# dog-vision

## 0.1 End-to-end Multi-class Dog Breed Classification

This notebook builds an end-to-end multi-class image classifier using TensorFlow 2.x 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 proba for each dog breed of each test image.

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

**4. Features**

Some information about the data:

We're dealing with images (unstructured data) so it's probably best we use deep learning/transfer learning. There are 120 breeds of dogs (this means there are 120 different classes). There are around 10,000+ images in the training set (these images have labels). There are around 10,000+ images in the test set (these images have no labels, because we'll want to predict them).

[1]:
```
# Unzip the uploaded data into Google Drive
#!unzip "/content/drive/MyDrive/Dog Vision/dog-breed-identification.zip" -d "/
↪content/drive/MyDrive/Dog Vision/"
```

**Get our workspace ready** Import TensorFlow 2.x Import TensorFlow Hub Make sure we're using a GPU

[2]:
```
# Import TensorFlow into Colab
import tensorflow as tf
print("TF version:", tf.__version__)
```

```
TF version: 2.15.0
```

[3]:
```python
# Import necessary tools
import tensorflow_hub as hub
print("TF Hub version:", hub.__version__)

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

```
TF Hub version: 0.16.1
GPU available (YESSSS!!!!!)
```

## 0.2 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.

[4]:
```python
import pandas as pd
labels_csv = pd.read_csv("/content/drive/MyDrive/Dog Vision/labels.csv")
labels_csv.describe()
```

[4]:

| | id | breed |
|---|---|---|
| count | 10222 | 10222 |
| unique | 10222 | 120 |
| top | 000bec180eb18c7604dcecc8fe0dba07 | scottish_deerhound |
| freq | 1 | 126 |

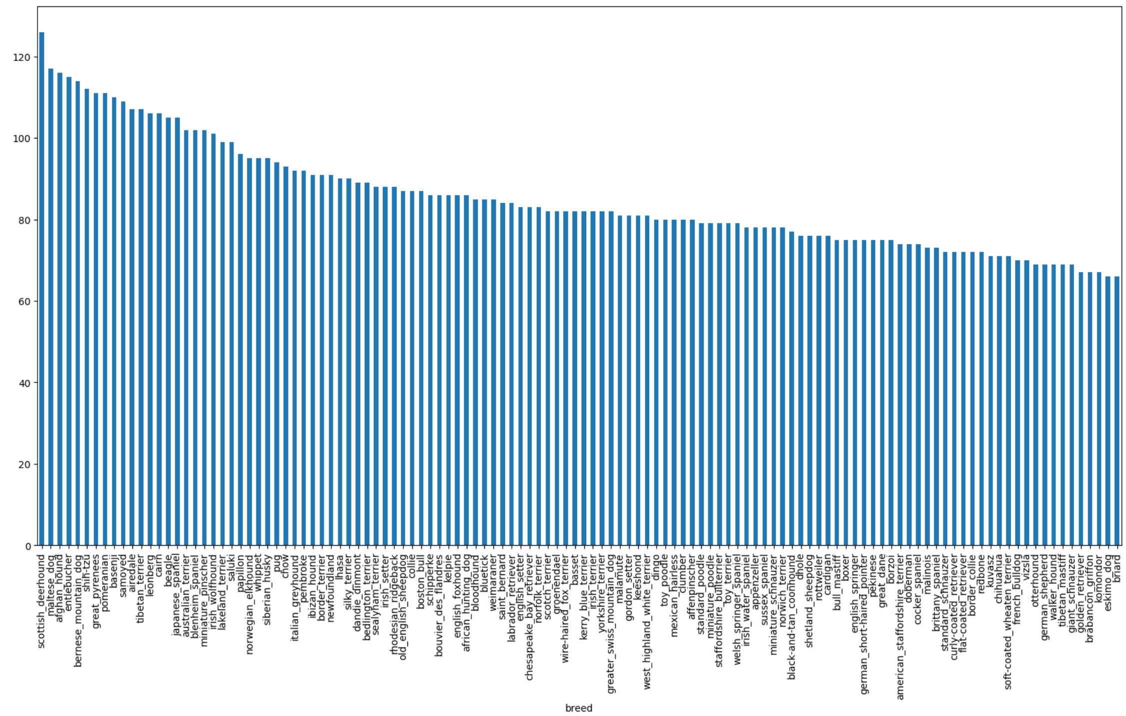
[5]:
```python
labels_csv.head()
```

[5]:

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

[6]:
```python
# how many images are there per breed?
labels_csv["breed"].value_counts().plot.bar(figsize= (20,10))
```

[6]:
```
<Axes: xlabel='breed'>
```




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.

[7]:
```python
from IPython.display import display, Image
# Image("drive/My Drive/Data/train/000bec180eb18c7604dcecc8fe0dba07.jpg")
```

## 0.3 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.

[8]:
```python
# Create pathnames from image ID's
filenames = ["drive/My Drive/Dog Vision/train/" + fname + ".jpg" for fname in␣
 ↪labels_csv["id"]]

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

[8]:
```
['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']
```

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

[9]:
```python
# let's view an image...
from IPython.display import Image
Image("/content/drive/MyDrive/Dog Vision/train/001cdf01b096e06d78e9e5112d419397.
 ↪jpg")
```

[9]:

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.

