Open in Colab

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

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

New Section

1. Problem

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

2. Data

The data used is from Kaggle's dog bread identification competition: https://www.kaggle.com/c/dog-breed-identification/overview

3. Evaluation

Multi-Class Log Loss: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.log_loss.html

4. Features

Info about data:

- Dealing with images (unstructured data) so is probably best to use deep learning/transfer learning;
- There are 120 breeds of dogs (this means there are 120 different classes);
- There are ~ 10,000 + images in the training set(these images have labels) and ~ 10,000 images in the test set(no labels);

Get my space ready

- Import TF 2.X ✅
- Import TF Hub ✅

- Make sure I am using a GPU ✅

```
# Import necessary tools
from google.colab import drive

drive.mount("/content/drive", force_remount=True)

# Import TensorFlow and TensorFlow Hub into Colab
import tensorflow as tf
import tensorflow_hub as hub
print("TF version:", tf.__version__)
print("TF Hub version:", hub.__version__)

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

```
Mounted at /content/drive
TF version: 2.7.0
TF Hub version: 0.12.0
GPU not available!
```

Acces the data and getting it ready (turning into tensors)

- With all ML lerning models, data has to be in numerical format.

```
# Acces the data and checking the labels
import pandas as pd
labels_csv = pd.read_csv("/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identif
print(labels_csv.describe())
print(labels_csv.head())
```

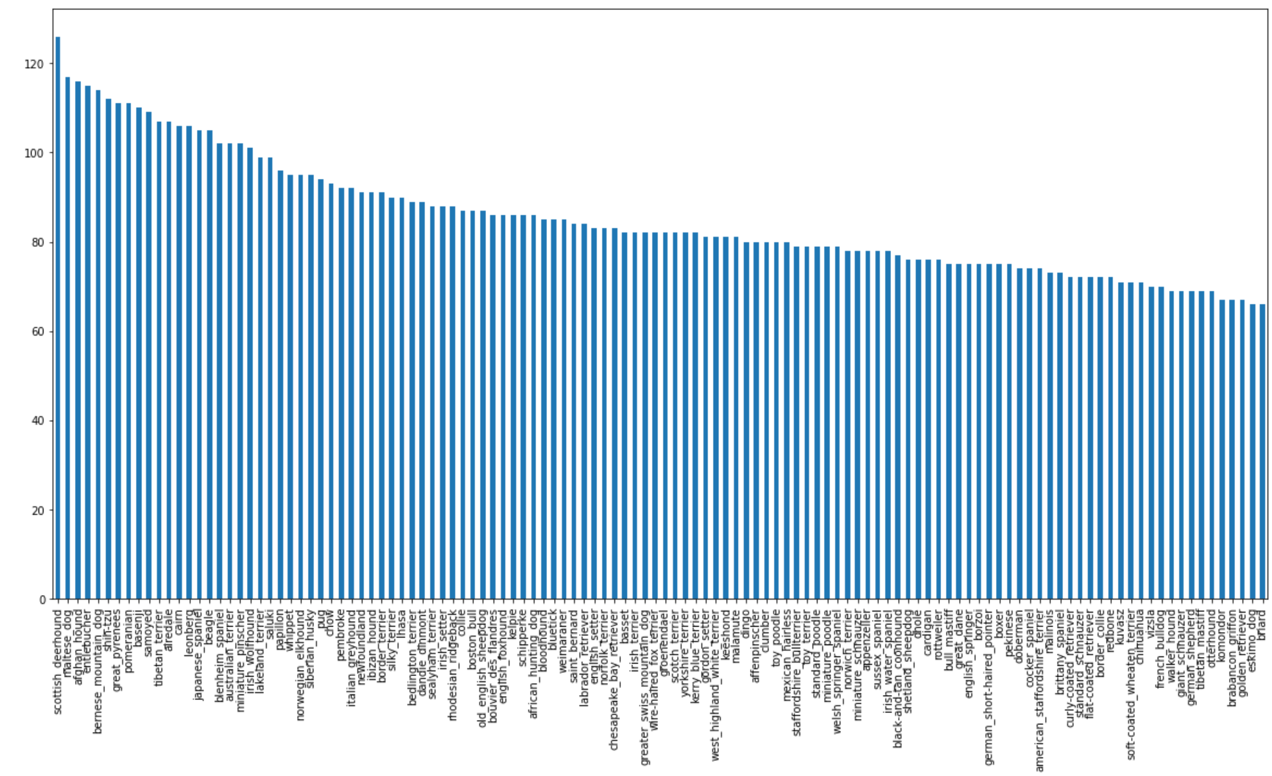
```
                                      id               breed
count                              10222               10222
unique                             10222                 120
top     c4bf9248192b875822e30f5e2a240c19  scottish_deerhound
freq                                   1                 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

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

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f912f860f10>
```




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

```
82.0
```

Double-click (or enter) to edit

```
# View an image
from IPython.display import Image
Image("/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00151
```

Getting images and their labels

Get a list of all of the images file pathnames.

```
# Create pathnames from image ID's
filenames = ["/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/trai

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

```
['/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/000be
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00151
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/001cc
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00214
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/0021f
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00221
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00290
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/002a2
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/003df
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00421
```

```
# Check if number of filenames matches the number of actual image files
import os
if len(os.listdir("/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification
  print("They are matching! Proceed!")
else:
  print("They don't match! Check the target directory.")
```

```
They are matching! Proceed!
```

```
# Check
Image(filenames[9000])
```


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

```
'tibetan_mastiff'
```

Prepare the labels

```
import numpy as np
labels = labels_csv["breed"] # or you can directly use `labels = labels_csv["breed"].to_numpy
labels = np.array(labels)
labels
```

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

```
len(labels)
```

```
10222
```

```
# Check if number of labels matches the number of filenames (to figure out if I have missing
if len(labels) == len(filenames):
  print("Number of labels matches number of filenames!")
else:
  print("No match! Check data directories.")
```

```
Number of labels matches number of filenames!
```

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

```
from numpy.ma.extras import unique
# 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])]
```

```
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 bolean 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]
```

```
filenames[:10]
```

```
['/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/000be
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00151
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/001cd
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00214
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/0021f
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00221
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00290
```

```
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/002a2
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/003d1
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00421
```

Creating my own validation set

Since the dataset from Kaggle doesn't include a validation set, can create one to use.

