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

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

## 1. Problem

Identiifying the breeds of a dog given an imahe of a dog.

When I'm going on streets 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/competitions/dog-breed-identification/data>

## 3. Evaluation

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

<https://www.kaggle.com/competitions/dog-breed-identification/overview/evaluation>

## 4. Features

Some informations about the data:

* We're dealing with images (unstructured data) so itis probably best we use deep learning/transfer learning.
* There are 120 breed 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)

from google.colab import drive  
drive.mount('/content/drive')

Mounted at /content/drive

# Unzip the zip data file  
# !unzip "drive/MyDrive/Dog Vision/dog-breed-identification.zip" -d "drive/MyDrive/Dog Vision/"

### Get Our workspace ready

* Import TensorFlow 2✅
* Import TensorFlow Hub✅
* Make sure we're using a GPU✅

# Import necessary tools  
import tensorflow as tf  
import tensorflow\_hub as hub  
print("TF version:", tf.\_\_version\_\_)  
print("TF Hub version:", hub.\_\_version\_\_)  
  
# Check for GPU availability  
print("GPU", "available (YESS!)" if tf.config.list\_physical\_devices("GPU") else "not available :(")

TF version: 2.12.0  
TF Hub version: 0.14.0  
GPU available (YESS!)

## Getting our data ready (truning into Tensors)

With 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/MyDrive/Dog Vision/labels.csv")  
print(labels\_csv.describe())  
print(labels\_csv.head())

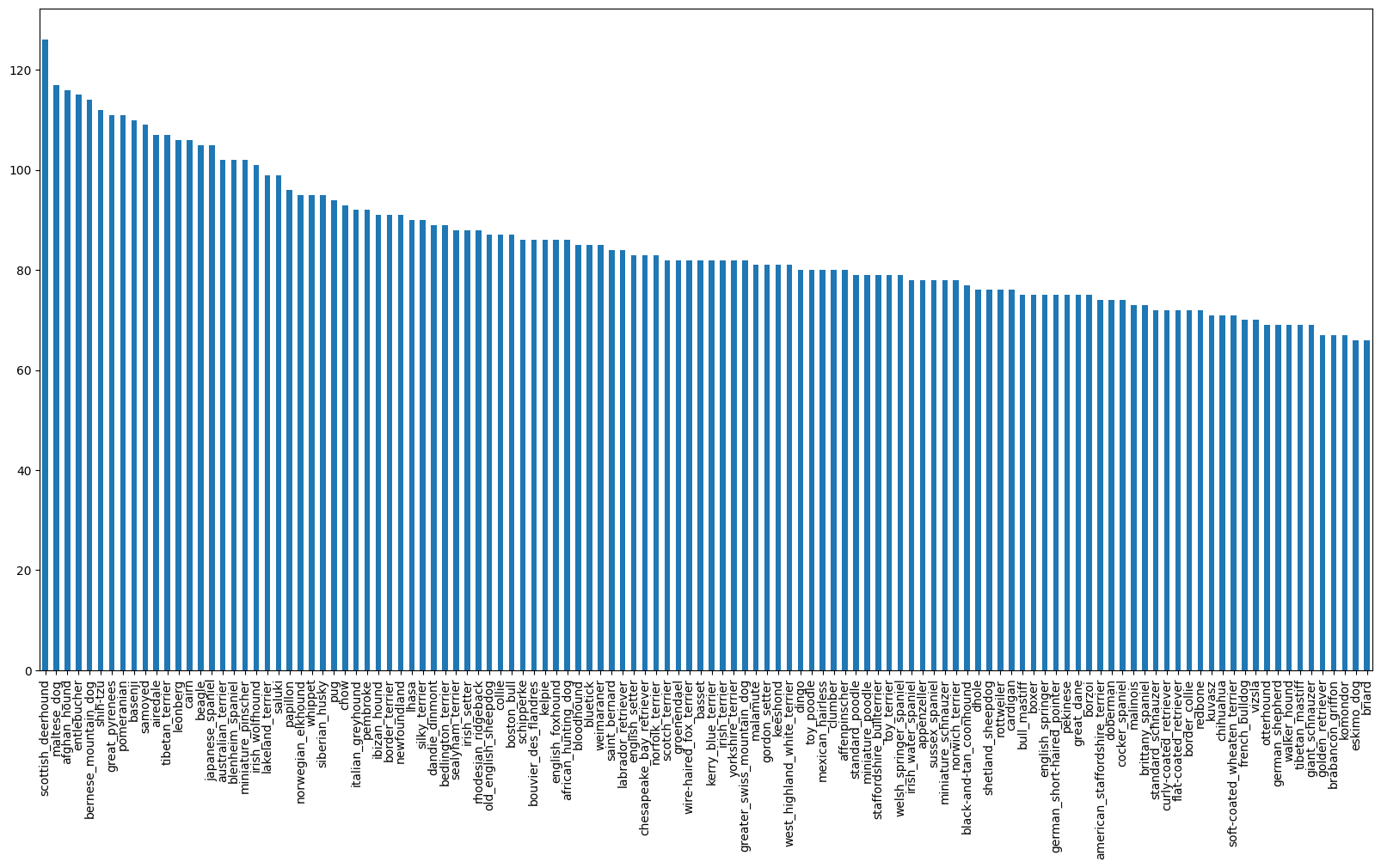
id breed  
count 10222 10222  
unique 10222 120  
top 000bec180eb18c7604dcecc8fe0dba07 scottish\_deerhound  
freq 1 126  
 id breed  
0 000bec180eb18c7604dcecc8fe0dba07 boston\_bull  
1 001513dfcb2ffafc82cccf4d8bbaba97 dingo  
2 001cdf01b096e06d78e9e5112d419397 pekinese  
3 00214f311d5d2247d5dfe4fe24b2303d bluetick  
4 0021f9ceb3235effd7fcde7f7538ed62 golden\_retriever

labels\_csv.head()

id breed  
0 000bec180eb18c7604dcecc8fe0dba07 boston\_bull  
1 001513dfcb2ffafc82cccf4d8bbaba97 dingo  
2 001cdf01b096e06d78e9e5112d419397 pekinese  
3 00214f311d5d2247d5dfe4fe24b2303d bluetick  
4 0021f9ceb3235effd7fcde7f7538ed62 golden\_retriever

# How many images are there of each breed?  
labels\_csv["breed"].value\_counts().plot.bar(figsize = (20, 10))

<Axes: >



labels\_csv["breed"].value\_counts().median()

82.0

# Let's view an image  
from IPython.display import Image  
Image("drive/MyDrive/Dog Vision/train/00214f311d5d2247d5dfe4fe24b2303d.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 images ID's  
filenames = ["drive/MyDrive/Dog Vision/train/" + fname + ".jpg" for fname in labels\_csv["id"]]  
  
# Check the first 10 file names  
filenames[:10]

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

# Check whether number of file names matches number of image files  
import os  
if len(os.listdir("drive/MyDrive/Dog Vision/train/")) == len(filenames):  
 print("Filenames match actual number of files! Proceed")  
else:  
 print("Filename doesn't match actual number of files, check the target directory!")

