
- 20200907-202223/train
- 20200907-202223/validation
- 20200907-212716/train

TOGGLE ALL RUNS

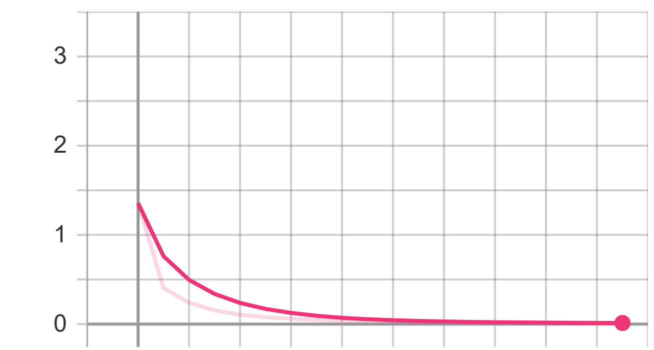
epoch_accuracy

epoch_accuracy

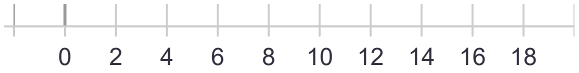


epoch_loss

epoch_loss



drive/My Drive/Dog Vision/logs



Making & evaluating predictions using a trained model

```
val_data
```

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

```
#loaded_1000_image_model = load_model("drive/My Drive/Dog Vision/models/20200907-21061599512817-1000-images-mobilenetv2-Adam.h5")
```

```
Loading saved model from: drive/My Drive/Dog Vision/models/20200907-21061599512817-1000-images-mobilenetv2-Adam.h5
```

```
# Make prediction on the validation data (not used to train on)
predictions = loaded_1000_image_model.predict(val_data, verbose=1)
predictions
```

```
1/7 [===>..........................] - ETA: 0sWARNING:tensorflow:Callbacks method `on_predict_batch_end` is slow compared to th
WARNING:tensorflow:Callbacks method `on_predict_batch_end` is slow compared to the batch time (batch time: 0.0154s vs `on_predi
```

- X_val: images to be predicted upon
- y_val: truth labels of breeds to compare the predictions

```
        1.9291471e-03, 7.1360439e-05],
```

```
predictions.shape
```

```
(200, 120)
```

```
        5.3903434e-05, 5.6749443e-05],
```

```
predictions[0] # predicted breed_labels of X_val(images)
```

```
array([1.2692247e-03, 1.5655077e-04, 1.4224066e-03, 6.1726030e-05,
       1.2277943e-03, 1.4124677e-04, 4.3431573e-02, 7.6954329e-04,
       6.4321357e-04, 6.4214313e-04, 1.3704959e-04, 1.6139327e-05,
       1.9150991e-03, 2.6021918e-04, 4.1013682e-04, 9.6193183e-04,
       8.5792890e-05, 8.8516772e-02, 6.0315822e-05, 9.5297575e-05,
```

```
# First prediction
index=0
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[np.argmax(predictions[index])]}")
```

```
[1.2692247e-03 1.5655077e-04 1.4224066e-03 6.1726030e-05 1.2277943e-03
 1.4124677e-04 4.3431573e-02 7.6954329e-04 6.4321357e-04 6.4214313e-04
```

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 being made on!

Note: Prediction probabilities are also known as confidence levels.

```
 2.8670216e-03 4.2297277e-05 9.5375080e-04 7.9822046e-04 1.8682824e-04
```

```
# 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'
Sum: 1.0
```

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

```
val_data
```

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

Unbatchify: It is so because to obtain validation images and the truth labels corresponding to it.

- Val_images = images for predicting
- Val_labels = truth labels

```
# Create a function to unbatch a batch dataset
def unbatchify(data):
  """
  Takes a batched dataset of (image, label) Tensors and returns separate arrays
  """
  images = []
  labels = []
  # Loop through unbatched data
  for image, label in data.unbatch().as_numpy_iterator():
    images.append(image)
    labels.append(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 ]],
```

```
get_pred_label(val_labels[0]) # truth labels (breed of validation dataset)
```

```
'cairn'
        [0.08394483, 0.08394483, 0.08394483],
```

```
get_pred_label(predictions[0]) # predicted breed after training
```

```
'cairn'
        False, False, False, False, False, False, False, False,  True,
```

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, it's predicted probability, the truth label and the target image on a single plot.

```
def plot_pred(prediction_probabilities, labels, images, n=1):
  """
  View the prediction label, 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)
  true_label = get_pred_label(true_label)

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

```
  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 = "blue"
  else:
    color = "red"