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

```
len(filenames)
```

```
10222
```

Start off experimenting with ~ 1000 images and increase it as need it

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

NUM_IMAGES: 1000

```
# Split the data into train and validation sets
from sklearn.model_selection import train_test_split

# Split 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(y_train), len(X_val), len(y_val)
```

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

```
# Quick look at the training data
X_train[:2], y_train[:2]
```

```
(['/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/00be
  '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/train/0d2f
 [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 the images into Tensors gonna write a function which does a few things:

1. Take an image filepath as input;
2. Use TensorFlow to read the file and save it to a variable `image`;
3. Turn the `image` (jpg) into Tensors and normalize the data;
4. Resize the `image` to be a shape of (224, 224);
5. Return the modified `image`.

```
# Importing example
# Convert image to a NumPy
from matplotlib.pyplot import imread
image = imread(filenames[41])
image.shape
tf.constant(image) # turn image into tensor
```

```
<tf.Tensor: shape=(375, 500, 3), dtype=uint8, numpy=
array([[[ 63,  52,  24],
        [ 62,  51,  23],
        [ 60,  48,  22],
        ...,
        [ 60,  41,  26],
        [ 62,  43,  28],
        [ 63,  44,  29]],

       [[ 64,  53,  25],
```

```
        [ 63,  52,  24],
        [ 61,  49,  23],
        ...,
        [ 61,  42,  27],
        [ 62,  43,  28],
        [ 64,  45,  30]],

       [[ 65,  54,  26],
        [ 64,  53,  25],
        [ 62,  50,  24],
        ...,
        [ 62,  43,  28],
        [ 63,  44,  29],
        [ 64,  45,  30]],

       ...,

       [[ 13,   6,   0],
        [ 18,  11,   1],
        [ 26,  17,  10],
        ...,
        [193, 166,  79],
        [197, 170,  83],
        [199, 172,  85]],

       [[ 21,  14,   4],
        [ 21,  14,   4],
        [ 22,  13,   6],
        ...,
        [196, 168,  84],
        [201, 173,  89],
        [204, 176,  92]],

       [[ 25,  18,   8],
        [ 20,  13,   3],
        [ 18,   9,   2],
        ...,
        [201, 172,  92],
        [208, 179,  99],
        [211, 182, 102]]], dtype=uint8)>
```

Make a function to preprocess images

```
# Define image size
IMG_SIZE = 224

# Create a function for preprocessing images
def process_image(image_path, img_size=IMG_SIZE):
  """
  Takes an image file path and turns it into a tensor
  """
```

```
  # Read in an image file
  image = tf.io.read_file(image_path)
  # Turn the 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 from 0-255 to 0-1 values(normalization)
  image = tf.cast(image, tf.float32) / 255.0
  # Resize the image to the desired value (244, 244)
  image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])

  return image
```

Turning Data into batches (32 is the recommended value)

Why turn it into batches?

If trying to process 10,000+ images in one go, they might not fit into the memory. That's why is recommended to use batches and the value recommended is 32 but it can be adjusted manually. In order to use TF effectively, need the data in the form of Tensor tuples likethis: (image, label)

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

```
# Example for the above function
(process_image(X[42]), tf.constant(y[42]))
```

```
(<tf.Tensor: shape=(224, 224, 3), dtype=float32, numpy=
 array([[[0.3264178 , 0.5222886 , 0.3232816 ],
         [0.25371668, 0.4436649 , 0.24117757],
         [0.25699762, 0.44670868, 0.2389375 ],
         ...,
         [0.29325107, 0.5189916 , 0.3215547 ],
         [0.29721776, 0.52466875, 0.33030328],
         [0.2948505 , 0.5223015 , 0.33406618]],

        [[0.25903144, 0.45378068, 0.27294815],
         [0.24375685, 0.44070187, 0.2554778 ],
         [0.2838985 , 0.4721338 , 0.28298813],
         ...,
         [0.2785345 , 0.5027992 , 0.31004712],
         [0.28428748, 0.5108719 , 0.32523635],
         [0.28821915, 0.5148036 , 0.32916805]],
```

```
        [[0.20941193, 0.4069295 , 0.25792548],
         [0.24045376, 0.43900943, 0.2868911 ],
         [0.29001117, 0.47937486, 0.32247734],
         ...,
         [0.26074055, 0.4841477 , 0.30125174],
         [0.27101526, 0.49454466, 0.32096273],
         [0.27939945, 0.5029289 , 0.32934693]],

        ...,

        [[0.00634795, 0.03442048, 0.0258106 ],
         [0.01408936, 0.04459916, 0.0301715 ],
         [0.01385711, 0.04856448, 0.02839671],
         ...,
         [0.42205158, 0.39761975, 0.21622121],
         [0.479325  , 0.4537054 , 0.2696505 ],
         [0.4818102 , 0.4582808 , 0.27004552]],

        [[0.00222061, 0.02262166, 0.03176915],
         [0.01008397, 0.03669046, 0.02473482],
         [0.00608852, 0.03890046, 0.01207283],
         ...,
         [0.36070833, 0.33803678, 0.16216144],
         [0.42499563, 0.39768007, 0.2170171 ],
         [0.44054326, 0.41395888, 0.23183355]],

        [[0.05608025, 0.06760228, 0.10401428],
         [0.05441073, 0.07435255, 0.05428263],
         [0.04734281, 0.07581793, 0.02060942],
         ...,
         [0.3397559 , 0.31265694, 0.14725602],
         [0.38772497, 0.360274  , 0.18714727],
         [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,
```

Found a way to turn data into tuples of Tensors in the form of: `(image, label)`. Next, make a function to turn all the data (`X` and `y`) into batches.