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

[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.

[11]:
```
# 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).

[12]:
```
# Find the unique label values
unique_breeds = np.unique(labels)
len(unique_breeds)
```

[12]:
```
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.

[13]:
```
# Example: Turn one label into an array of booleans
print(labels[0])
labels[3] == unique_breeds # use comparison operator to create boolean array
```

```
boston_bull
```

[13]:
```
array([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, False,
       False, False, False])
```

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

[14]:
```python
# Turn every label into a boolean array
boolean_labels = [label == np.array(unique_breeds) for label in labels]
boolean_labels[:2]
```

[14]:
```
[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.

[15]:
```python
# 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.

### 0.4 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.

[16]:
```python
# 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.

[17]:
```python
# Set number of images to use for experimenting
NUM_IMAGES = 1000 #@param {type:"slider", min:1000, max:10000, step:1000}
NUM_IMAGES
```

[17]:
```
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).

[18]:
```python
# 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)
```

[18]:
```
(800, 800, 200, 200)
```

[19]:
```
# Check out the training data (image file paths and labels)
X_train[:2], y_train[:2]
```

[19]:
```
(['drive/My Drive/Dog Vision/train/00bee065dcec471f26394855c5c2f3de.jpg',
  'drive/My Drive/Dog Vision/train/0d2f9e12a2611d911d91a339074c8154.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])])
```

### 0.5 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?

This is height, width, colour channel value. We're getting ahead of ourselves but that's the number of colour channels per pixel, red, green and blue.

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

[20]:
```
# 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
```

## 0.6 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.

[21]:
```
# 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.

[22]:
```
# 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
```

[23]:
```
# 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…
```

[24]:
```
# Check out the different attributes of our data batches
train_data.element_spec, val_data.element_spec
```

[24]:
```
((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.

## 0.7 Visualizing data batches

[25]:
```
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.

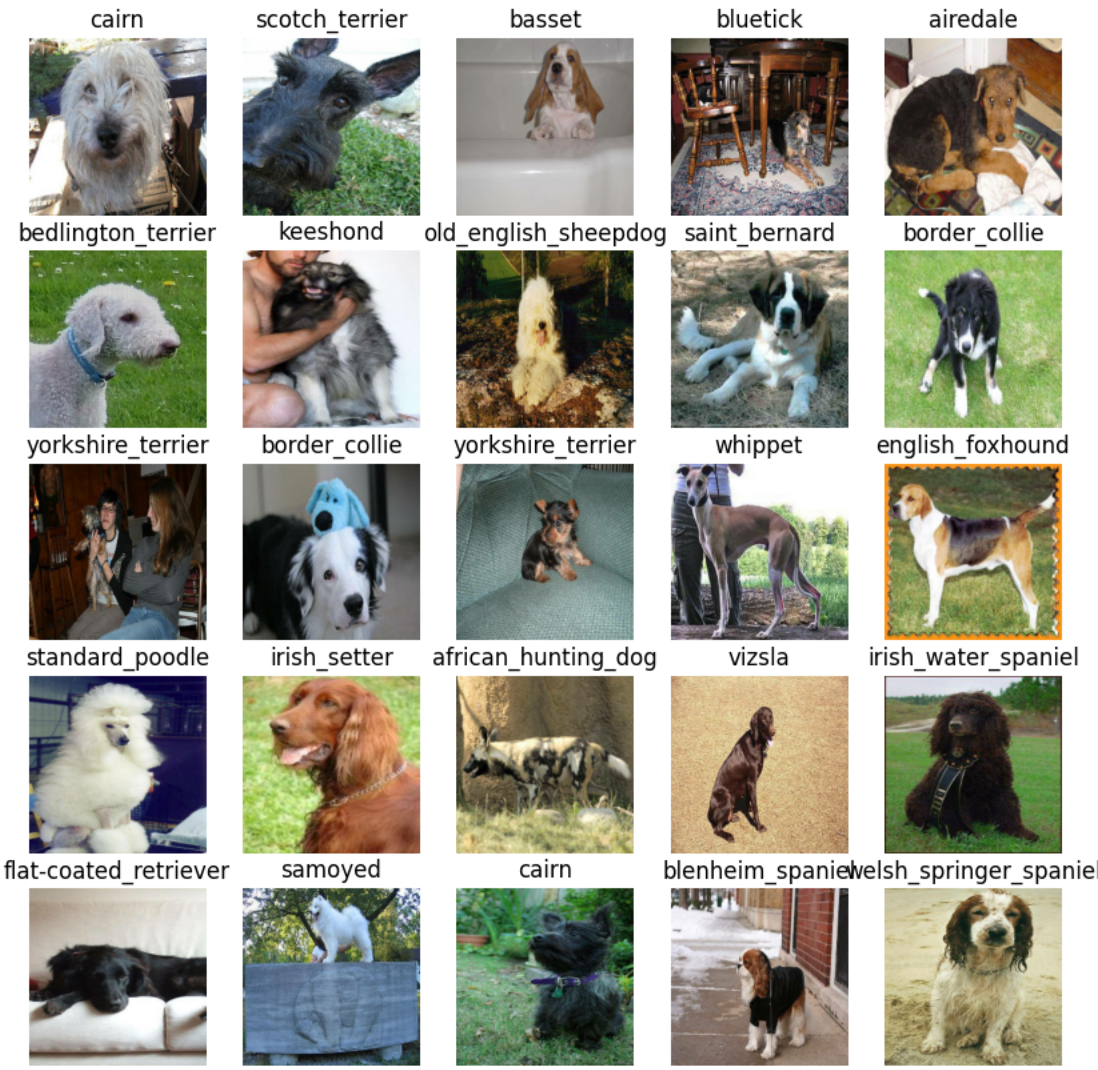
In our case, next will return a batch of 32 images and label pairs.

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

[26]:
```
# 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)
```


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

[27]:
```
# 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)
```




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

## 0.8 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.

### 0.8.1 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.

### 0.8.2 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.

## 0.9 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.

[28]:
```
# 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:

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

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

[29]:
```
# 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 guesses are)
      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?

### 0.10 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.

[30]:
```
# 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"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1001)              5432713

 dense (Dense)               (None, 120)               120240

=================================================================
Total params: 5552953 (21.18 MB)
Trainable params: 120240 (469.69 KB)
Non-trainable params: 5432713 (20.72 MB)
_________________________________________________________________
```

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.

## 0.11 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.

### 0.11.1 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).

[31]:
```
# Load the TensorBoard notebook extension
%load_ext tensorboard
```

```
[32]: import datetime
import os
# 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)
```

### 0.11.2 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."

```
[33]: # 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
```

## 0.12 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.

[34]:
```python
# 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!!!!)
```

[35]:
```python
# 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}
```

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.

[36]:
```python
# 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.

[37]:
```python
# Fit the model to the data
model = train_model()
```

```
Building model with:
```

```
https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/4
Epoch 1/100
25/25 [==============================] - 150s 6s/step - loss: 4.5153 - accuracy:
0.1100 - val_loss: 3.4772 - val_accuracy: 0.2400
Epoch 2/100
25/25 [==============================] - 3s 134ms/step - loss: 1.6194 -
accuracy: 0.6963 - val_loss: 2.1791 - val_accuracy: 0.4600
Epoch 3/100
25/25 [==============================] - 4s 160ms/step - loss: 0.5566 -
accuracy: 0.9388 - val_loss: 1.7263 - val_accuracy: 0.5650
Epoch 4/100
25/25 [==============================] - 5s 190ms/step - loss: 0.2549 -
accuracy: 0.9887 - val_loss: 1.5471 - val_accuracy: 0.6250
Epoch 5/100
25/25 [==============================] - 4s 160ms/step - loss: 0.1490 -
accuracy: 0.9950 - val_loss: 1.4500 - val_accuracy: 0.6400
Epoch 6/100
25/25 [==============================] - 4s 172ms/step - loss: 0.1005 -
accuracy: 0.9987 - val_loss: 1.3945 - val_accuracy: 0.6650
Epoch 7/100
25/25 [==============================] - 5s 182ms/step - loss: 0.0758 -
accuracy: 1.0000 - val_loss: 1.3664 - val_accuracy: 0.6500
Epoch 8/100
25/25 [==============================] - 3s 139ms/step - loss: 0.0597 -
accuracy: 1.0000 - val_loss: 1.3392 - val_accuracy: 0.6550
Epoch 9/100
25/25 [==============================] - 5s 190ms/step - loss: 0.0493 -
accuracy: 1.0000 - val_loss: 1.3191 - val_accuracy: 0.6650
```

### 0.12.1 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.

[38]:
```
%tensorboard --logdir drive/My\ Drive/Dog\ Vision/logs
```

```
<IPython.core.display.Javascript object>
```

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.

# 1 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.

[39]:
```
# 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 [==============================] - 1s 106ms/step
```

[39]:
```
array([[9.3639648e-04, 3.5975700e-05, 6.2587224e-03, …, 1.4431070e-03,
         5.0108763e-05, 7.2839893e-03],
        [4.5382637e-03, 5.4723484e-04, 1.8560402e-02, …, 3.1226252e-03,
         1.9307237e-03, 2.6382180e-03],
        [1.0691765e-05, 1.9314090e-05, 8.7581038e-06, …, 2.1326194e-05,
         4.8809481e-05, 5.0562179e-05],
        …,
        [6.2716186e-05, 1.2430151e-04, 2.9027327e-05, …, 1.1487178e-04,
         1.2554626e-04, 2.4237227e-06],
        [6.9612567e-04, 1.6897908e-04, 2.2735530e-04, …, 7.6423457e-05,
         6.2308951e-05, 3.3435635e-02],
        [1.9862785e-03, 6.0584334e-05, 1.9085219e-03, …, 5.5716425e-02,
         3.5628832e-03, 4.9403938e-05]], dtype=float32)
```

[40]:
```
# Check the shape of predictions
predictions.shape
```

[40]:
```
(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.

[41]:
```
# First prediction
print(predictions[0])
print(f"Max value (probability of prediction): {np.max(predictions[0])}") # the␣
↪max probability value predicted by the model
```