Filenames match actual number of files! Proceed

# One more check  
Image(filenames[8000])



labels\_csv["breed"][8000]

{"type":"string"}

type(filenames)

list

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)

len(labels)

10222

# 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 doesn't matches number of filenames. Check data diretctories!")

Number of labels matches number 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',  
 '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',  
 'shetland\_sheepdog', 'shih-tzu', 'siberian\_husky', 'silky\_terrier',  
 'soft-coated\_wheaten\_terrier', 'staffordshire\_bullterrier',  
 'standard\_poodle', 'standard\_schnauzer', 'sussex\_spaniel',  
 'tibetan\_mastiff', 'tibetan\_terrier', 'toy\_poodle', 'toy\_terrier',  
 'vizsla', 'walker\_hound', 'weimaraner', 'welsh\_springer\_spaniel',  
 'west\_highland\_white\_terrier', 'whippet',  
 'wire-haired\_fox\_terrier', 'yorkshire\_terrier'], dtype=object)

len(unique\_breeds)

120

# Turn a single label into an array of booleans  
print(labels[0])  
labels[0] == unique\_breeds

boston\_bull

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

# Turn every label into a boolean array  
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,  
 False, False, False, False, False, False, False, False, False,  
 False, False, False, False, False, False, False, False, False,  
 False, False, False, False, False, False, False, False, False,  
 False, False, False, False, False, False, False, False, False,  
 False, False, False])]

labels[1]

{"type":"string"}

len(boolean\_labels)

10222

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

boston\_bull  
(array([19]),)  
19  
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0]

print(labels[2])  
print(boolean\_labels[2].astype(int))

pekinese  
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 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]

filenames[:2]

['drive/MyDrive/Dog Vision/train/000bec180eb18c7604dcecc8fe0dba07.jpg',  
 'drive/MyDrive/Dog Vision/train/001513dfcb2ffafc82cccf4d8bbaba97.jpg']

boolean\_labels[:2]

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

### 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 and y variables  
X = filenames  
y = boolean\_labels

len(filenames)

10222

we're going to start off experimenting with ~1000 imaegs and increase as needed..

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

# Lets split our data into traina and validation sets  
from sklearn.model\_selection import train\_test\_split  
  
# Split them into training and validation of total size NUM\_IMAGES  
X\_train, X\_val, y\_train, y\_val = train\_test\_split(X[:NUM\_IMAGES], y[:NUM\_IMAGES], test\_size = 0.2, random\_state = 42)  
  
len(X\_train), len(X\_val), len(y\_train), len(y\_val)

(800, 200, 800, 200)

# Lets have a geez at training data  
X\_train[:2], y\_train[:2]

(['drive/MyDrive/Dog Vision/train/00bee065dcec471f26394855c5c2f3de.jpg',  
 'drive/MyDrive/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])])

## 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 file and save it to a variable, image
3. Turn ouimage (a jpg) into Tensors
4. Resize the image to be a shape of (224, 224)
5. Return the modified image

## Before we do, lets see what importing an image looks like...

# Convert an image into Numpy array  
from matplotlib.pyplot import imread  
image = imread(filenames[42])  
image.shape

(257, 350, 3)

image[:2]

array([[[ 89, 137, 89],  
 [ 76, 124, 76],  
 [ 63, 111, 61],  
 ...,  
 [ 77, 133, 86],  
 [ 76, 134, 86],  
 [ 76, 134, 86]],  
  
 [[ 72, 119, 75],  
 [ 67, 114, 68],  
 [ 63, 110, 64],  
 ...,  
 [ 75, 131, 84],  
 [ 74, 132, 84],  
 [ 74, 132, 84]]], dtype=uint8)

# Turn image into Tensors  
tf.constant(image)[:2]

<tf.Tensor: shape=(2, 350, 3), dtype=uint8, numpy=  
array([[[ 89, 137, 89],  
 [ 76, 124, 76],  
 [ 63, 111, 61],  
 ...,  
 [ 77, 133, 86],  
 [ 76, 134, 86],  
 [ 76, 134, 86]],  
  
 [[ 72, 119, 75],  
 [ 67, 114, 68],  
 [ 63, 110, 64],  
 ...,  
 [ 75, 131, 84],  
 [ 74, 132, 84],  
 [ 74, 132, 84]]], dtype=uint8)>

image.max(), image.min()

(255, 0)

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 file and save it to a variable, image
3. Turn our image (a jpg) into Tensors
4. Normallize our image (convert colour channel values to 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):  
 """  
 Take an image file path and turns the image into a Tensor.  
 """  
 # Read in an image file  
 image = tf.io.read\_file(image\_path)  
 # Turn jpg image into numerical Tensor with 3 colour channels (Red, Green, Blue)  
 image = tf.image.decode\_jpeg(image, channels=3)  
 # convert the colour channel values to from 0-255 to 0-1 values  
 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[2])  
# tensor

# tensor = tf.image.decode\_jpeg(tensor, channels=3)

# tf.image.convert\_image\_dtype(tensor, tf.float32)

## Turning our data into branches

Why turn our data into batches?

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

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

in order to use TensorFlow effectively, we need our data in the form of Tensor tuple which looks like this: (image, labels)

# Create a simple function to create a tuple (image, labels)  
def get\_image\_label(image\_path, label):  
 """  
 Takes an image file path name and the associated label,  
 processes the image and returns a tuple if (image, label).  
 """  
  
 image = process\_image(image\_path)  
 return image, label

# Demo of the 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],  
 [0.05441074, 0.07435255, 0.05428263],  
 [0.04734282, 0.07581793, 0.02060942],  
 ...,  
 [0.3397559 , 0.31265694, 0.14725602],  
 [0.387725 , 0.360274 , 0.18714729],  
 [0.43941984, 0.41196886, 0.23884216]]], dtype=float32)>,  
 <tf.Tensor: shape=(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,  
 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])>)

Now we've got a wa to turn our data into tuples of Tensors in the form (image, label), lets make a function to turn all of our data(X and y) into batches.

# Define the batch size, 32 is a good size  
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):  
 """  
 Create batches of data out of image (X) and label (y) pairs.  
 Shuffles the data if it's training data but deosn't shuffles 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  
 if valid\_data:  
 print("Creating validation data batches....")  
 data = tf.data.Dataset.from\_tensor\_slices((tf.constant(X), # filepaths  
 tf.constant(y))) # labels  
 data\_batch = data.map(get\_image\_label).batch(BATCH\_SIZE)  
 return data\_batch  
  
 else:  
 print("Creating training data batches....")  
 # Turn filepaths and labels into Tensors  
 data = tf.data.Dataset.from\_tensor\_slices((tf.constant(X), tf.constant(y)))  
  
 # Shuffling pathnames and labels before mapping image processor function is faster than shuffling images  
 data = data.shuffle(buffer\_size = len(X))  
  
 # Create (image, label) tuples (this also turns the image path into a preprocessed image)  
 data = data.map(get\_image\_label)  
  
 # Turn the training data into batches  
 data\_batch = data.batch(BATCH\_SIZE)  
 return data\_batch

# Create training and validation data batches  
train\_data = create\_data\_batches(X\_train, y\_train)  
val\_data = create\_data\_batches(X\_val, y\_val, valid\_data = True)

Creating training data batches....  
Creating validation data batches....