```
# Define the batch 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.
```

```
  Suffles the data if it's training data but doesn't shuffle if it's a validation data.
  Also accepts test data as input (no labels)
  """
  # If data is a test dataset, 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 data is a valid dataset, don't need to shuffle it
  elif valid_data:
    print("Creating validation data batches...")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(X), #filepaths
                                               tf.constant(y))) # labels
    data_batch = data.map(get_image_label).batch(BATCH_SIZE)
    return data_batch

  else:
    print("Creating training data batches...")
    # turn file paths 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 s
    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...
```

```
# Create out the different attributes of our data batches
train_data.element_spec, val_data.element_spec
```

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

Vizualizing Data Batches

Data is now in batches, these can be a little hard to understand/comprehend, so it's helpful to visualize it.

```
import matplotlib.pyplot as plt

# Create a function for viewing images in a data batch
def show_25_images(images, labels):
  """
  Diplays a plot of 25 images and their labels from a data batch.
  """
  # Setup the figure
  plt.figure(figsize=(10, 10))
  # loop through 25 ( for diplaying 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 title
    plt.title(unique_breeds[labels[i].argmax()])
    # Turn the grid lines off
    plt.axis("off")
```

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

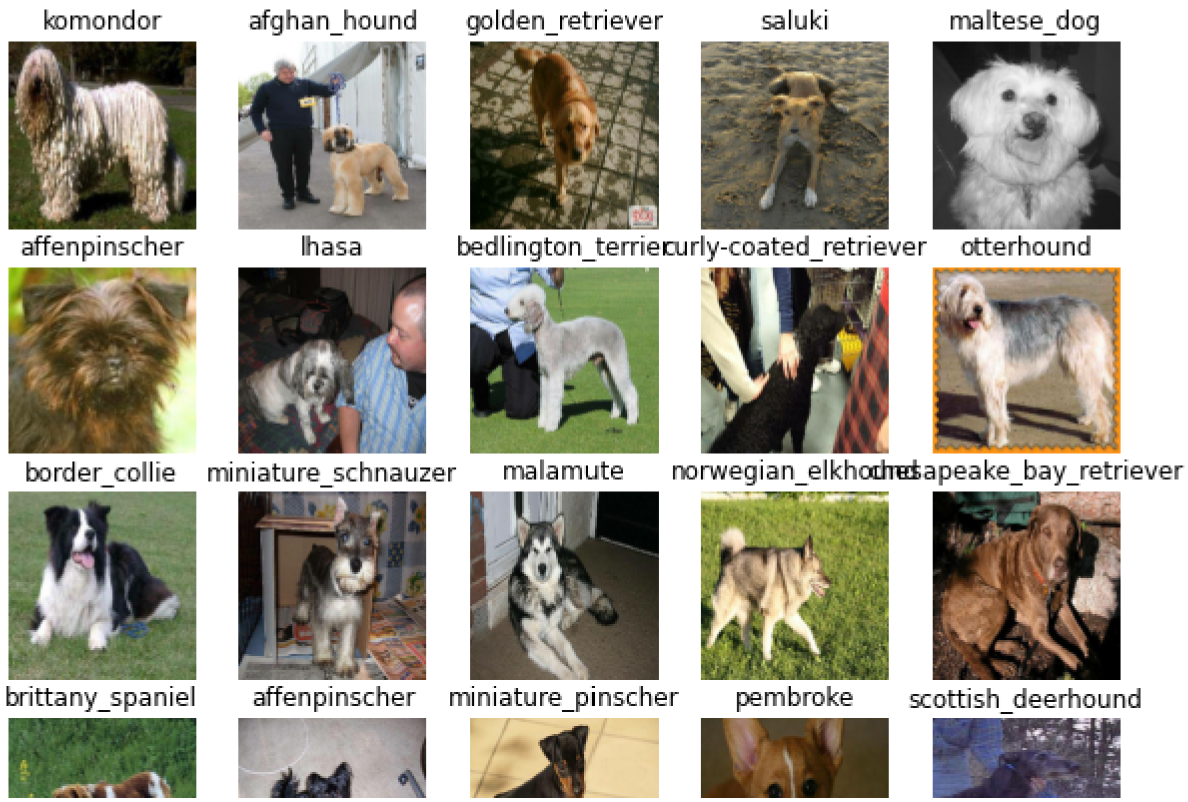
```
(array([[[[2.29164332e-01, 2.01713338e-01, 9.97525603e-02],
          [2.76823640e-01, 2.39840075e-01, 1.41056806e-01],
          [2.59717405e-01, 2.13630229e-01, 1.18339613e-01],
          ...,
          [1.93777740e-01, 1.58483624e-01, 2.90718619e-02],
          [1.95167005e-01, 1.59872890e-01, 4.61473875e-02],
          [1.53738230e-01, 1.15581803e-01, 1.29733179e-02]],

         [[2.39417613e-01, 2.11283848e-01, 1.11371383e-01],
          [2.26948544e-01, 1.89282238e-01, 9.25472900e-02],
          [2.39487126e-01, 1.92956448e-01, 9.94748697e-02],
          ...,
          [1.83715284e-01, 1.47055626e-01, 1.83266215e-02],
          [1.88757062e-01, 1.52097389e-01, 3.90546694e-02],
          [1.50638670e-01, 1.11116692e-01, 9.77877434e-03]],