  # Change the plot title to be predicted label, 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)
```

cairn 29% cairn

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:

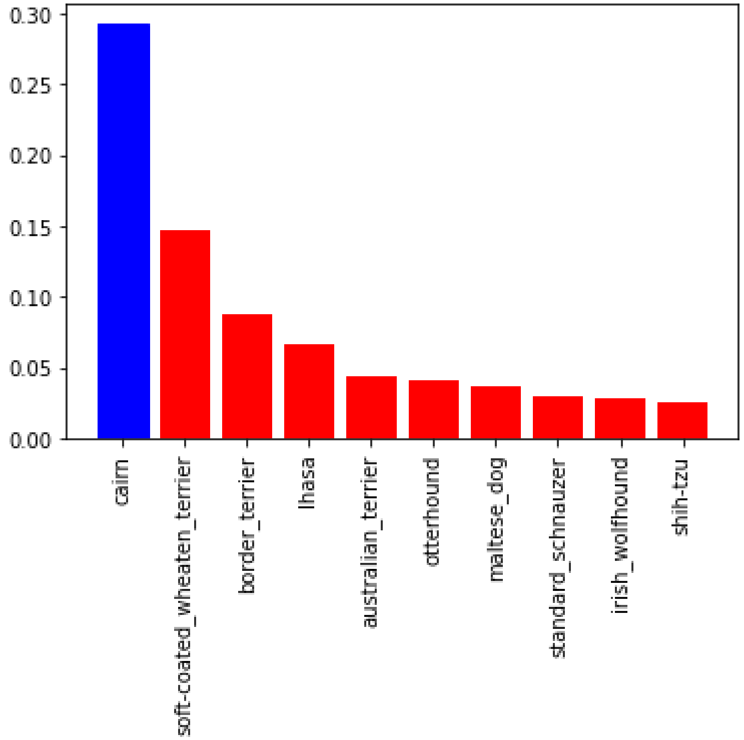
- Take an input of prediction probabilities array and a ground truth array and an integer.

- Find the predicted label using `get_pred_label()`.
- Find the top 10:
    - Prediction probabilities index
    - Prediction probabilities values
    - Prediction labels
- Plot the top 10 prediction probability values and labels, colouring the true label `blue`.

```
def plot_pred_conf(prediction_probabilities, labels, n=1):
  """
  Plot 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 & true label
  pred_label = get_pred_label(pred_prob)
  true_label = get_pred_label(true_label)

  # 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(x=np.arange(len(top_10_pred_labels)),
                     height=top_10_pred_values,
                     color="red")
  plt.xticks(np.arange(len(top_10_pred_labels)),
             labels=top_10_pred_labels,
             rotation="vertical")

  # Change colour of true label
```

```
  if np.isin(true_label, top_10_pred_labels):
    top_plot[np.argmax(true_label == top_10_pred_labels)].set_color("blue")
  else:
    pass
```