```python
print(f"Sum: {np.sum(predictions[0])}") # because we used softmax activation in our model, this will be close to 1
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
```

```
[9.36396478e-04 3.59756996e-05 6.25872239e-03 2.03615913e-04
 6.48430214e-05 5.68347350e-05 2.98845172e-02 4.91946936e-04
 5.47624113e-05 1.03878789e-03 1.66218006e-03 8.30548815e-04
 1.37342955e-03 5.93216500e-05 3.67297186e-03 2.55698164e-04
 7.98691035e-05 2.67828852e-01 6.64644758e-05 2.86838058e-05
 2.95436097e-04 8.82788081e-05 8.73638201e-05 3.92746320e-03
 2.22008366e-05 1.61564516e-04 1.07737280e-01 2.99510011e-05
 5.45523711e-04 6.02294866e-04 7.10464665e-05 1.00564922e-03
 1.61251193e-03 3.53995129e-05 3.96967189e-05 2.42921244e-02
 9.06769419e-05 1.26303028e-04 1.21945879e-04 7.59489849e-05
 9.46707465e-03 1.54575202e-04 1.21742181e-04 1.56699476e-04
 6.95166018e-05 1.00811798e-04 5.96391983e-05 1.96527195e-04
 1.03403209e-03 4.25819890e-05 6.41900333e-05 9.32145995e-05
 1.27373962e-04 3.08478629e-05 8.28760385e-05 4.08352644e-05
 9.84108192e-05 6.68742228e-03 7.68377620e-04 2.72395611e-01
 5.84065798e-04 2.22196468e-05 3.07083392e-04 1.24057478e-04
 3.98619159e-04 3.00422590e-02 1.91087282e-04 2.23970259e-04
 2.31570918e-02 1.11144916e-04 1.98491570e-02 1.54587367e-04
 2.77718442e-04 7.67735112e-03 7.96659500e-04 6.20621649e-05
 3.81404115e-03 1.55428154e-02 2.67232215e-04 2.89460458e-02
 2.21037524e-04 5.24319708e-03 6.99426600e-05 4.24403790e-03
 1.65933714e-04 1.89715368e-03 8.14099039e-04 1.77322450e-04
 3.00741638e-04 2.94073485e-04 3.93116934e-04 1.59251853e-04
 2.80145359e-05 3.24022723e-03 1.64403915e-04 5.54526516e-04
 1.53607529e-04 6.75155502e-03 2.19429538e-04 2.46980217e-05
 1.02002965e-02 2.19480899e-05 1.98370237e-02 8.70472565e-03
 1.76278758e-04 1.44417616e-04 2.66256705e-02 1.15293296e-04
 5.21793045e-05 5.54511743e-03 3.83600272e-04 7.59867718e-04
 3.36311350e-05 2.12635292e-04 8.89221614e-04 1.91391573e-05
 1.21906931e-02 1.44310703e-03 5.01087634e-05 7.28398934e-03]
Max value (probability of prediction): 0.27239561080932617
Sum: 1.0
Max index: 59
Predicted label: irish_wolfhound
```

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.

[42]:
```python
# 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
```

[42]:
```
'irish_wolfhound'
```

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.

[43]:
```python
# 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]
```

[43]:
```
(array([[[0.29599646, 0.43284872, 0.3056691 ],
         [0.26635826, 0.32996926, 0.22846507],
         [0.31428418, 0.27701408, 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 , 0.91253304, 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.