         [[1.98112205e-01, 1.66739658e-01, 7.65435845e-02],
          [2.14145795e-01, 1.73240677e-01, 8.62221122e-02],
          [2.45732993e-01, 1.97098523e-01, 1.12198323e-01],
          ...,
          [1.73253998e-01, 1.30116746e-01, 6.90245815e-03],
          [1.84903130e-01, 1.41765878e-01, 3.19619514e-02],
```

```
        [1.32282361e-01, 8.62827972e-02, 7.41481408e-03]],

       ...,

       [[2.13962853e-01, 2.45335400e-01, 5.59446663e-02],
        [1.15368500e-01, 1.46741062e-01, 3.16885626e-03],
        [1.69562861e-01, 2.00935394e-01, 2.29824614e-02],
        ...,
        [2.29639724e-01, 2.64933854e-01, 9.63063836e-02],
        [3.02567571e-01, 3.37861687e-01, 1.69234246e-01],
        [1.98423579e-01, 2.33717695e-01, 6.50902390e-02]],

       [[2.41788641e-01, 2.73161203e-01, 8.10043365e-02],
        [1.88795984e-01, 2.20168531e-01, 3.74163166e-02],
        [1.63141862e-01, 1.94514409e-01, 7.94520788e-03],
        ...,
        [2.35361025e-01, 2.77132779e-01, 1.08505316e-01],
        [2.66598552e-01, 3.08370292e-01, 1.39742836e-01],
        [2.27994323e-01, 2.69766062e-01, 1.01138607e-01]],

       [[2.45456934e-01, 2.76829481e-01, 8.46726224e-02],
        [2.05147073e-01, 2.36519620e-01, 4.43627499e-02],
        [1.67857140e-01, 1.99229687e-01, 9.62009281e-03],
        ...,
        [2.30364874e-01, 2.73502141e-01, 1.04874678e-01],
        [2.61055648e-01, 3.04192901e-01, 1.35565430e-01],
        [2.47191474e-01, 2.90328741e-01, 1.21701270e-01]]],


      [[[1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
        [1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
        [1.00000000e+00, 1.00000000e+00, 1.00000000e+00],
        ...,
        [8.42201114e-01, 8.61808956e-01, 8.73573661e-01],
        [8.25348258e-01, 8.44956100e-01, 8.56720805e-01],
        [8.71499956e-01, 8.91107798e-01, 9.02872503e-01]],
```

```
len(train_images), len(train_labels)
```

```
(32, 32)
```

```
# Visualize data in a training batch
show_25_images(train_images, train_labels)
```




```
# Visualize the validation set
val_images, val_labels = next(val_data.as_numpy_iterator())
show_25_images(val_images, val_labels)
```

Building a model

Before bulding the model, there are a few things to define:

1. The input shape(images shape, in the form of Tensors) to the model;
2. The output shape(images labels, in the form of Tensors) to the model;
3. The URL of the model to be used(use transfer learning, to safe time and money) from TensorFlow hub: https://tfhub.dev

```
# 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)

# Setup model URL from TensorFlow Hub
MODEL_URL = "https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/5"
```

Inputs, outputs and model are ready to go. Put them together into a Keras Deep Learning Model.

Need to 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, than that etc.);
- Builds the model (tells the model the input shape it will be getting);
- Returns the model.

```
# 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
  ])

  # 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/classificat
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 keras_layer (KerasLayer)    (None, 1001)              5432713

 dense (Dense)               (None, 120)               120240

=================================================================
Total params: 5,552,953
Trainable params: 120,240
Non-trainable params: 5,432,713
_________________________________________________________________
```

Creating callbacks

Helper functions a model use during training to do such things as save it's progress, check its progress or stop training early if a model stops improving.

1. One callback for TensorBoard which helps track the models progress.
2. Another callback that prevents the model from training for too long.

TensorBoard Callback

To setup a TensorBoard callback need to do 3 things:

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

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

```
import datetime

# Create a function to build a TensorBoard callback
def create_tensorboard_callback():
  # Create a log directory for storing TensorBoard logs
  logdir = os.path.join("/content/drive/MyDrive/Colab Notebooks/Dog Vision/logs",
                        # Make it so the logs get tracked whenever 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 the model from overfitting by stopping training if a certain evaluation metric stops improving.

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

Training a model (on subset of data)

Train the model on 1000 images to make sure everything is working.

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

NUM_EPOCHS: 100

```
# Check to make sure still running on GPU
print("GPU", "available (YESSSSSSSS!)" if tf.config.list_physical_devices("GPU") else "not av
```

```
GPU not available
```

Create a function which trains the model.

- Create a model using `create_model()`
- Setup a TensorBoard callback using `create_tensorboard_callback()`
- Call the `fit()` function on the model passing it the training data, validation data, number of epochs to trainbf for (`NUM_EPOCHS`);