# Check out the diff 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

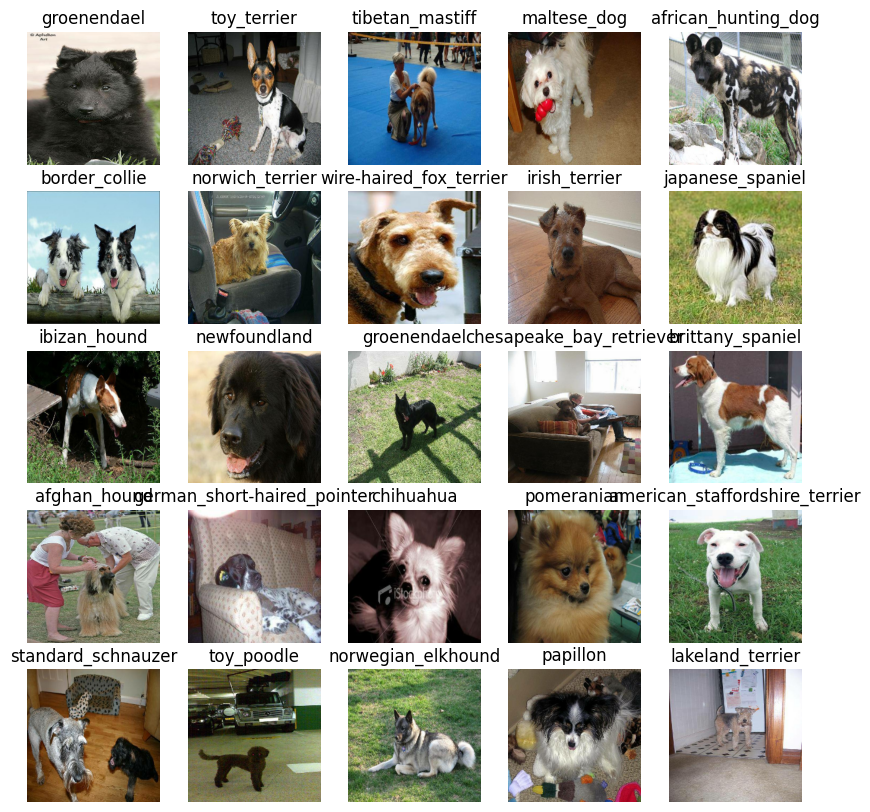
Our data is now in batche, however, these can be little hard to understand/comprehaend, let's visualize them!

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 fugure  
 plt.figure(figsize = (10,10))  
 # Loop through 25 (for displaying 25 images)  
 for i in range(25):  
 # Create subplots  
 ax = plt.subplot(5, 5, i+1)  
 # Display an image  
 plt.imshow(images[i])  
 # Add the image label as the title  
 plt.title(unique\_breeds[labels[i].argmax()])  
 # turn the grid lines off  
 plt.axis("off")

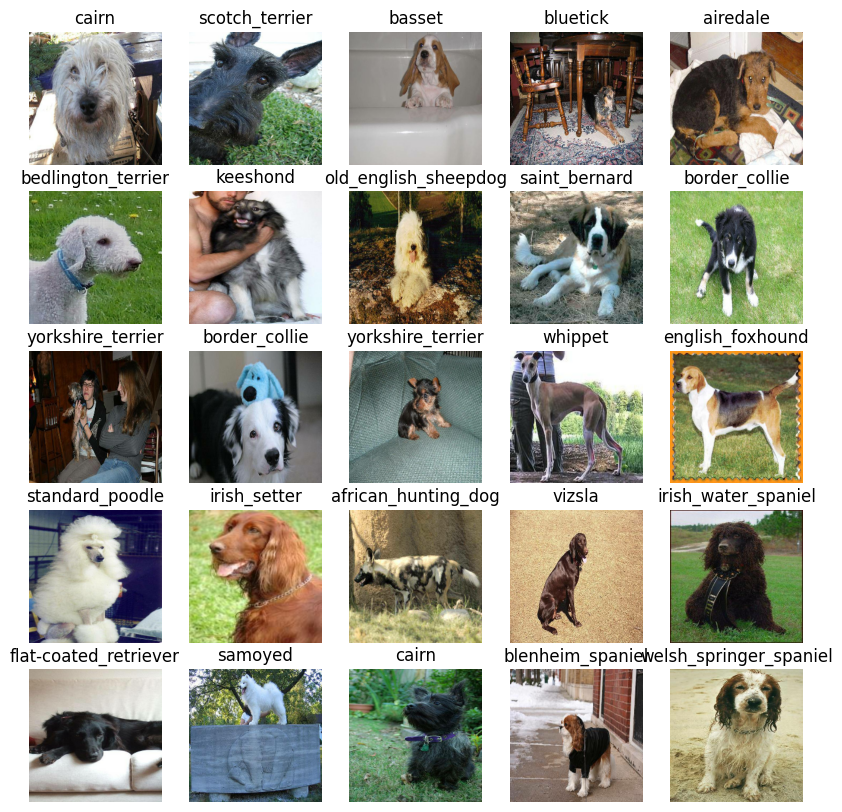
train\_data

<\_BatchDataset element\_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))>

# Now let's visualize the data in training batch  
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/5>

IMG\_SIZE

224

# Setup input shape to the model  
INPUT\_SHAPE = [None, IMG\_SIZE, IMG\_SIZE, 3] # batch, height, width, colour channels  
  
# Setup output shape of our model  
OUTPUT\_SHAPE = len(unique\_breeds)  
  
# Setup model URL from TensorFlow Hub  
MODEL\_URL = "https://tfhub.dev/google/imagenet/mobilenet\_v2\_130\_224/classification/5"

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>

# Create a function which buildsa a Keras model  
def create\_model(input\_shape = INPUT\_SHAPE, output\_shape = OUTPUT\_SHAPE, model\_url = MODEL\_URL):  
 print("Building model with:", MODEL\_URL)  
 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

model = create\_model()  
model.summary()

Building model with: https://tfhub.dev/google/imagenet/mobilenet\_v2\_130\_224/classification/5  
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  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

outputs = np.ones(shape = (1,1,1280))  
outputs

array([[[1., 1., 1., ..., 1., 1., 1.]]])

## Create callbacks

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

We'll create two callbacks, one for TensorBoard wich helps track out models progress and another for early stopping which prevents out model from training for too long.

### TenosorBoard Callbacks

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

1. Load the TensorBoard notebook extension ✅
2. Create a TensorBoard callback which is able to save logs to a directory and passs it to our models's fit() function.✅
3. Visualize our models training logs with the %tensorboard magic function (well do this after model training).

# Load TensorBoard notebook extension  
%reload\_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/MyDrive/Dog Vision/logs",  
 # Make it so that the logs get tracked whenevr 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 evaluation metric stops improving.

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

# Create early stopping callbacks  
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 aure everything is working

NUM\_EPOCHS = 100 #@param {type:"slider", min:10, max:100, step:10}

# Check to make sure we're still running on a GPU  
print("GPU", "Available (Yessss!)" if tf.config.list\_physical\_devices("GPU") else "not available")

GPU Available (Yessss!)