[44]:
```
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)
```

[45]:
```
# 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.

[46]:
```
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]
```

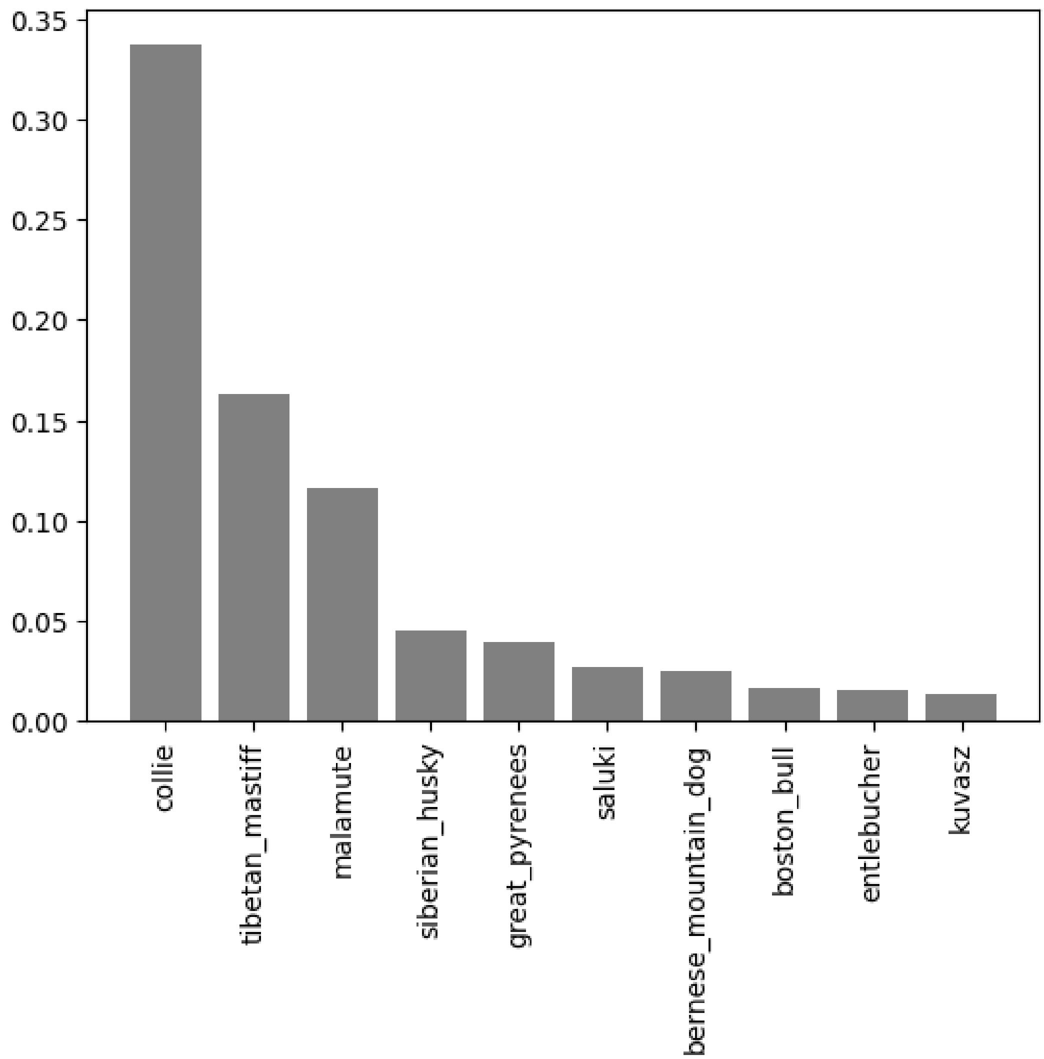
```
  # Get the predicted label
  pred_label = get_pred_label(pred_prob)

  # 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
```

[47]:
```
plot_pred_conf(prediction_probabilities=predictions,
               labels=val_labels,
               n=9)
```

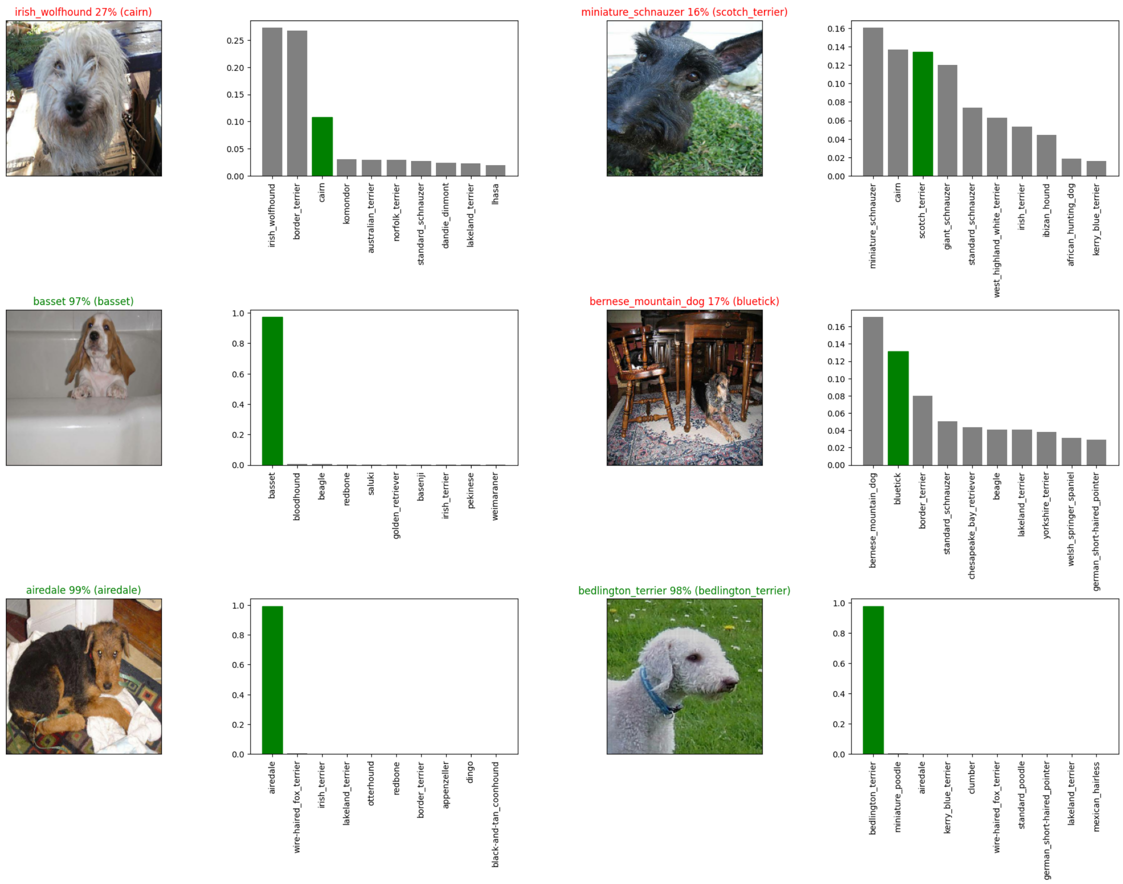



[48]:

```
# 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()
```




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

## 1.1 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.

[49]:
```
def save_model(model, suffix=None):
  """
  Saves a given model in a models directory and appends a suffix (str)
```

```
    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.