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

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

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

  # return the model
  return model
```

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

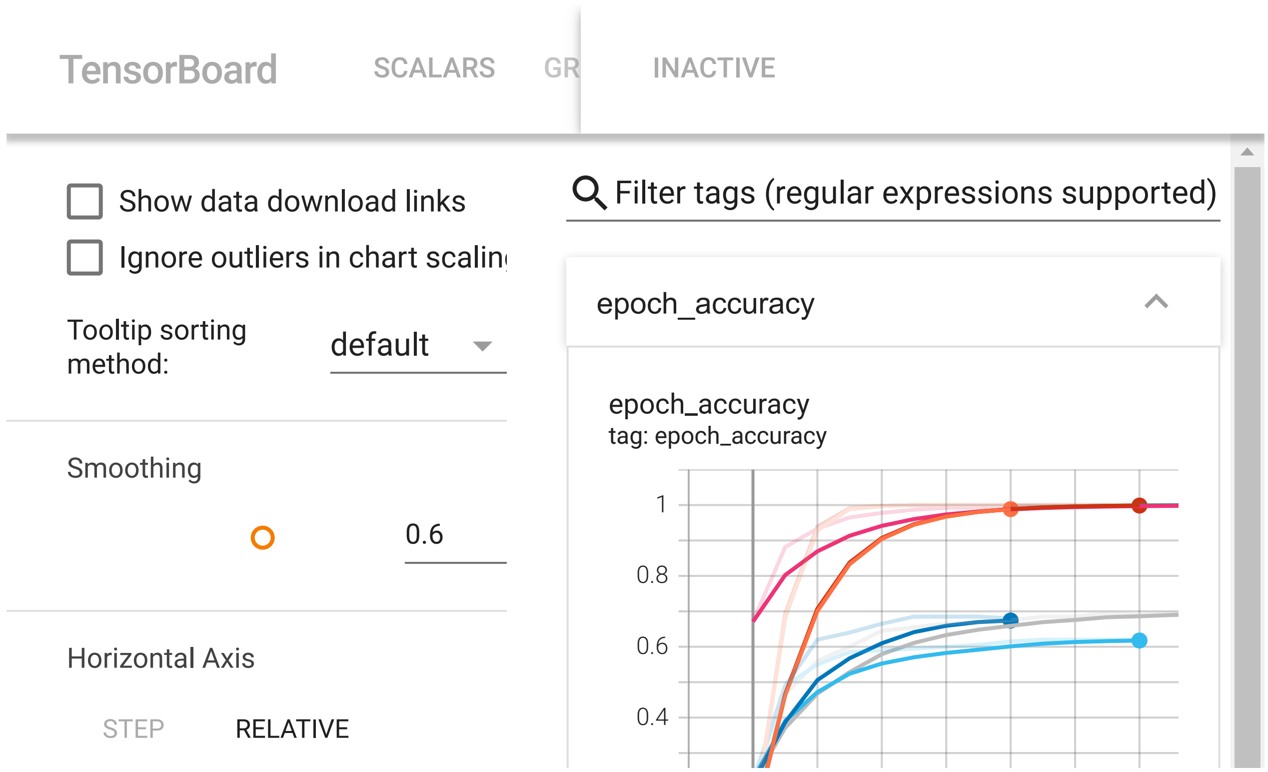
```
Building model with: https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classificat
Epoch 1/100
25/25 [==============================] - 206s 8s/step - loss: 4.5385 - accuracy: 0.1013
Epoch 2/100
25/25 [==============================] - 42s 2s/step - loss: 1.6084 - accuracy: 0.6812 -
Epoch 3/100
25/25 [==============================] - 41s 2s/step - loss: 0.5415 - accuracy: 0.9388 -
Epoch 4/100
25/25 [==============================] - 41s 2s/step - loss: 0.2422 - accuracy: 0.9887 -
Epoch 5/100
25/25 [==============================] - 42s 2s/step - loss: 0.1411 - accuracy: 0.9975 -
Epoch 6/100
25/25 [==============================] - 57s 2s/step - loss: 0.0973 - accuracy: 1.0000 -
Epoch 7/100
25/25 [==============================] - 45s 2s/step - loss: 0.0738 - accuracy: 1.0000 -
Epoch 8/100
```

```
25/25 [==============================] - 45s 2s/step - loss: 0.0585 - accuracy: 1.0000 -
Epoch 9/100
25/25 [==============================] - 77s 3s/step - loss: 0.0484 - accuracy: 1.0000 -
Epoch 10/100
25/25 [==============================] - 56s 2s/step - loss: 0.0408 - accuracy: 1.0000 -
Epoch 11/100
25/25 [==============================] - 75s 3s/step - loss: 0.0350 - accuracy: 1.0000 -
Epoch 12/100
25/25 [==============================] - 69s 3s/step - loss: 0.0305 - accuracy: 1.0000 -
Epoch 13/100
25/25 [==============================] - 63s 3s/step - loss: 0.0269 - accuracy: 1.0000 -
Epoch 14/100
25/25 [==============================] - 49s 2s/step - loss: 0.0240 - accuracy: 1.0000 -
Epoch 15/100
25/25 [==============================] - 62s 2s/step - loss: 0.0216 - accuracy: 1.0000 -
```

Checking the TensorBoard logs

The TensorBoard magic function `%tensorboard` will acces the logs directory created earlier and visualize the content.

```
%reload_ext tensorboard
%tensorboard --logdir drive/MyDrive/Colab\ Notebooks/Dog\ Vision/logs
```



Making and evaluating predictions using a trained model

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

```
7/7 [==============================] - 10s 1s/step
array([[9.7217259e-04, 8.7541477e-05, 9.6009899e-05, ..., 1.9748615e-04,
        9.6973772e-06, 2.0239521e-03],
       [3.5270315e-03, 1.1797610e-03, 5.2355427e-02, ..., 6.6433812e-04,
        3.2545510e-03, 3.3637232e-05],
       [6.0190509e-06, 3.6105008e-05, 5.6961526e-06, ..., 9.8408400e-06,
        1.3065373e-05, 3.8558664e-04],
       ...,
       [6.0132861e-06, 3.8452290e-05, 9.5863852e-06, ..., 2.8500131e-06,
        3.8434213e-05, 1.6776417e-05],
       [6.8588699e-03, 1.1527174e-04, 9.3395131e-05, ..., 1.5727810e-04,
        1.5858709e-04, 9.5501067e-03],
       [5.4358924e-04, 7.6076240e-06, 7.4367075e-05, ..., 1.5606291e-02,
        7.0339302e-04, 2.3805624e-05]], dtype=float32)
```