```
plot_pred_conf(predictions, val_labels, n=0)
```




```
# predictions[0]
```

```
# predictions[0].argsort() # sorting in ascending order
```

```
# predictions[0].argsort()[-10:] # top 10 values in ascending order
```

```
# predictions[0].argsort()[-10:][::-1] # top 10 values in descending order
```

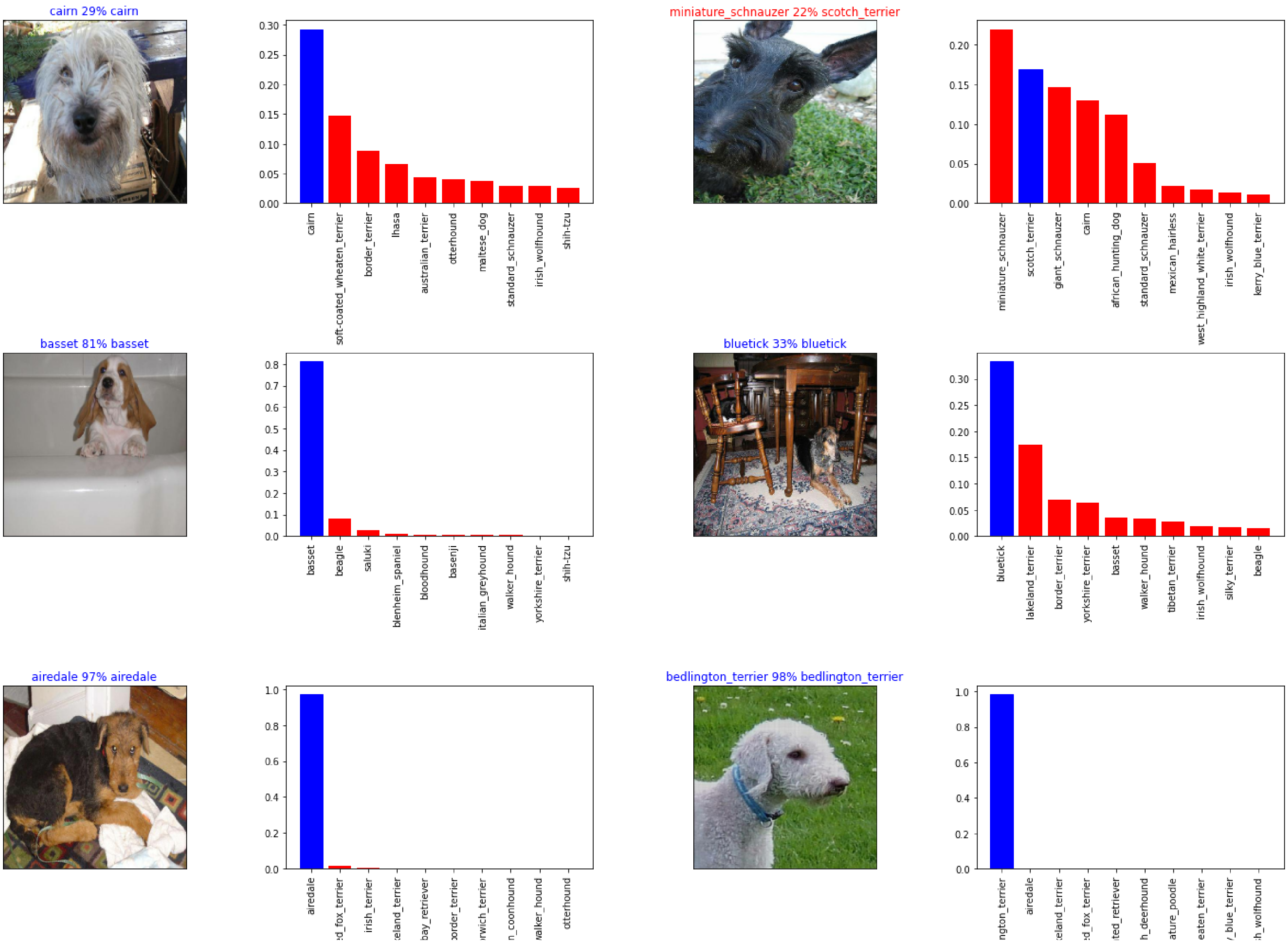
```
# unique_breeds[predictions[0].argsort()[-10:][::-1]]
```

```
# predictions[0][predictions[0].argsort()[-10:][::-1]] # top 10 in descending order with prediction values
```

```
# predictions[0].max() # maximum prediction
```

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

```
# Let's checkout 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=(10*num_cols, 5*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_pred(predictions,
            val_labels,
            val_images,
            n=i+i_multiplier)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_pred_conf(predictions,
                 val_labels,
                 n=i+i_multiplier)
plt.tight_layout(h_pad=1.0)
plt.show()
```


cairn 29% cairn
cairn
soft-coated_wheaten_terrier
border_terrier
lhasa
australian_terrier
otterhound
maltese_dog
standard_schnauzer
irish_wolfhound
shih-tzu
miniature_schnauzer 22% scotch_terrier
miniature_schnauzer
scotch_terrier
giant_schnauzer
cairn
african_hunting_dog
standard_schnauzer
mexican_hairless
west_highland_white_terrier
irish_wolfhound
kerry_blue_terrier
basset 81% basset
basset
beagle
saluki
blenheim_spaniel
bloodhound
basenji
italian_greyhound
walker_hound
yorkshire_terrier
shih-tzu
bluetick 33% bluetick
bluetick
lakeland_terrier
border_terrier
yorkshire_terrier
basset
walker_hound
tibetan_terrier
irish_wolfhound
silky_terrier
beagle
airedale 97% airedale
airedale
bedlington_terrier 98% bedlington_terrier
airedale

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

Saving and reloading a trained model

```
# Create a function to save a model
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
```

```
# Create a function to load a trained model
def load_model(model_path):
  """
  Loads a saved model from 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 function to load and trained model, let's make sure they work!

```
# Save our model trained on 1000 images
save_model(model, suffix="1000-images-mobilenetv2-Adam")
```

```
Saving model to: drive/My Drive/Dog Vision/models/20200908-10251599560740-1000-images-mobilenetv2-Adam.h5...
'drive/My Drive/Dog Vision/models/20200908-10251599560740-1000-images-mobilenetv2-Adam.h5'
```

```
# Load our saved model of 1000 images
loaded_1000_image_model = load_model("drive/My Drive/Dog Vision/models/20200904-20161599250582-1000-images-mobilenetv2-Adam.h5")
```

```
Loading saved model from: drive/My Drive/Dog Vision/models/20200904-20161599250582-1000-images-mobilenetv2-Adam.h5
```