Lets 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 model.  
 """  
 # 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/5  
Epoch 1/100  
25/25 [==============================] - 122s 3s/step - loss: 4.4341 - accuracy: 0.1200 - val\_loss: 3.3352 - val\_accuracy: 0.2650  
Epoch 2/100  
25/25 [==============================] - 3s 123ms/step - loss: 1.5977 - accuracy: 0.6888 - val\_loss: 2.0857 - val\_accuracy: 0.5000  
Epoch 3/100  
25/25 [==============================] - 3s 134ms/step - loss: 0.5393 - accuracy: 0.9463 - val\_loss: 1.6075 - val\_accuracy: 0.6150  
Epoch 4/100  
25/25 [==============================] - 3s 129ms/step - loss: 0.2399 - accuracy: 0.9912 - val\_loss: 1.4124 - val\_accuracy: 0.6600  
Epoch 5/100  
25/25 [==============================] - 4s 146ms/step - loss: 0.1395 - accuracy: 0.9975 - val\_loss: 1.3485 - val\_accuracy: 0.6500  
Epoch 6/100  
25/25 [==============================] - 4s 161ms/step - loss: 0.0966 - accuracy: 1.0000 - val\_loss: 1.3025 - val\_accuracy: 0.6700  
Epoch 7/100  
25/25 [==============================] - 3s 123ms/step - loss: 0.0730 - accuracy: 1.0000 - val\_loss: 1.2685 - val\_accuracy: 0.6700  
Epoch 8/100  
25/25 [==============================] - 3s 122ms/step - loss: 0.0583 - accuracy: 1.0000 - val\_loss: 1.2558 - val\_accuracy: 0.6800  
Epoch 9/100  
25/25 [==============================] - 4s 151ms/step - loss: 0.0479 - accuracy: 1.0000 - val\_loss: 1.2312 - val\_accuracy: 0.6900  
Epoch 10/100  
25/25 [==============================] - 5s 198ms/step - loss: 0.0405 - accuracy: 1.0000 - val\_loss: 1.2190 - val\_accuracy: 0.6900  
Epoch 11/100  
25/25 [==============================] - 3s 120ms/step - loss: 0.0350 - accuracy: 1.0000 - val\_loss: 1.2015 - val\_accuracy: 0.6900  
Epoch 12/100  
25/25 [==============================] - 3s 120ms/step - loss: 0.0305 - accuracy: 1.0000 - val\_loss: 1.1975 - val\_accuracy: 0.7000  
Epoch 13/100  
25/25 [==============================] - 4s 181ms/step - loss: 0.0269 - accuracy: 1.0000 - val\_loss: 1.1868 - val\_accuracy: 0.7000  
Epoch 14/100  
25/25 [==============================] - 3s 124ms/step - loss: 0.0240 - accuracy: 1.0000 - val\_loss: 1.1788 - val\_accuracy: 0.7050  
Epoch 15/100  
25/25 [==============================] - 3s 123ms/step - loss: 0.0216 - accuracy: 1.0000 - val\_loss: 1.1726 - val\_accuracy: 0.7050  
Epoch 16/100  
25/25 [==============================] - 4s 148ms/step - loss: 0.0195 - accuracy: 1.0000 - val\_loss: 1.1685 - val\_accuracy: 0.7050  
Epoch 17/100  
25/25 [==============================] - 4s 172ms/step - loss: 0.0178 - accuracy: 1.0000 - val\_loss: 1.1623 - val\_accuracy: 0.7050

***Question:*** It looks like our model is overfitting because its performing far better on the training data set than the validation data set, 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 TenosrBoard logs

The TensorBoard magic funtion (%tensorboard) wil access the logs directory we created earlier and visulaize its contents.

%tensorboard --logdir drive/MyDrive/Dog\ Vision/logs

<IPython.core.display.Javascript object>

## Making and Evaluating predictions using a trained model

val\_data

<\_BatchDataset element\_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))>

# Make predictons on the validation data (not used to train on)  
predictions = model.predict(val\_data, verbose = 1)  
predictions

7/7 [==============================] - 1s 92ms/step

array([[2.3323307e-03, 1.4351885e-04, 1.5960230e-03, ..., 1.1944692e-04,  
 4.0561030e-05, 2.0377403e-03],  
 [2.0938925e-03, 2.4629015e-04, 3.0662778e-03, ..., 3.6892082e-04,  
 1.6580694e-03, 2.3120658e-04],  
 [3.2294734e-05, 4.4552664e-05, 5.3766256e-05, ..., 9.4677343e-06,  
 1.1917074e-05, 6.0043199e-05],  
 ...,  
 [7.2961069e-07, 1.2847597e-06, 1.1792684e-05, ..., 1.7193157e-05,  
 8.7511844e-06, 9.7661532e-06],  
 [2.4502989e-02, 1.2660999e-04, 1.8827035e-04, ..., 8.1888735e-05,  
 1.3437713e-04, 1.3917766e-02],  
 [3.9295212e-04, 6.4595937e-05, 9.4654196e-04, ..., 2.5689621e-02,  
 1.8754305e-04, 9.0504304e-04]], dtype=float32)

np.sum(predictions[0])

1.0

# First prediction  
index = 30  
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])]}")

[1.17489639e-04 2.32985826e-06 8.60671644e-05 8.92593380e-05  
 8.67065537e-05 1.10463061e-05 5.62795624e-03 1.60039417e-05  
 4.77279391e-05 8.56178012e-05 2.30257181e-04 9.79411780e-05  
 9.77654054e-06 1.47002374e-05 2.45218198e-05 8.47105184e-05  
 2.11783972e-05 2.86900700e-04 7.20766911e-06 3.20501458e-05  
 3.53815703e-05 1.15682296e-05 1.16288456e-05 6.36021832e-06  
 1.65434703e-05 5.15171523e-05 5.46889729e-04 1.19535514e-04  
 2.88896317e-05 1.38716787e-04 2.06071763e-06 1.64883118e-03  
 1.37715935e-04 5.20444246e-06 2.27107485e-05 9.09594297e-01  
 1.48358845e-06 7.96120639e-06 1.09472121e-05 1.33702515e-05  
 2.01794799e-04 1.15982497e-04 3.96404630e-06 1.29797945e-05  
 4.15918066e-06 8.60564178e-05 2.30534670e-06 5.81987297e-06  
 1.24832138e-03 1.23645121e-04 7.50334730e-05 1.29352093e-05  
 2.05487249e-05 8.15945714e-06 1.25676415e-05 3.61186503e-05  
 2.36243159e-05 2.30755541e-04 2.00531213e-05 1.63665914e-04  
 2.49744644e-05 3.32510467e-06 9.14276607e-05 9.75847070e-06  
 8.32374935e-05 4.27151506e-04 1.81662926e-05 5.74385786e-06  
 2.49011064e-04 3.53488867e-05 9.91273904e-04 4.00378158e-05  
 9.00014413e-07 4.09503048e-03 6.05373898e-05 3.77406795e-05  
 8.46541545e-04 5.44108870e-03 3.38458849e-05 8.48331663e-04  
 1.58239782e-05 3.35658988e-04 1.22489489e-03 1.07728271e-03  
 1.41766315e-04 1.61232638e-05 1.59437695e-05 4.76318164e-05  
 3.83028892e-05 9.75313451e-05 1.82449112e-05 4.36196715e-05  
 1.72493055e-05 1.18234406e-04 2.30831233e-06 6.02201499e-06  
 5.39589892e-05 1.02316219e-04 8.51297053e-04 3.54779004e-05  
 1.14392918e-02 1.11991149e-05 2.81326063e-02 8.22207658e-05  
 1.01850477e-04 9.86050945e-05 4.83751035e-04 1.82343338e-05  
 1.99448565e-04 1.29625155e-02 7.53457600e-04 9.54416319e-05  
 2.59295884e-05 3.04237456e-05 3.11748678e-04 5.24085135e-06  
 3.04846937e-04 8.58395254e-07 1.37019197e-05 5.52595127e-03]  
Max value (probability of prediction): 0.9095942974090576  
Sum: 1.0  
Max index: 35  
Predicted label: dandie\_dinmont

unique\_breeds[35]