```
predictions.shape
```

```
(200, 120)
```

```
len(y_val)
```

```
200
```

```
len(unique_breeds)
```

```
120
```

```
predictions[0]
```

```
array([9.72172595e-04, 8.75414771e-05, 9.60098987e-05, 2.91923097e-05,
       2.33773331e-04, 1.79899889e-05, 6.64141849e-02, 8.62331945e-05,
       5.40716937e-05, 5.77388506e-04, 4.17529314e-04, 2.19266833e-04,
       5.53978258e-04, 1.83912649e-04, 7.41863158e-04, 5.63672977e-04,
       4.88770056e-05, 1.42390728e-01, 7.35380172e-06, 8.61210938e-05,
       6.75975461e-04, 4.34503920e-04, 6.80295561e-05, 9.16461460e-04,
       2.90728694e-05, 1.70433195e-04, 1.94974318e-01, 2.25060885e-05,
       1.47873419e-03, 1.29889743e-03, 1.11630070e-04, 4.51526372e-04,
       1.59131800e-04, 6.59524449e-06, 2.25171563e-04, 1.25106052e-01,
       5.77629144e-05, 4.02979291e-04, 3.45461798e-04, 1.30779707e-04,
       1.85574245e-04, 5.65533492e-06, 4.18773488e-05, 9.36616343e-05,
       2.85254246e-05, 5.73552315e-05, 3.61771126e-05, 2.62562971e-04,
       5.42615075e-04, 2.34073494e-04, 6.20982901e-04, 1.81157811e-04,
       4.76526795e-04, 3.00647898e-05, 1.46512211e-05, 5.73798534e-05,
       2.44878000e-04, 6.89897104e-04, 6.52588089e-04, 6.26748577e-02,
       4.24687489e-04, 1.37106481e-05, 1.69561640e-03, 8.57205305e-05,
       1.55567410e-04, 1.33058513e-02, 5.64681191e-04, 5.21543479e-05,
       1.52107328e-02, 3.49637179e-04, 1.62295625e-02, 6.09972885e-05,
       1.25071922e-04, 1.78175960e-02, 9.78214899e-04, 7.73988140e-05,
       9.67146922e-03, 4.73528262e-03, 8.44162249e-04, 3.91914435e-02,
       1.48332946e-03, 2.50510802e-03, 6.89533714e-04, 1.17409611e-02,
       7.45182333e-05, 4.38690739e-04, 1.53253903e-04, 5.74932958e-04,
       5.34356455e-04, 1.30480598e-03, 3.47327278e-03, 3.37501842e-04,
       2.94368710e-05, 2.27534748e-03, 4.77138638e-05, 1.68356608e-04,
       1.20593887e-03, 8.29031505e-03, 1.36170609e-04, 1.91321888e-05,
       1.57114456e-03, 1.22160418e-04, 1.14414066e-01, 7.10999146e-02,
       5.18146029e-04, 2.89751624e-04, 1.15241474e-02, 6.22391599e-05,
       3.99534365e-05, 2.74459217e-02, 1.37037470e-03, 3.16238147e-04,
       5.97338658e-05, 5.61064844e-05, 1.13305781e-04, 1.19524138e-05,
       3.70255928e-03, 1.97486152e-04, 9.69737721e-06, 2.02395208e-03],
      dtype=float32)
```

```
len(predictions[0])
```

```
120
```

```
np.sum(predictions[0])
```

```
1.0
```

```
# First prediction
index = 1
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])]}")
```

```
[3.52703151e-03 1.17976102e-03 5.23554273e-02 5.83241228e-04
 1.15045032e-03 5.21558359e-05 5.14031900e-03 7.07067200e-04
 7.22123194e-04 3.97640775e-04 6.76966883e-05 2.49025852e-05
 6.47070628e-05 1.55918005e-05 2.82913461e-05 2.62479327e-04
 2.73790141e-03 1.42404647e-03 1.50975495e-04 1.87091515e-04
 9.71950067e-04 2.68230051e-05 9.07632129e-05 1.57495335e-04
 4.38295720e-05 2.21054943e-04 7.93318823e-02 1.90733548e-03
 9.23499465e-05 6.31430885e-04 3.47015361e-04 7.22439363e-05
 6.19931379e-04 1.78689952e-05 1.40718868e-04 5.04427415e-04
 7.56159570e-05 2.67024210e-04 9.11130628e-05 4.96871107e-05
 2.82526569e-04 4.60942210e-05 5.81133600e-06 9.68285385e-05
 1.04763240e-04 2.17579026e-03 9.71773465e-04 8.43899033e-05
 9.27635431e-02 2.91082961e-05 6.36857731e-05 1.50746011e-04
 1.41874943e-05 8.24581221e-05 1.68165643e-04 1.85578736e-03
 4.50996340e-05 8.29389412e-03 1.26109138e-04 6.66809443e-04
 1.29066364e-04 2.35957032e-06 1.33676331e-05 3.66795575e-05
 8.28663586e-04 3.05591857e-05 2.98048166e-04 5.92862838e-04
 9.52771399e-04 1.23656464e-05 2.99298612e-04 2.95579463e-04
 1.60267716e-03 1.50763779e-04 2.28735153e-02 8.61352964e-05
 4.49844083e-04 2.05189064e-01 7.96338718e-05 1.13678754e-04
 1.42884976e-03 2.64319475e-03 1.88958889e-04 6.23063243e-05
 1.17690222e-04 1.87295827e-05 5.68777148e-04 1.19143569e-05
 3.15778088e-05 2.53422604e-05 9.80431068e-05 1.65123347e-05
 3.83040096e-05 4.15478899e-05 1.64365527e-04 1.37801166e-03
 4.28496391e-01 8.85046180e-03 2.30562538e-04 2.80108390e-04
 1.19881159e-04 9.66366570e-05 1.86745077e-04 5.79642190e-04
 1.08480884e-03 2.58884538e-04 3.95039134e-02 5.64440488e-05
 4.64348341e-05 1.33196911e-04 5.84406262e-05 3.51003604e-04
 3.96313590e-05 5.71592182e-06 4.61707241e-05 1.09450379e-03
 9.18972120e-03 6.64338120e-04 3.25455097e-03 3.36372323e-05]
Max value (probability of prediction): 0.428496390581131
Sum: 0.9999999403953552
Max index: 96
Predicted label: scotch_terrier
```

Having the above functionality is great but want to be able to do it at scale. It would be great if I could see the image the prediction is being made on!

Note: Prediction propabilities are alson known as confidence levels.

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

# Get a predicted label based on an array of prediction probabilities
```

```
pred_label = get_pred_label(predictions[91])
pred_label
```

```
'pomeranian'
```

```
# Create a function to unbacth a batched dataset
def unbatchify(data):
  """
  Takes a batched dataset of (image, labels) Tensors and returns separate arrays of images an
  """
  images = []
  labels = []

  # loop through unbatched data
  for image, label in data.unbatch().as_numpy_iterator():
    images.append(image)
    labels.append(unique_breeds[np.argmax(label)])
  return images, labels

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