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

```
1 # Load our saved model of 1000 images
```

```
loaded_1000_image_model.evaluate(val_data)
```

```
7/7 [==============================] - 1s 114ms/step - loss: 1.2654 - accuracy: 0.6750
[1.2654260396957397, 0.675000011920929]
```

```
111                    (export_dir,
```

Training a big dog model (on the full data)

```
113
```

```
len(X), len(y)
```

```
(10222, 10222)
```

SEARCH STACK OVERFLOW

```
# Create a data batch with the full dataset
full_data = create_data_batches(X,y)
```

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

```
full_data
```

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

```
# Create a model for full model
full_model = create_model()
```

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

```
full_model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer_1 (KerasLayer)   (None, 1001)              5432713
_________________________________________________________________
dense_1 (Dense)              (None, 120)               120240
=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

```
# 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 minutes 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 [==============================] - 5138s 16s/step - loss: 1.3559 - accuracy: 0.6647
Epoch 2/100
320/320 [==============================] - 37s 115ms/step - loss: 0.4042 - accuracy: 0.8801
Epoch 3/100
320/320 [==============================] - 36s 113ms/step - loss: 0.2406 - accuracy: 0.9342
Epoch 4/100
320/320 [==============================] - 36s 113ms/step - loss: 0.1539 - accuracy: 0.9636
Epoch 5/100
320/320 [==============================] - 37s 116ms/step - loss: 0.1057 - accuracy: 0.9790
Epoch 6/100
320/320 [==============================] - 36s 114ms/step - loss: 0.0781 - accuracy: 0.9866
Epoch 7/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0610 - accuracy: 0.9900
Epoch 8/100
320/320 [==============================] - 36s 114ms/step - loss: 0.0464 - accuracy: 0.9943
Epoch 9/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0367 - accuracy: 0.9960
Epoch 10/100
320/320 [==============================] - 36s 114ms/step - loss: 0.0314 - accuracy: 0.9975
Epoch 11/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0267 - accuracy: 0.9976
Epoch 12/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0238 - accuracy: 0.9982
Epoch 13/100
320/320 [==============================] - 36s 114ms/step - loss: 0.0197 - accuracy: 0.9985
Epoch 14/100
320/320 [==============================] - 37s 115ms/step - loss: 0.0170 - accuracy: 0.9986
Epoch 15/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0159 - accuracy: 0.9985
Epoch 16/100
320/320 [==============================] - 36s 113ms/step - loss: 0.0155 - accuracy: 0.9984
Epoch 17/100
```

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

```
Saving model to: drive/My Drive/Dog Vision/models/20200907-23091599520165-full-image-set-mobilenetv2-Adam.h5...
'drive/My Drive/Dog Vision/models/20200907-23091599520165-full-image-set-mobilenetv2-Adam.h5'
```

```
Epoch 20/100
```

```
# Load the full model
loaded_full_model = load_model("drive/My Drive/Dog Vision/models/20200907-23091599520165-full-image-set-mobilenetv2-Adam.h5")
```

```
Loading saved model from: drive/My Drive/Dog Vision/models/20200907-23091599520165-full-image-set-mobilenetv2-Adam.h5
```

Making predictions on the test dataset

Since our model has been trained on images in the form of Tensor bathes, 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 predictions array by passing the test batches to the `predict()` method called on our model.

```
# Load the test image filenames
test_path = "drive/My Drive/Dog Vision/test/"
test_filenames = [test_path + fname for fname in os.listdir(test_path)]
test_filenames[:10]
```

```
['drive/My Drive/Dog Vision/test/9c367f64333c1f6a15af6374a98ea194.jpg',
 'drive/My Drive/Dog Vision/test/9de355552bcfb01109a8c0f653ab19a0.jpg',
 'drive/My Drive/Dog Vision/test/9caa3789cfa67aa2a494e43b4af09e8b.jpg',
 'drive/My Drive/Dog Vision/test/9b53256312d57f6299a0cac786dfec11.jpg',
 'drive/My Drive/Dog Vision/test/9d7ad4ab8dc4663ec7aeaf224f44610a.jpg',
 'drive/My Drive/Dog Vision/test/9cde54bb3e7768a02666aab0972df287.jpg',
 'drive/My Drive/Dog Vision/test/9c3c4d2f72310435a7107303e50e5ec5.jpg',
 'drive/My Drive/Dog Vision/test/9c8d7eeba2bcfed3bd1fba0a7de3ed21.jpg',
 'drive/My Drive/Dog Vision/test/9de87aba130f43ecffe870790873e219.jpg',
 'drive/My Drive/Dog Vision/test/9b55ca08b94e9210530ff6da1f606ebe.jpg']
```

```
len(test_filenames)
```

```
10357
```

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

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

```
test_data.element_spec
```

```
TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None)
```

```
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 ~1 hour). This is because we have to process ~10,000+ images and get our model to find patterns in those images and generate predictions based on what its learned in the training dataset.