[50]:
```
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
```

[51]:
```
# Save our model trained on 1000 images
save_model(model, suffix="1000-images-Adam")
```

```
Saving model to: drive/My Drive/Data/models/20240627-07381719473909-1000-images-
Adam.h5…

/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103:
UserWarning: You are saving your model as an HDF5 file via `model.save()`. This
file format is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

[51]:
```
'drive/My Drive/Data/models/20240627-07381719473909-1000-images-Adam.h5'
```

[52]:
```
# Load our model trained on 1000 images
model_1000_images = load_model('/content/drive/MyDrive/Data/models/
 ↪20240624-08531719219190-1000-images-Adam.h5')
```

```
Loading saved model from:
/content/drive/MyDrive/Data/models/20240624-08531719219190-1000-images-Adam.h5
```

[53]:
```
# Evaluate the pre-saved model
model.evaluate(val_data)
```

```
7/7 [==============================] - 1s 110ms/step - loss: 1.3191 - accuracy:
0.6650
```

[53]:
```
[1.319081425666809, 0.6650000214576721]
```

### 1.2 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.

[54]:
```python
# Remind ourselves of the size of the full dataset
len(X), len(y)
```

[54]:
```
(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).

[55]:
```python
# 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.

[56]:
```python
# 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.

[57]:
```python
#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
```

```python
# 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).

[58]:
```
%tensorboard --logdir drive/My\ Drive/Data/logs
```

```
<IPython.core.display.Javascript object>
```

**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.

[59]:
```python
# 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])
```

```
Epoch 1/100
320/320 [==============================] - 1445s 5s/step - loss: 1.3492 -
accuracy: 0.6679
Epoch 2/100
320/320 [==============================] - 41s 130ms/step - loss: 0.4021 -
accuracy: 0.8806
Epoch 3/100
320/320 [==============================] - 43s 133ms/step - loss: 0.2371 -
accuracy: 0.9359
Epoch 4/100
320/320 [==============================] - 43s 135ms/step - loss: 0.1551 -
accuracy: 0.9617
Epoch 5/100
320/320 [==============================] - 43s 135ms/step - loss: 0.1072 -
accuracy: 0.9776
Epoch 6/100
320/320 [==============================] - 42s 130ms/step - loss: 0.0776 -
accuracy: 0.9872
Epoch 7/100
320/320 [==============================] - 45s 140ms/step - loss: 0.0584 -
accuracy: 0.9923
Epoch 8/100
320/320 [==============================] - 41s 127ms/step - loss: 0.0461 -
accuracy: 0.9943
```

```
Epoch 9/100
320/320 [==============================] - 41s 129ms/step - loss: 0.0369 -
accuracy: 0.9962
Epoch 10/100
320/320 [==============================] - 42s 131ms/step - loss: 0.0312 -
accuracy: 0.9965
Epoch 11/100
320/320 [==============================] - 44s 136ms/step - loss: 0.0272 -
accuracy: 0.9977
Epoch 12/100
320/320 [==============================] - 43s 133ms/step - loss: 0.0222 -
accuracy: 0.9985
Epoch 13/100
320/320 [==============================] - 41s 129ms/step - loss: 0.0199 -
accuracy: 0.9986
Epoch 14/100
320/320 [==============================] - 43s 135ms/step - loss: 0.0189 -
accuracy: 0.9983
Epoch 15/100
320/320 [==============================] - 43s 134ms/step - loss: 0.0150 -
accuracy: 0.9988
Epoch 16/100
320/320 [==============================] - 43s 133ms/step - loss: 0.0158 -
accuracy: 0.9982
Epoch 17/100
320/320 [==============================] - 43s 133ms/step - loss: 0.0121 -
accuracy: 0.9986
Epoch 18/100
320/320 [==============================] - 42s 130ms/step - loss: 0.0109 -
accuracy: 0.9989
Epoch 19/100
320/320 [==============================] - 42s 132ms/step - loss: 0.0126 -
accuracy: 0.9988
Epoch 20/100
320/320 [==============================] - 42s 131ms/step - loss: 0.0107 -
accuracy: 0.9989
Epoch 21/100
320/320 [==============================] - 43s 135ms/step - loss: 0.0097 -
accuracy: 0.9989
```

```
[59]: <keras.src.callbacks.History at 0x7d9383675930>
```

### 1.3 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.

[60]:
```
# Save model to file
save_model(full_model, suffix="all-images-Adam")
```