```
(array([[[0.29599646, 0.4328487 , 0.30566907],
         [0.26635826, 0.32996926, 0.22846505],
         [0.31428418, 0.2770141 , 0.22934894],
         ...,
         [0.7761434 , 0.8232022 , 0.81015944],
         [0.8129115 , 0.828535  , 0.84069437],
         [0.82092965, 0.82637364, 0.84236676]],

        [[0.23448709, 0.31603682, 0.19543913],
         [0.3414841 , 0.36560842, 0.27241898],
         [0.45016074, 0.4011709 , 0.33964607],
         ...,
         [0.76639867, 0.81341374, 0.8135083 ],
         [0.7304248 , 0.75012016, 0.7659073 ],
         [0.74518913, 0.7600257 , 0.7830808 ]],

        [[0.30157745, 0.3082587 , 0.2101833 ],
         [0.2905954 , 0.27066195, 0.18401104],
         [0.4138316 , 0.36170745, 0.2964005 ],
         ...,
         [0.7987162 , 0.84185344, 0.8606442 ],
         [0.79577374, 0.8285994 , 0.8605654 ],
         [0.7518163 , 0.7790497 , 0.81552553]],

        ...,

        [[0.97467786, 0.9878954 , 0.9342278 ],
         [0.9915305 , 0.99772066, 0.94278556],
         [0.98925114, 0.9792081 , 0.9137933 ],
```

```
        ...,
        [0.09876009, 0.09876009, 0.09876009],
        [0.05703771, 0.05703771, 0.05703771],
        [0.03600177, 0.03600177, 0.03600177]],

       [[0.9819785 , 0.98206586, 0.937941  ],
        [0.98119915, 0.9701541 , 0.91256475],
        [0.97223157, 0.9366602 , 0.86971855],
        ...,
        [0.09682597, 0.09682597, 0.09682597],
        [0.07196062, 0.07196062, 0.07196062],
        [0.0361607 , 0.0361607 , 0.0361607 ]],

       [[0.97279435, 0.9545953 , 0.9238974 ],
        [0.963602  , 0.9319913 , 0.8840748 ],
        [0.96271574, 0.91253304, 0.84603375],
        ...,
        [0.08394483, 0.08394483, 0.08394483],
        [0.0886985 , 0.0886985 , 0.0886985 ],
        [0.04514172, 0.04514172, 0.04514172]]], dtype=float32), 'cairn')
```

So far, i've got:

- Prediction labels;
- Validation labels(truth labels);
- Validation Images.

Make a function to make all these a bit more visual friendly.

```
# Create a function which takes an array of preds probabilities, an array of truth labels and
def plot_pred(prediction_probabilities, labels, images, n=1):
  """
  View the prefiction, ground truth and image for sample n
  """
  pred_prob, true_label, image = prediction_probabilities[n], labels[n], images[n]

  # get the pred label
  pred_label = get_pred_label(pred_prob)

  # Plot image and 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'
```

```
  # 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=5)
```

bedlington_terrier 97% bedlington_terrier

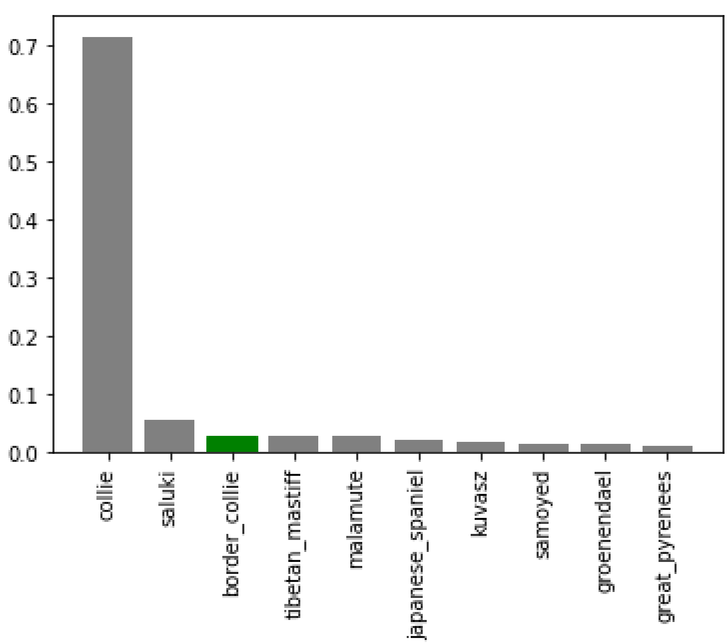
Make a function to view models to 10 predictions.

This function will:

- Take an input of prediction probabilites array and a ground truth array and an integer;
- Find a prediction using `get_pred_label()`;
- Find the top 10:
    - Prediction probabilities indexes;
    - Prediction probabilities values;
    - Prediction labels;
- Plot the top 10 prediction probability values and labels, coloring the true label green.

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

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


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