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

```
324/324 [==============================] - 7352s 23s/step
```

```
# Save predictions [Numpy Array] to csv file (for access later)
np.savetxt("drive/My Drive/Dog Vision/preds_array.csv", test_predictions, delimiter=",")
```

```
# Load predictions [Numpy Array] from csv file
test_predictions = np.loadtxt("drive/My Drive/Dog Vision/preds_array.csv", delimiter=",")
```

```
test_predictions[0]
```

```
array([1.33263609e-06, 5.20833765e-10, 4.31628934e-11, 1.41465756e-10,
       2.42075004e-09, 3.32206374e-10, 1.89002629e-14, 1.70123098e-08,
       5.07003506e-07, 4.09411893e-10, 4.63370592e-11, 3.09195364e-10,
       5.31677979e-10, 8.96493768e-10, 2.96163871e-10, 2.40141101e-10,
       8.95338581e-10, 2.36255904e-09, 3.05699399e-09, 1.79487543e-07,
       9.37010136e-10, 1.72853890e-06, 5.25485575e-07, 2.66373608e-08,
       9.29126553e-09, 1.85870547e-07, 1.31967837e-09, 8.92246543e-09,
       7.61499575e-09, 7.51978813e-10, 1.00204825e-05, 1.04165467e-08,
       4.81516658e-08, 2.57514223e-08, 1.69352206e-12, 1.42466905e-09,
       1.55777824e-09, 2.66373945e-09, 1.76658843e-09, 6.54096111e-10,
       4.41944745e-11, 1.79611337e-09, 1.62790940e-07, 2.54963245e-10,
       1.50382062e-09, 3.84166654e-08, 1.21109096e-08, 5.58862912e-09,
       1.30888911e-09, 4.17486601e-11, 2.09647566e-09, 2.93293445e-11,
       1.46691637e-08, 4.75905670e-09, 1.14128778e-08, 6.52321600e-11,
       3.37374328e-09, 6.42198447e-11, 3.48390605e-09, 3.89563981e-09,
       3.92428978e-10, 3.33014172e-09, 6.33414231e-07, 2.42297671e-09,
       4.73309864e-11, 3.36258937e-10, 4.10872204e-11, 1.59895250e-10,
       5.54981998e-11, 1.87456567e-06, 2.33070409e-06, 4.31106040e-09,
       7.12281178e-10, 9.14731416e-11, 2.98671678e-11, 3.76848108e-09,
       1.85638432e-11, 2.56436001e-11, 5.93290617e-09, 1.16567603e-10,
       1.31900455e-08, 8.06485850e-11, 3.46276452e-10, 4.95420302e-11,
       2.13294452e-10, 9.99966860e-01, 8.03652769e-11, 2.89565243e-08,
       3.14420549e-06, 1.15016663e-08, 7.68512337e-12, 1.25241515e-08,
       5.56088617e-06, 2.26655512e-08, 2.64910205e-09, 1.63801221e-08,
       6.31718636e-12, 3.46131781e-14, 5.68719828e-11, 2.29934294e-08,
       2.56122098e-06, 7.83578447e-09, 1.40653373e-08, 8.67696315e-09,
       1.21080812e-10, 1.60438818e-09, 7.70667974e-10, 6.28162269e-08,
       1.70444957e-06, 5.27338742e-08, 8.43633341e-09, 2.47829118e-11,
       6.28128626e-12, 9.38007519e-11, 1.98154219e-11, 1.05180682e-08,
       4.26490088e-09, 1.00500868e-12, 9.49400292e-09, 8.04011968e-10])
```

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.

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

To get the data in this format:

- Create a pandas DataFrame with an ID column as well as a column for each dog breed
- Add data to 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 to Kaggle.

```
["id"] + list(unique_breeds)
```

```
['id',
 'affenpinscher',
 'afghan_hound',
 'african_hunting_dog',
 'airedale',
 'american_staffordshire_terrier',
 'appenzeller',
 'australian_terrier',
 'basenji',
 'basset',
 'beagle',
 'bedlington_terrier',
 'bernese_mountain_dog',
 'black-and-tan_coonhound',
 'blenheim_spaniel',
 'bloodhound',
 'bluetick',
 'border_collie',
 'border_terrier',
 'borzoi',
 'boston_bull',
 'bouvier_des_flandres',
 'boxer',
 'brabancon_griffon',
 'briard',
 'brittany_spaniel',
 'bull_mastiff',
 'cairn',
 'cardigan',
 'chesapeake_bay_retriever',
 'chihuahua',
 'chow',
 'clumber',
 'cocker_spaniel',
 'collie',
 'curly-coated_retriever',
 'dandie_dinmont',
 'dhole',
 'dingo',
 'doberman',
 'english_foxhound',
 'english_setter',
```

```
 'english_setter',
 'english_springer',
 'entlebucher',
 'eskimo_dog',
 'flat-coated_retriever',
 'french_bulldog',
 'german_shepherd',
 'german_short-haired_pointer',
 'giant_schnauzer',
 'golden_retriever',
 'gordon_setter',
 'great_dane',
 'great_pyrenees',
 'greater_swiss_mountain_dog',
 'groenendael',
 'ibizan_hound',
 'irish_setter',
 'irish_terrier',
 'irish_water_spaniel',
 'irish_wolfhound',
 'italian_greyhound',
 'japanese_spaniel',
 'keeshond',
 'kelpie',
 'kerry_blue_terrier',
 'komondor',
 'kuvasz',
 'labrador_retriever',
 'lakeland_terrier',
 'leonberg',
 'lhasa',
 'malamute',
 'malinois',
 'maltese_dog',
 'mexican_hairless',
 'miniature_pinscher',
 'miniature_poodle',
 'miniature_schnauzer',
 'newfoundland',
 'norfolk_terrier',
 'norwegian_elkhound',
 'norwich_terrier',
 'old_english_sheepdog',
```

```
       'otterhound',
       'papillon',
       'pekinese',
       'pembroke',
       'pomeranian',
       'pug',
       'redbone',
```

```
# Create a 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	appenzeller	australian_terri

0 rows × 121 columns

```
       'siberian husky',
```

```
# Append test image ID's to predictions DataFrame
test_ids = [os.path.splitext(path)[0] for path in os.listdir(test_path)]
test_ids[:10]
preds_df["id"] = test_ids
```

```
       'sussex_spaniel',
```

```
preds_df.head()
```

id affenpinscher afghan hound african hunting dog airedale american staffordshire terrier

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

	id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffordshire_terrier
0	9c367f64333c1f6a15af6374a98ea194	1.33264e-06	5.20834e-10	4.31629e-11	1.41466e-10	2.42075e-09
1	9de355552bcfb01109a8c0f653ab19a0	1.11417e-10	1.95093e-11	6.74175e-12	1.36548e-08	9.57503e-07
2	9caa3789cfa67aa2a494e43b4af09e8b	9.8866e-10	1.71997e-10	2.69859e-09	7.7467e-09	0.703544
3	9b53256312d57f6299a0cac786dfec11	3.49597e-12	7.06352e-11	8.36399e-13	4.2818e-07	0.000402455
4	9d7ad4ab8dc4663ec7aeaf224f44610a	9.64526e-11	7.63873e-09	1.52164e-09	5.52058e-08	8.48313e-10

5 rows × 121 columns

```
# Save our predictions dataframe to CSV for submission to Kaggle
preds_df.to_csv("drive/My Drive/Dog Vision/full_model_predictions_submission_1_mobilenetV2.csv", index=False)
```

Making predictions on custom images

To make predictions on custom images, we'll:

- Get the filepaths of our own images.

- Turn the filepaths into data vatches 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 robabilities to prediction labels.
- Compare the predicted labels to the custom images

```
# Get custom image filepaths
custom_path = "drive/My Drive/Dog Vision/my-dog-photos/"
custom_image_paths = [custom_path + fname for fname in os.listdir(custom_path)]
```

```
os.listdir(custom_path)
```

```
['puppy-flickr.jpg',
 'basenji-GettyImages-90633583-77acb9bcfcd64af698b0d3330da48dd2.jpg',
 'beagle2-e1523513313893.jpg']
```

```
custom_image_paths
```

```
['drive/My Drive/Dog Vision/my-dog-photos/puppy-flickr.jpg',
 'drive/My Drive/Dog Vision/my-dog-photos/basenji-GettyImages-90633583-77acb9bcfcd64af698b0d3330da48dd2.jpg',
 'drive/My Drive/Dog Vision/my-dog-photos/beagle2-e1523513313893.jpg']
```

```
# Turn custim images into batch dataset
custom_data = create_data_batches(custom_image_paths, test_data=True)
custom_data
```

```
Creating test data batches...
<BatchDataset shapes: (None, 224, 224, 3), types: tf.float32>
```

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