{"type":"string"}

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

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

**Note:** Prediction probabilites are aslo known as confidence levels.

# Turn Prediction probabilities into their respective lebel (easier to understand)  
def get\_pred\_label(prediction\_probabilities):  
 """  
 Turns an array of prediction probabilites 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[81])  
pred\_label

{"type":"string"}

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

val\_data

<\_BatchDataset element\_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))>

# Create a function to unbatch a batch dataset  
def unbatch\_fy(batched\_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 batched\_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 = unbatch\_fy(val\_data)  
val\_images[0], val\_labels[0]

(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')

get\_pred\_label(val\_labels[0])

{"type":"string"}

Now we've got ways to get:

* Prediction labels
* Validation labels (truth labels)
* Validation images

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

We'll create a function which:

* Takes an array of prediction probabilities, an array of truth labels and an array of images and an integer.✅
* Convert the prediction probabilities to a predicted label.✅
* Plot the predicted label, its predicted probability, the truth label and the target image on a sinle plot.✅

def plot\_pred(prediction\_probabilities, labels, images, n=1):  
 """  
 View the prediction, ground truth and image for sample n  
 """  
 pred\_prob, true\_label, image = prediction\_probabilities[n], labels[n], images[n]  
  
 # Get the pred label  
 pred\_label = get\_pred\_label(pred\_prob)  
  
  
 # Plot the image and remove ticks  
 plt.imshow(image)  
 plt.xticks([])  
 plt.yticks([])  
  
 # Change the colour of the title depending on if the prediction is right or wrong  
 if pred\_label == true\_label:  
 color = "green"  
 else:  
 color = "red"  
  
 # Change plot title to be predicted, probability of prediction and truth label  
 plt.title("{} {:2.0f}% {}".format(pred\_label,  
 np.max(pred\_prob)\*100,  
 true\_label), color = color)

plot\_pred(prediction\_probabilities=predictions,  
 labels = val\_labels,  
 images = val\_images,  
 n=9)



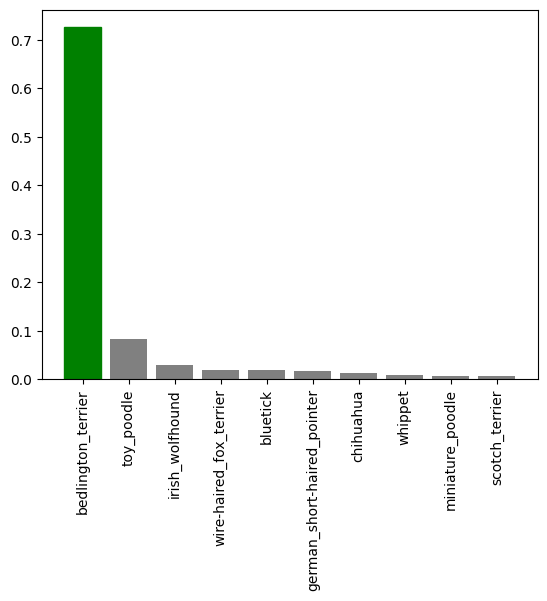
Now we've got one function to visualize our models top prediction, let's make another to view out models top 10 predictions.

This function will:

* Take an input of predictio probabilities array and a ground truth label and an integer✅
* Find the prediction using get\_pred\_label() ✅
* Find the top 10:
* Prediction probabilities indexes✅
* Preddiction probabilities values✅
* Prediction labels✅
* Plot the top 10 prediction probability values and labels, colouring the truth label green ✅

def plot\_pred\_conf(prediction\_probabilities, labels, n):  
 """  
 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 confidense 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 the colour of true label  
 if np.isin(true\_label, top\_10\_pred\_labels):  
 top\_plot[np.argmax(top\_10\_pred\_labels == true\_label)].set\_color("green")  
 else:  
 pass

plot\_pred\_conf(prediction\_probabilities=predictions,  
 labels = val\_labels,  
 n = 54)



predictions[0]

array([2.3323307e-03, 1.4351885e-04, 1.5960230e-03, 4.0613366e-05,  
 2.3384753e-04, 2.3148558e-04, 6.0024809e-02, 2.3006821e-04,  
 5.9366106e-05, 3.6384500e-04, 5.7703361e-04, 3.7427223e-04,  
 1.3209016e-04, 7.3472111e-05, 1.5877941e-04, 6.2758476e-04,  
 1.5374251e-05, 1.4747034e-01, 3.3000553e-05, 8.3741026e-05,  
 6.0366723e-04, 2.1531276e-04, 1.8979286e-05, 4.9911125e-04,  
 6.3064123e-05, 3.4766673e-04, 2.2310114e-01, 2.8664074e-04,  
 1.6805741e-04, 8.7790308e-05, 3.7806170e-05, 2.7161816e-03,  
 2.9265197e-04, 2.0542631e-05, 3.1127795e-05, 8.4102049e-02,  
 7.0941319e-06, 4.6019765e-04, 3.3680612e-05, 6.4502720e-04,  
 7.7460607e-04, 6.2255625e-05, 1.1809181e-05, 3.1099893e-04,  
 1.4921200e-05, 2.2060465e-04, 3.6638423e-05, 7.0271950e-04,  
 3.5426244e-03, 5.8735608e-05, 7.4974574e-05, 1.2435875e-05,  
 2.7787610e-04, 1.3711919e-04, 2.8433011e-05, 1.6332243e-04,  
 1.1918617e-04, 2.3871553e-03, 1.1346986e-03, 8.3381273e-02,  
 2.2722170e-04, 1.8610973e-04, 2.6625118e-03, 1.0773844e-05,  
 8.6408028e-05, 5.3324662e-02, 2.1609878e-04, 2.3045097e-04,  
 2.7462353e-03, 1.9390932e-04, 5.5497628e-02, 6.7007008e-05,  
 6.6790279e-05, 6.8217643e-02, 7.2129071e-04, 1.0701471e-04,  
 7.5801695e-03, 7.4285697e-03, 3.9126331e-04, 1.8932901e-02,  
 4.0166092e-04, 4.7111781e-03, 3.4499061e-04, 8.7231165e-03,  
 2.5154746e-05, 3.5880500e-04, 5.5861754e-05, 1.7730106e-04,  
 3.1088831e-04, 1.0470554e-03, 2.3311342e-03, 9.6631251e-05,  
 1.7999193e-05, 2.3616706e-03, 2.8225197e-05, 1.2311582e-04,  
 5.0393282e-04, 8.4084542e-03, 8.6292200e-04, 4.2991232e-05,  
 8.8998741e-03, 1.9576892e-04, 2.0473052e-02, 5.5034988e-02,  
 2.4103976e-04, 1.4045130e-04, 9.5950691e-03, 5.0142342e-05,  
 1.4705799e-04, 2.7856283e-02, 7.2379340e-04, 6.6087360e-04,  
 3.5796518e-05, 1.3662131e-04, 5.6514819e-04, 1.1478041e-05,  
 1.2152595e-03, 1.1944692e-04, 4.0561030e-05, 2.0377403e-03],  
 dtype=float32)

predictions[0].argsort()

array([ 36, 63, 115, 42, 51, 44, 16, 92, 22, 33, 84, 94, 54,  
 34, 18, 38, 112, 46, 30, 118, 3, 99, 107, 86, 49, 8,  
 41, 24, 72, 71, 13, 50, 19, 64, 29, 91, 75, 56, 117,  
 95, 12, 113, 53, 105, 1, 108, 14, 55, 28, 87, 61, 69,  
 101, 21, 66, 45, 60, 7, 67, 5, 4, 104, 52, 27, 32,  
 88, 43, 82, 25, 85, 9, 11, 78, 80, 37, 23, 96, 114,  
 10, 20, 15, 39, 111, 47, 74, 110, 40, 98, 89, 58, 116,  
 2, 119, 90, 0, 93, 57, 62, 31, 68, 48, 81, 77, 76,  
 97, 83, 100, 106, 79, 102, 109, 65, 103, 70, 6, 73, 59,  
 35, 17, 26])

predictions[0].argsort()[-10:]

array([109, 65, 103, 70, 6, 73, 59, 35, 17, 26])