```
Saving model to: drive/My Drive/Data/models/20240627-08241719476650-all-images-
Adam.h5…
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/engine/training.py:3103:
UserWarning: You are saving your model as an HDF5 file via `model.save()`. This
file format is considered legacy. We recommend using instead the native Keras
format, e.g. `model.save('my_model.keras')`.
  saving_api.save_model(
```

[60]:
```
'drive/My Drive/Data/models/20240627-08241719476650-all-images-Adam.h5'
```

[62]:
```
# Load in the full model
loaded_full_model = load_model('drive/My Drive/Data/models/
 ↪20240627-08241719476650-all-images-Adam.h5')
```

```
Loading saved model from: drive/My
Drive/Data/models/20240627-08241719476650-all-images-Adam.h5
```

### 1.4 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.

[67]:
```
# Load test image filenames (since we're using os.listdir(), these already have␣
 ↪.jpg)
test_path = "drive/MyDrive/Dog Vision/test/"
test_filenames = [test_path + fname for fname in os.listdir(test_path)]

test_filenames[:10]
```

[67]:
```
['drive/MyDrive/Dog Vision/test/eaa65260eb9a2f7d3b5484ac97962788.jpg',
 'drive/MyDrive/Dog Vision/test/eb0f81618a71ccf82982d70879464d89.jpg',
 'drive/MyDrive/Dog Vision/test/f13508048aa548af116601fa59bd4c62.jpg',
 'drive/MyDrive/Dog Vision/test/f1705303b30da3a48db2a5f34376b947.jpg',
 'drive/MyDrive/Dog Vision/test/effcbcfa58ba87eac8439b7106d8623c.jpg',
```

```
 'drive/MyDrive/Dog Vision/test/f062edd0025c4ecad1a34125ee6a761a.jpg',
 'drive/MyDrive/Dog Vision/test/ec429df7ceab9763e4e790b569997346.jpg',
 'drive/MyDrive/Dog Vision/test/ebbbbe14c3e968a8f73490d024472768.jpg',
 'drive/MyDrive/Dog Vision/test/e982fc6397c1f837100f28a8ed26d86e.jpg',
 'drive/MyDrive/Dog Vision/test/e91ffd67dd303f59029d041ff4fb65b8.jpg']
```

[68]:
```
# How many test images are there?
len(test_filenames)
```

[68]:
```
10357
```

[69]:
```
# 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.

[70]:
```
# Make predictions on test data batch using the loaded full model
test_predictions = loaded_full_model.predict(test_data,
                                             verbose=1)
```

```
324/324 [==============================] - 232s 702ms/step
```

[71]:
```
# Check out the test predictions
test_predictions[:10]
```

[71]:
```
array([[3.2875896e-08, 2.3277396e-12, 3.9047571e-08, …, 7.5094366e-12,
        2.7013922e-10, 2.2670082e-10],
       [3.6641207e-11, 8.1492608e-12, 6.5499950e-10, …, 8.6160042e-05,
        1.2211482e-11, 1.6836145e-06],
       [4.8045918e-11, 2.5765463e-13, 5.2686613e-07, …, 7.8086181e-05,
        9.0008204e-13, 1.2732207e-08],
       …,
       [4.7383864e-08, 9.3116846e-08, 7.0281541e-09, …, 3.2573627e-08,
        9.4455629e-07, 6.4458593e-04],
       [5.7396260e-16, 9.0670648e-12, 3.8036926e-14, …, 6.5204495e-14,
        1.7354178e-12, 1.4861841e-11],
       [1.5184834e-10, 2.6253907e-10, 7.3410328e-10, …, 2.5668125e-08,
        2.6383420e-10, 1.9472661e-10]], dtype=float32)
```

## 1.5 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.

[72]:
```python
# Create pandas DataFrame with empty columns
preds_df = pd.DataFrame(columns=["id"] + list(unique_breeds))
preds_df.head()
```

[72]:
```
Empty DataFrame
Columns: [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_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, rhodesian_ridgeback, rottweiler,
saint_bernard, saluki, samoyed, schipperke, scotch_terrier, scottish_deerhound,
sealyham_terrier, …]
Index: []

[0 rows x 121 columns]
```

[75]:
```python
# Append test image ID's to predictions DataFrame
test_path = "drive/MyDrive/Dog Vision/test/"
preds_df["id"] = [os.path.splitext(path)[0] for path in os.listdir(test_path)]
preds_df.head()
```