```
custom_preds
```

```
array([[3.55337648e-10, 2.55457482e-12, 9.31433153e-09, 8.82358293e-08,
        1.80829113e-04, 5.65436602e-01, 6.54313135e-06, 7.00423436e-04,
        1.16806705e-12, 2.92201641e-08, 2.36597566e-06, 2.49438667e-14,
        1.64780978e-09, 4.52057192e-09, 2.55163872e-08, 4.36312948e-05,
        1.74138972e-06, 2.66407520e-01, 1.30802098e-08, 1.08700135e-07,
        7.28469854e-11, 3.24789848e-11, 6.72387250e-06, 1.48348234e-09,
        1.43364517e-07, 1.78764037e-09, 1.59417729e-10, 8.68016912e-08,
        3.62784704e-07, 1.91555284e-02, 8.85692519e-10, 9.92321711e-11,
        2.48525384e-11, 6.56389375e-06, 3.18588955e-09, 1.73362477e-10,
        7.36660922e-06, 1.22184438e-05, 5.87940474e-10, 1.04185131e-06,
        2.97969621e-10, 4.99260828e-08, 4.98717213e-07, 8.30129409e-07,
        2.73540390e-09, 2.35263198e-11, 7.73085207e-10, 4.28202682e-08,
        9.79635817e-10, 2.43922187e-07, 1.35540814e-08, 6.93688662e-10,
        1.94238439e-08, 9.70366964e-05, 7.09499612e-11, 7.31829005e-07,
        1.10796499e-07, 2.14245487e-02, 7.93987543e-13, 2.08732729e-08,
        1.85433280e-04, 1.07823062e-09, 1.28925454e-10, 2.77027744e-03,
        5.31415800e-08, 2.73684062e-08, 3.78450721e-07, 3.77994111e-05,
        8.45056056e-05, 2.89960189e-09, 8.36865924e-11, 7.44114926e-12,
        4.20793441e-07, 4.88195161e-12, 3.98387647e-06, 3.97971869e-02,
        2.18964706e-05, 2.34048714e-09, 2.64567390e-08, 3.16826845e-05,
        1.47832100e-06, 3.81027746e-07, 3.99453581e-10, 1.01026965e-10,
        2.08613938e-06, 1.89756189e-09, 1.64421678e-07, 1.75192603e-11,
        1.88961621e-07, 6.35893550e-04, 1.44173509e-06, 1.61613087e-07,
        4.81133244e-10, 1.73249273e-08, 6.74615572e-11, 1.66480980e-08,
        3.53175433e-10, 3.21306434e-05, 5.26958033e-10, 8.78497448e-12,
        6.73739331e-09, 4.28946345e-09, 1.29363380e-05, 2.37523814e-08,
        5.40815992e-04, 2.69393968e-06, 6.14173645e-09, 7.25013161e-10,
        1.96152539e-11, 5.59400824e-11, 1.87849949e-07, 7.56951421e-02,
        7.30598204e-09, 2.44765488e-07, 2.31541470e-08, 1.31437389e-07,
        8.67786731e-10, 5.22668147e-03, 1.41891022e-03, 3.63531740e-07],
       [3.24483843e-11, 1.57657276e-09, 4.43132421e-06, 1.04271103e-09,
        5.52010275e-02, 8.63192763e-05, 1.23766402e-03, 8.07992160e-01,
        4.47600925e-08, 1.27599003e-06, 8.40397121e-08, 8.93756694e-13,
        8.71311627e-08, 1.65455578e-08, 2.09334274e-08, 5.60730795e-10,
        9.64531750e-07, 1.86702009e-09, 5.31820988e-05, 1.58025759e-09,
        9.36604394e-09, 4.42236692e-09, 1.01559924e-11, 5.42656764e-09,
        1.15837562e-08, 3.46357283e-11, 6.82407686e-10, 5.75691869e-04,
        3.23006955e-10, 3.19072774e-06, 3.24499627e-09, 3.60671422e-08,
        7.80275933e-09, 1.91546604e-03, 7.89455896e-08, 8.01224065e-09,
        3.76387078e-07, 6.44524189e-05, 2.79377076e-07, 4.15491149e-06,
        6.41629777e-06, 1.38082585e-08, 3.67979180e-09, 1.41506774e-07,
        6.55248527e-07, 2.93918512e-09, 1.95194417e-07, 3.26625980e-08,
```