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

array([0.22310114, 0.14747034, 0.08410205, 0.08338127, 0.06821764,  
 0.06002481, 0.05549763, 0.05503499, 0.05332466, 0.02785628],  
 dtype=float32)

predictions[0].max()

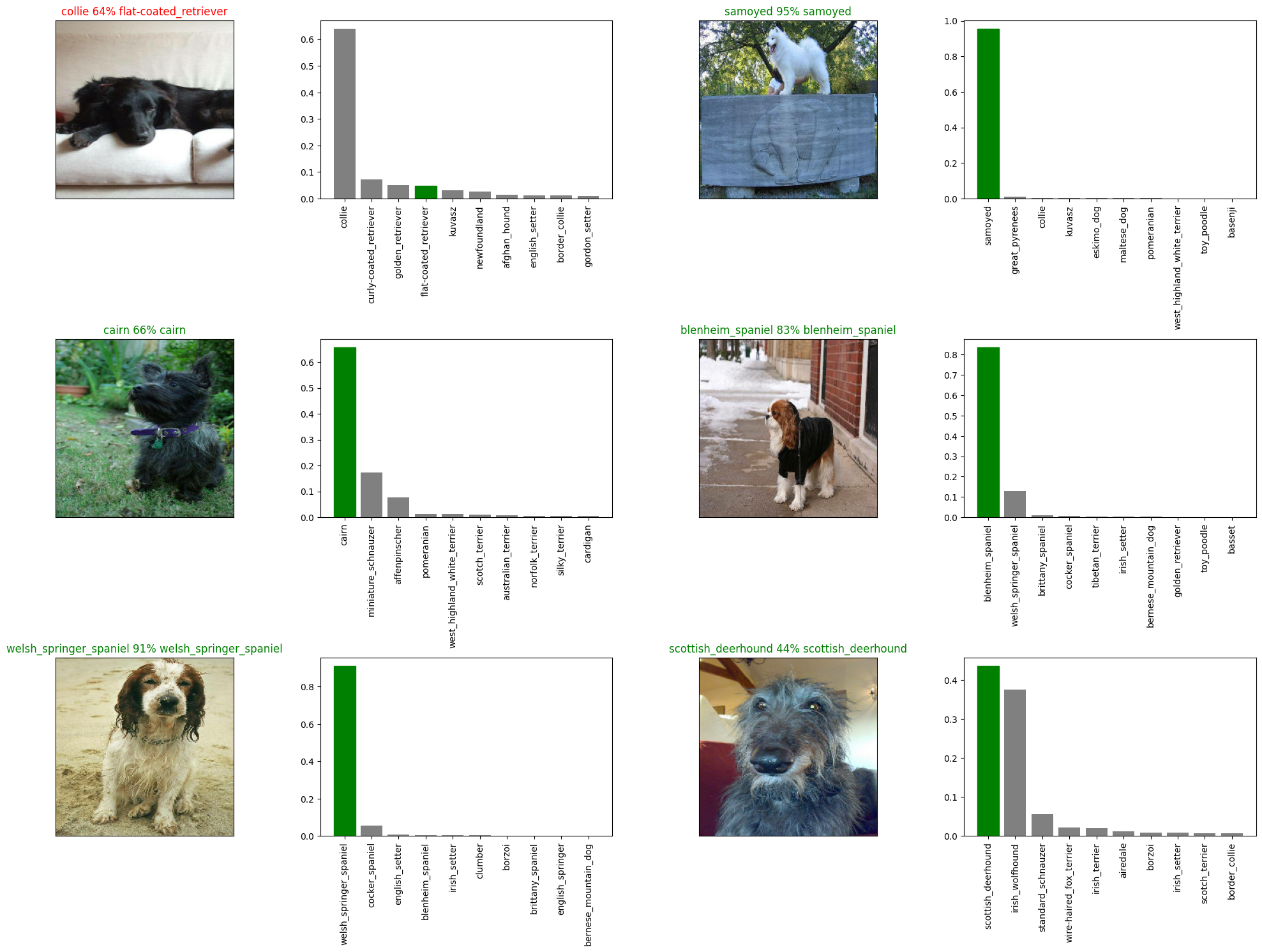
0.22310114

unique\_breeds[predictions[0].argsort()[-10:]]

array(['tibetan\_terrier', 'komondor', 'soft-coated\_wheaten\_terrier',  
 'lhasa', 'australian\_terrier', 'maltese\_dog', 'irish\_wolfhound',  
 'dandie\_dinmont', 'border\_terrier', 'cairn'], dtype=object)

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

# Lets check out a few predictions and their different values  
i\_multiplier = 20  
num\_rows = 3  
num\_cols = 2  
num\_images = num\_rows\*num\_cols  
plt.figure(figsize = (10\*num\_cols, 5\*num\_rows))  
for i in range(num\_images):  
 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()  
plt.show()



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

# import tensorflow\_addons as tfa  
# def confusion\_mat(prediction\_probabilities, labels, n):  
# pred\_prob, true\_label = prediction\_probabilities[n], labels[n]  
  
# metric = tfa.metrics.MultiLabelConfusionMatrix(num\_classes=3)  
# metric.update\_state(true\_label, pred\_prob)  
# result = metric.result()  
# return result.numpy()

## 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/MyDrive/Dog Vision/models",  
 datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))  
 model\_path = modeldir + "-" + suffix + ".h5" # Save model format  
  
 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 a specified path  
 """  
 print(f"Loading saved model from: {model\_path}...")  
 model = tf.keras.models.load\_model(model\_path,  
 custom\_objects = {"KerasLayer":hub.KerasLayer})  
 return model

Now we've got functions to save and load a trained model, let's make sure they work

# Save out model trained on 1000 images  
save\_model(model, suffix = "1000-images-mbilenetv2-Adam")

Saving model to: drive/MyDrive/Dog Vision/models/20230827-100417-1000-images-mbilenetv2-Adam.h5...

{"type":"string"}

# Load a trained model  
loaded\_1000\_image\_model = load\_model("drive/MyDrive/Dog Vision/models/20230826-124848-1000-images-mbilenetv2-Adam.h5")  
loaded\_1000\_image\_model

Loading saved model from: drive/MyDrive/Dog Vision/models/20230826-124848-1000-images-mbilenetv2-Adam.h5...