[75]:
```
                                 id  affenpinscher  afghan_hound  \
0  eaa65260eb9a2f7d3b5484ac97962788   3.287590e-08  2.327740e-12
1  eb0f81618a71ccf82982d70879464d89   3.664121e-11  8.149261e-12
2  f13508048aa548af116601fa59bd4c62   4.804592e-11  2.576546e-13
3  f1705303b30da3a48db2a5f34376b947   6.297410e-15  1.330738e-06
4  effcbcfa58ba87eac8439b7106d8623c   2.609200e-05  2.752249e-06
```

```
   african_hunting_dog      airedale  american_staffordshire_terrier  \
0         3.904757e-08  3.030017e-11                    2.613368e-11
1         6.549995e-10  1.062621e-10                    7.468612e-02
2         5.268661e-07  4.852466e-10                    6.602772e-10
3         1.112689e-09  2.378535e-07                    4.402898e-07
4         3.104307e-08  5.195176e-06                    3.458213e-08

    appenzeller  australian_terrier       basenji        basset  …  \
0  1.204307e-13        1.203418e-06  1.061296e-10  5.831188e-13  …
1  2.672065e-11        5.775407e-12  2.256344e-08  1.272009e-10  …
2  7.096934e-15        4.804425e-08  4.109623e-08  1.672333e-10  …
3  3.982718e-08        1.319850e-11  4.697857e-12  4.909054e-10  …
4  5.058706e-06        4.821976e-08  1.078744e-07  7.674207e-05  …

     toy_poodle   toy_terrier        vizsla  walker_hound    weimaraner  \
0  6.405312e-12  9.678727e-10  4.969217e-14  5.795595e-13  5.811393e-12
1  2.291536e-07  1.363248e-08  1.080994e-04  6.408765e-11  1.590602e-10
2  1.009743e-08  4.149468e-09  8.254673e-09  7.310380e-08  2.077918e-08
3  1.927377e-10  1.186103e-10  7.494622e-02  1.618104e-07  9.229465e-01
4  2.586232e-06  5.975171e-08  6.933097e-08  2.448857e-07  1.718309e-08

   welsh_springer_spaniel  west_highland_white_terrier       whippet  \
0            7.599647e-13                 9.404764e-08  7.509437e-12
1            1.551529e-09                 3.670167e-13  8.616004e-05
2            1.152183e-07                 3.997749e-09  7.808618e-05
3            5.283622e-13                 8.268629e-13  7.093327e-05
4            2.170761e-07                 9.284037e-09  9.764519e-09

   wire-haired_fox_terrier  yorkshire_terrier
0             2.701392e-10       2.267008e-10
1             1.221148e-11       1.683615e-06
2             9.000820e-13       1.273221e-08
3             3.712220e-10       2.266503e-08
4             1.188712e-06       1.010993e-05

[5 rows x 121 columns]
```

[76]:
```python
# Add the prediction probabilities to each dog breed column
preds_df[list(unique_breeds)] = test_predictions
preds_df.head()
```

[76]:
```
                                 id  affenpinscher  afghan_hound  \
0  eaa65260eb9a2f7d3b5484ac97962788   3.287590e-08  2.327740e-12
1  eb0f81618a71ccf82982d70879464d89   3.664121e-11  8.149261e-12
2  f13508048aa548af116601fa59bd4c62   4.804592e-11  2.576546e-13
3  f1705303b30da3a48db2a5f34376b947   6.297410e-15  1.330738e-06
4  effcbcfa58ba87eac8439b7106d8623c   2.609200e-05  2.752249e-06
```

```
   african_hunting_dog      airedale  american_staffordshire_terrier  \
0         3.904757e-08  3.030017e-11                    2.613368e-11
1         6.549995e-10  1.062621e-10                    7.468612e-02
2         5.268661e-07  4.852466e-10                    6.602772e-10
3         1.112689e-09  2.378535e-07                    4.402898e-07
4         3.104307e-08  5.195176e-06                    3.458213e-08

    appenzeller  australian_terrier       basenji        basset  …  \
0  1.204307e-13        1.203418e-06  1.061296e-10  5.831188e-13  …
1  2.672065e-11        5.775407e-12  2.256344e-08  1.272009e-10  …
2  7.096934e-15        4.804425e-08  4.109623e-08  1.672333e-10  …
3  3.982718e-08        1.319850e-11  4.697857e-12  4.909054e-10  …
4  5.058706e-06        4.821976e-08  1.078744e-07  7.674207e-05  …

     toy_poodle   toy_terrier        vizsla  walker_hound    weimaraner  \
0  6.405312e-12  9.678727e-10  4.969217e-14  5.795595e-13  5.811393e-12
1  2.291536e-07  1.363248e-08  1.080994e-04  6.408765e-11  1.590602e-10
2  1.009743e-08  4.149468e-09  8.254673e-09  7.310380e-08  2.077918e-08
3  1.927377e-10  1.186103e-10  7.494622e-02  1.618104e-07  9.229465e-01
4  2.586232e-06  5.975171e-08  6.933097e-08  2.448857e-07  1.718309e-08

   welsh_springer_spaniel  west_highland_white_terrier       whippet  \
0            7.599647e-13                 9.404764e-08  7.509437e-12
1            1.551529e-09                 3.670167e-13  8.616004e-05
2            1.152183e-07                 3.997749e-09  7.808618e-05
3            5.283622e-13                 8.268629e-13  7.093327e-05
4            2.170761e-07                 9.284037e-09  9.764519e-09

   wire-haired_fox_terrier  yorkshire_terrier
0             2.701392e-10       2.267008e-10
1             1.221148e-11       1.683615e-06
2             9.000820e-13       1.273221e-08
3             3.712220e-10       2.266503e-08
4             1.188712e-06       1.010993e-05

[5 rows x 121 columns]
```

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

[77]:
```
preds_df.to_csv("drive/My Drive/Data/full_submission_1_mobilienetV2_adam.csv",
                 index=False)
```