```
        2.78957080e-10, 1.47176597e-05, 1.09227733e-06, 7.11086886e-06,
        7.30665750e-10, 3.77471743e-08, 1.65608362e-05, 1.23666070e-01,
        3.35618779e-07, 3.03348287e-07, 5.65374469e-10, 2.01958315e-07,
        4.75437701e-05, 2.31355868e-09, 1.31164163e-10, 1.32996838e-05,
        1.15156140e-09, 8.95748897e-08, 1.41749211e-07, 7.90346348e-08,
        4.93142124e-06, 4.69610351e-10, 8.87863798e-14, 1.00440838e-08,
        4.09914691e-07, 3.99471167e-09, 1.38066534e-04, 1.33360700e-09,
        5.47245982e-08, 5.77075263e-11, 7.63557395e-08, 4.41619278e-07,
        4.78266715e-09, 8.68657424e-10, 2.45385667e-09, 1.22566997e-08,
        5.94416747e-07, 2.14022862e-11, 7.85306841e-03, 8.94685073e-08,
        2.38478154e-10, 2.52433492e-05, 8.60049295e-06, 1.50226151e-11,
        1.81278725e-09, 6.88727741e-05, 1.16164256e-06, 6.49601528e-09,
        8.97829182e-07, 1.41226666e-08, 4.27421182e-10, 3.95046955e-05,
        2.85836251e-07, 4.86864883e-06, 5.90186779e-08, 1.06554643e-09,
        4.71239092e-09, 9.06013824e-08, 3.58029695e-07, 4.81501417e-09,
        2.20874378e-11, 9.60464042e-10, 2.06924544e-09, 2.43761406e-05,
        3.08520089e-06, 1.39257332e-04, 2.74457765e-04, 4.13241447e-04,
        3.74428367e-07, 8.14215891e-05, 5.50057599e-09, 7.27062410e-10],
       [1.35091671e-09, 1.09258969e-09, 5.26486144e-09, 2.41524333e-07,
        8.16568502e-07, 8.23920345e-06, 5.81696850e-06, 1.66961436e-05,
        1.65748327e-06, 9.86593366e-01, 2.43416287e-09, 7.92084398e-10,
        6.49176343e-07, 1.66220366e-06, 6.98927494e-08, 3.13357214e-06,
        1.31581567e-06, 9.75845978e-07, 1.14263954e-09, 7.21556415e-09,
        1.97478633e-09, 3.66514179e-08, 4.38231629e-09, 1.09412618e-10,
        1.68852624e-04, 1.89413058e-07, 4.77853646e-10, 3.43777992e-06,
        5.22267776e-08, 2.08062856e-09, 5.73349475e-08, 2.23755237e-08,
        1.52870583e-09, 1.84687252e-08, 9.44708534e-10, 4.20265309e-07,
        6.88247439e-07, 1.98729051e-08, 3.11811976e-09, 1.24603668e-02,
        9.97859644e-08, 2.35450770e-08, 1.09996749e-04, 2.72818539e-07,
        2.56727674e-07, 9.50962331e-09, 1.26369204e-09, 1.43396383e-05,
        2.02013410e-08, 1.00996447e-08, 4.64152308e-08, 2.02449169e-09,
        6.01681620e-08, 1.65409779e-06, 5.52030643e-09, 1.80284463e-08,
        2.71968936e-08, 2.78629120e-10, 2.34196884e-10, 1.11742504e-09,
        5.44614709e-09, 1.25451527e-06, 7.58654639e-10, 2.37375630e-09,
        1.78252400e-08, 8.37567793e-09, 8.76488326e-10, 1.25504664e-06,
        2.38797124e-08, 2.21475416e-09, 1.54148927e-09, 1.18736432e-09,
        3.99473841e-08, 1.23653365e-07, 5.65909053e-09, 5.75603627e-11,
        5.35213873e-09, 1.48040913e-10, 2.85393912e-06, 9.32787589e-06,
        8.40189145e-08, 1.36767472e-10, 9.59670743e-08, 3.73633441e-07,
        1.96584873e-10, 1.45227589e-11, 4.50902519e-07, 1.06478394e-11,
        1.13831522e-09, 1.91010640e-05, 2.02844301e-08, 3.50032253e-10,
        9.71892788e-08, 3.33635882e-07, 7.76719689e-10, 6.01084871e-09,
```

```
        4.39788657e-11, 5.63510394e-09, 5.13410914e-09, 1.45413389e-06,
        7.39543182e-09, 2.84549895e-09, 1.46153369e-07, 1.37607552e-08,
        3.78801490e-10, 1.58428838e-06, 3.24734458e-07, 4.82806861e-08,
        1.65826186e-09, 1.16473231e-08, 1.75233239e-09, 2.30170230e-07,
        3.86505235e-06, 5.09755162e-04, 1.42376484e-08, 5.10797217e-05,
        1.36565594e-08, 3.14217523e-08, 2.79045963e-07, 3.28038841e-09]],
      dtype=float32)
```

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

```
['appenzeller', 'basenji', 'beagle']
```

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

```
# Check custom images 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)
```


appenzeller
basenji
beagle