<keras.engine.sequential.Sequential at 0x79259807d4e0>

# Evaluate the pre-loaded model  
model.evaluate(val\_data)

7/7 [==============================] - 1s 84ms/step - loss: 1.1623 - accuracy: 0.7050

[1.1622645854949951, 0.7049999833106995]

# Evaluate the loaded model  
loaded\_1000\_image\_model.evaluate(val\_data)

7/7 [==============================] - 2s 91ms/step - loss: 1.3305 - accuracy: 0.6400

[1.330535888671875, 0.6399999856948853]

# Training a big dog model 🐶 (on the full data)

len(X), len(y)

(10222, 10222)

# Create a data batch with the full data set  
full\_data = create\_data\_batches(X, y)

Creating training data batches....

full\_data

<\_BatchDataset element\_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))>

# Create a model for full model  
full\_model = create\_model()

Building model with: https://tfhub.dev/google/imagenet/mobilenet\_v2\_130\_224/classification/5

# Create ful 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 up to 30 minutes for the first epoch) because the GPU we're using in the runtime has to load all 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 [==============================] - 40s 112ms/step - loss: 1.3279 - accuracy: 0.6716  
Epoch 2/100  
320/320 [==============================] - 34s 106ms/step - loss: 0.3993 - accuracy: 0.8842  
Epoch 3/100  
320/320 [==============================] - 36s 112ms/step - loss: 0.2392 - accuracy: 0.9348  
Epoch 4/100  
320/320 [==============================] - 34s 107ms/step - loss: 0.1524 - accuracy: 0.9623  
Epoch 5/100  
320/320 [==============================] - 35s 111ms/step - loss: 0.1061 - accuracy: 0.9795  
Epoch 6/100  
320/320 [==============================] - 35s 109ms/step - loss: 0.0778 - accuracy: 0.9861  
Epoch 7/100  
320/320 [==============================] - 35s 110ms/step - loss: 0.0606 - accuracy: 0.9905  
Epoch 8/100  
320/320 [==============================] - 45s 142ms/step - loss: 0.0460 - accuracy: 0.9943  
Epoch 9/100  
320/320 [==============================] - 40s 125ms/step - loss: 0.0376 - accuracy: 0.9960  
Epoch 10/100  
320/320 [==============================] - 34s 107ms/step - loss: 0.0314 - accuracy: 0.9967  
Epoch 11/100  
320/320 [==============================] - 34s 107ms/step - loss: 0.0252 - accuracy: 0.9979  
Epoch 12/100  
320/320 [==============================] - 38s 119ms/step - loss: 0.0230 - accuracy: 0.9980  
Epoch 13/100  
320/320 [==============================] - 38s 120ms/step - loss: 0.0199 - accuracy: 0.9986  
Epoch 14/100  
320/320 [==============================] - 35s 108ms/step - loss: 0.0172 - accuracy: 0.9983  
Epoch 15/100  
320/320 [==============================] - 34s 106ms/step - loss: 0.0155 - accuracy: 0.9988  
Epoch 16/100  
320/320 [==============================] - 36s 111ms/step - loss: 0.0158 - accuracy: 0.9983  
Epoch 17/100  
320/320 [==============================] - 36s 111ms/step - loss: 0.0137 - accuracy: 0.9987  
Epoch 18/100  
320/320 [==============================] - 34s 107ms/step - loss: 0.0133 - accuracy: 0.9987

<keras.callbacks.History at 0x792534474ca0>

save\_model(full\_model, suffix = "full-image-set-mobilenettv2-Adam")

Saving model to: drive/MyDrive/Dog Vision/models/20230827-103744-full-image-set-mobilenettv2-Adam.h5...

{"type":"string"}

# Loading full model  
loaded\_full\_model = load\_model("drive/MyDrive/Dog Vision/models/20230827-103744-full-image-set-mobilenettv2-Adam.h5")

Loading saved model from: drive/MyDrive/Dog Vision/models/20230827-103744-full-image-set-mobilenettv2-Adam.h5...

len(X)

10222

len(full\_data)

320

## Making predictions on test data set

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.

Lockily, we created create\_data\_battches() 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 calledd on out model.

# Load test image filenames  
test\_path = "drive/MyDrive/Dog Vision/test/"  
test\_filenames = [test\_path + fname for fname in os.listdir(test\_path)]  
len(test\_filenames), test\_filenames[:10]

(10357,  
 ['drive/MyDrive/Dog Vision/test/e6a0eeb3e82e14eb812d9ef22383ec11.jpg',  
 'drive/MyDrive/Dog Vision/test/e1e7949f64ff3283a5dafa9d39199193.jpg',  
 'drive/MyDrive/Dog Vision/test/e0e47873420b2ed420e920a74c663233.jpg',  
 'drive/MyDrive/Dog Vision/test/dfe25675d32a75ade0f2893cf2ab63a9.jpg',  
 'drive/MyDrive/Dog Vision/test/e35b90290702042d17ceee2aaf2d1475.jpg',  
 'drive/MyDrive/Dog Vision/test/e4d878f113883a3caab47d3126d2c2a4.jpg',  
 'drive/MyDrive/Dog Vision/test/e01a1ea716359e2f0062aca591ff9ca4.jpg',  
 'drive/MyDrive/Dog Vision/test/e6ec1891d8f52a13683c73cabd3045cb.jpg',  
 'drive/MyDrive/Dog Vision/test/dd703c7beeaf5cba5533d5f42b608f2e.jpg',  
 'drive/MyDrive/Dog Vision/test/e487437a727b3e4f0b919c878d39ae6e.jpg'])

# Create test data batches  
test\_data = create\_data\_batches(test\_filenames, test\_data = True)

Creating test data batches....

len(test\_data)

324

test\_data

<\_BatchDataset element\_spec=TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None)>

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

# Make predictions on test data batch using the loaded full model  
test\_predictions = loaded\_full\_model.predict(test\_data,  
 verbose = 1)

324/324 [==============================] - 39s 121ms/step

# Save predictions (Numpy array) to csv file( for access later)  
np.savetxt("drive/MyDrive/Dog Vision/preds\_array.csv", test\_predictions, delimiter=",")

# Load predictions (Numpy array) from csv file  
test\_predictions = np.loadtxt("drive/MyDrive/Dog Vision/preds\_array.csv", delimiter=",")

test\_predictions[:10]

array([[2.63442566e-08, 2.46197818e-09, 1.86607796e-09, ...,  
 2.68804912e-09, 3.44193040e-06, 8.37554879e-08],  
 [2.00158952e-11, 5.48215411e-08, 2.02552072e-10, ...,  
 1.44969215e-12, 8.08851819e-10, 1.16470222e-09],  
 [8.69825079e-15, 6.99958078e-11, 3.53193156e-14, ...,  
 1.20963810e-14, 5.90082835e-19, 1.93761023e-14],  
 ...,  
 [1.22964366e-06, 2.04135486e-10, 2.61234856e-09, ...,  
 5.26257757e-08, 9.26978173e-05, 2.28186803e-08],  
 [4.37165970e-10, 2.28618873e-07, 1.06506093e-09, ...,  
 8.44036840e-09, 1.25215626e-12, 1.13493437e-10],  
 [9.99979615e-01, 2.00875232e-12, 1.18881294e-11, ...,  
 3.45561660e-14, 2.37740053e-07, 3.17659149e-10]])

test\_predictions.shape

(10357, 120)

## Preparing test dataset predictions for Kaggle

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

www.kaggle.com/competitions/dog-breed-identification/overview/evaluation

To get the data in this format, we'll:

* Create a pandas DataFrame with an ID column as well as a column for each dog breed.✅
* Add data to the ID column by extracting the test image ID's from their filepaths.✅
* Add data (the prediction probabilities) to each of the dog breed columns.✅
* Export the DataFrame as a CSV t submit it to Kaggle.✅

# Create a pandas DataFrame with emmpty columns  
preds\_df = pd.DataFrame(columns=["id"] + list(unique\_breeds))

preds\_df

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]

# Append test image ID's to Predictions DataFrame  
test\_ids = [os.path.splitext(path)[0] for path in os.listdir(test\_path)]  
preds\_df["id"] = test\_ids

preds\_df.head()

id affenpinscher afghan\_hound \  
0 e6a0eeb3e82e14eb812d9ef22383ec11 NaN NaN   
1 e1e7949f64ff3283a5dafa9d39199193 NaN NaN   
2 e0e47873420b2ed420e920a74c663233 NaN NaN   
3 dfe25675d32a75ade0f2893cf2ab63a9 NaN NaN   
4 e35b90290702042d17ceee2aaf2d1475 NaN NaN   
  
 african\_hunting\_dog airedale american\_staffordshire\_terrier appenzeller \  
0 NaN NaN NaN NaN   
1 NaN NaN NaN NaN   
2 NaN NaN NaN NaN   
3 NaN NaN NaN NaN   
4 NaN NaN NaN NaN   
  
 australian\_terrier basenji basset ... toy\_poodle toy\_terrier vizsla \  
0 NaN NaN NaN ... NaN NaN NaN   
1 NaN NaN NaN ... NaN NaN NaN   
2 NaN NaN NaN ... NaN NaN NaN   
3 NaN NaN NaN ... NaN NaN NaN   
4 NaN NaN NaN ... NaN NaN NaN   
  
 walker\_hound weimaraner welsh\_springer\_spaniel west\_highland\_white\_terrier \  
0 NaN NaN NaN NaN   
1 NaN NaN NaN NaN   
2 NaN NaN NaN NaN   
3 NaN NaN NaN NaN   
4 NaN NaN NaN NaN   
  
 whippet wire-haired\_fox\_terrier yorkshire\_terrier   
0 NaN NaN NaN   
1 NaN NaN NaN   
2 NaN NaN NaN   
3 NaN NaN NaN   
4 NaN NaN NaN   
  
[5 rows x 121 columns]

# Add the prediction probabilites to each dog breed column  
preds\_df[list(unique\_breeds)] = test\_predictions  
preds\_df.head()

id affenpinscher afghan\_hound \  
0 e6a0eeb3e82e14eb812d9ef22383ec11 2.634426e-08 2.461978e-09   
1 e1e7949f64ff3283a5dafa9d39199193 2.001590e-11 5.482154e-08   
2 e0e47873420b2ed420e920a74c663233 8.698251e-15 6.999581e-11   
3 dfe25675d32a75ade0f2893cf2ab63a9 3.356070e-10 3.917449e-08   
4 e35b90290702042d17ceee2aaf2d1475 7.876174e-08 7.332662e-10   
  
 african\_hunting\_dog airedale american\_staffordshire\_terrier \  
0 1.866078e-09 1.896031e-10 5.028189e-08   
1 2.025521e-10 8.503612e-09 3.735856e-10   
2 3.531932e-14 5.836592e-14 9.967702e-12   
3 5.164413e-10 7.907516e-09 7.924814e-10   
4 1.554966e-07 4.029445e-08 4.276007e-08   
  
 appenzeller australian\_terrier basenji basset ... \  
0 9.580733e-10 1.262377e-07 2.218409e-08 5.716652e-10 ...   
1 4.871441e-11 2.213511e-09 2.424377e-08 5.259480e-12 ...   
2 3.127714e-15 1.585185e-14 2.124403e-14 1.689211e-14 ...   
3 1.930193e-11 6.528655e-10 3.234529e-09 8.735718e-07 ...   
4 1.332959e-11 1.477515e-05 6.875673e-08 3.503628e-08 ...   
  
 toy\_poodle toy\_terrier vizsla walker\_hound weimaraner \  
0 2.756775e-01 3.394872e-06 4.747842e-07 8.603870e-10 6.685388e-08   
1 8.289662e-01 4.575340e-09 1.065078e-06 7.299110e-13 3.190729e-09   
2 5.326429e-16 1.219094e-12 1.231160e-13 8.908330e-11 1.400277e-11   
3 1.379677e-09 1.800535e-11 2.308519e-10 1.648764e-10 8.406763e-11   
4 4.080529e-08 7.897322e-07 1.570365e-09 8.938938e-10 5.347433e-11   
  
 welsh\_springer\_spaniel west\_highland\_white\_terrier whippet \  
0 1.966724e-09 1.696117e-06 2.688049e-09   
1 1.970723e-11 4.738494e-12 1.449692e-12   
2 3.405166e-15 8.273758e-14 1.209638e-14   
3 7.297006e-10 4.816208e-09 3.159518e-08   
4 7.220291e-09 2.726640e-05 6.853620e-09   
  
 wire-haired\_fox\_terrier yorkshire\_terrier   
0 3.441930e-06 8.375549e-08   
1 8.088518e-10 1.164702e-09   
2 5.900828e-19 1.937610e-14   
3 1.364760e-06 1.292372e-09   
4 9.607684e-08 1.584055e-06   
  
[5 rows x 121 columns]

preds\_df["affenpinscher"]

0 2.634426e-08  
1 2.001590e-11  
2 8.698251e-15  
3 3.356070e-10  
4 7.876174e-08  
 ...   
10352 1.908874e-09  
10353 1.946528e-07  
10354 2.440172e-09  
10355 1.891247e-06  
10356 3.731553e-04  
Name: affenpinscher, Length: 10357, dtype: float64

# Save out predictions dataframe to CSV for submission to Kaggle  
preds\_df.to\_csv("drive/MyDrive/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 into data batches using create\_data\_batches() and since our custom images won't have labels, we set the test\_data parameter to True.✅
* Add the custom image data batch to our models predict() method.✅
* Convert the prediction output probabilities to prediction labels.✅
* Compare the predicted labels to the custom images.✅

# Get custom image filepaths  
custom\_path = "drive/MyDrive/Dog Vision/dog-photos/"  
custom\_image\_paths = [custom\_path + fname for fname in os.listdir(custom\_path)]  
custom\_image\_paths

['drive/MyDrive/Dog Vision/dog-photos/images.jpg',  
 'drive/MyDrive/Dog Vision/dog-photos/download.jpg',  
 'drive/MyDrive/Dog Vision/dog-photos/download1.jpg']

# Turn custom images into batch data set  
custom\_data = create\_data\_batches(custom\_image\_paths, test\_data = True)  
custom\_data

Creating test data batches....

<\_BatchDataset element\_spec=TensorSpec(shape=(None, 224, 224, 3), dtype=tf.float32, name=None)>

# Make predictions on custom data  
custom\_preds = loaded\_full\_model.predict(custom\_data)

1/1 [==============================] - 1s 985ms/step

custom\_preds.shape

(3, 120)

# Get custom image prediction labels  
custom\_preds\_labels = [get\_pred\_label(custom\_preds[i]) for i in range(len(custom\_preds))]  
custom\_preds\_labels

['irish\_wolfhound', 'scottish\_deerhound', 'irish\_wolfhound']

# Get custom images (our unbatchify() function won't work since there aren't labels...may be we could fix this later)  
custom\_images = []  
# Lop through unbatched data  
for image in custom\_data.unbatch().as\_numpy\_iterator():  
 custom\_images.append(image)

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