```
# Check out a few predictions and their deifferent 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(h_pad=1.0)
plt.show()
```

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 correct time
  modeldir = os.path.join("/content/drive/MyDrive/Colab Notebooks/Dog Vision/models",
                          datetime.datetime.now().strftime("%Y%m%d-%H%M%s"))
  model_path = modeldir + "-" + suffix + ".h5" # save format of the model
  print(f"Saving model to : {model_path}...")
  model.save(model_path)
  return model_path
```

```
# Create a function to load a trained model
def load_model(model_path):
  """
  Loads a saved model from specified path.
  """
  print(f"Loading saved model from: {model_path}...")
  model = tf.keras.models.load_model(model_path,
                                     custom_objects={"KerasLayer": hub.KerasLayer})
  return model
```

```
# Saving the model trained on 1000 images
save_model(model, suffix="1000-images-mobilev2-Adam")
```

```
Saving model to : /content/drive/MyDrive/Colab Notebooks/Dog Vision/models/2022010
'/content/drive/MyDrive/Colab Notebooks/Dog Vision/models/20220103-15321641223963
-1000-images-mobilev2-Adam.h5'
```

```
# Load the trained model
load_1000_image_model = load_model("/content/drive/MyDrive/Colab Notebooks/Dog Vision/models/
```

```
Loading saved model from: /content/drive/MyDrive/Colab Notebooks/Dog Vision/models/2022
```

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

```
7/7 [==============================] - 9s 1s/step - loss: 1.3236 - accuracy: 0.6950
[1.3235584497451782, 0.6949999928474426]
```

```
# Evaluate the loaded model
load_1000_image_model.evaluate(val_data)
```

```
7/7 [==============================] - 9s 1s/step - loss: 1.1191 - accuracy: 0.7100
[1.119142770767212, 0.7099999785423279]
```

Training a big dog model 🐶 (on 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 shapes: ((None, 224, 224, 3), (None, 120)), types: (tf.float32, tf.bool)>
```

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

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

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

Note: next cell will take some time (~ 30 min. for the first epoch) because the GPU has to load all of the images into memory

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

```
Epoch 1/100
320/320 [==============================] - 63s 168ms/step - loss: 1.3395 - accuracy: 0.6
Epoch 2/100
320/320 [==============================] - 55s 173ms/step - loss: 0.3992 - accuracy: 0.8
Epoch 3/100
320/320 [==============================] - 75s 234ms/step - loss: 0.2357 - accuracy: 0.9
Epoch 4/100
320/320 [==============================] - 56s 175ms/step - loss: 0.1508 - accuracy: 0.9
```

```
Epoch 5/100
320/320 [==============================] - 54s 168ms/step - loss: 0.1067 - accuracy: 0.9
Epoch 6/100
320/320 [==============================] - 57s 177ms/step - loss: 0.0769 - accuracy: 0.9
Epoch 7/100
320/320 [==============================] - 56s 176ms/step - loss: 0.0582 - accuracy: 0.9
Epoch 8/100
320/320 [==============================] - 57s 178ms/step - loss: 0.0457 - accuracy: 0.9
Epoch 9/100
320/320 [==============================] - 57s 177ms/step - loss: 0.0365 - accuracy: 0.9
Epoch 10/100
320/320 [==============================] - 57s 176ms/step - loss: 0.0297 - accuracy: 0.9
Epoch 11/100
320/320 [==============================] - 56s 176ms/step - loss: 0.0256 - accuracy: 0.9
Epoch 12/100
320/320 [==============================] - 56s 176ms/step - loss: 0.0235 - accuracy: 0.9
Epoch 13/100
320/320 [==============================] - 57s 176ms/step - loss: 0.0197 - accuracy: 0.9
Epoch 14/100
320/320 [==============================] - 57s 179ms/step - loss: 0.0184 - accuracy: 0.9
Epoch 15/100
320/320 [==============================] - 56s 176ms/step - loss: 0.0178 - accuracy: 0.9
Epoch 16/100
320/320 [==============================] - 58s 180ms/step - loss: 0.0160 - accuracy: 0.9
<keras.callbacks.History at 0x7f68f3a9df50>
```

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

```
Saving model to : /content/drive/MyDrive/Colab Notebooks/Dog Vision/models/2022010
'/content/drive/MyDrive/Colab Notebooks/Dog Vision/models/20220103-14071641218874
-full-image-set-mobilenetv2-Adam.h5'
```

```
# Loading the full model
loaded_full_model = load_model("/content/drive/MyDrive/Colab Notebooks/Dog Vision/models/2022
```

```
Loading saved model from: /content/drive/MyDrive/Colab Notebooks/Dog Vision/models/2022
```

Making predictions on the test dataset

Since the model has been trade on images in the form of tensor batches, to make predictions on the test data, need to get the data ub the same format.

Luckily, `create_data_bathes()` can take a list of filname as input and convert them into tensor batches.

To make predictions on the test data, need to:

- 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 bacthces to the `predict()` method called on the model.

```
# Load test image filanames
test_path = ("/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test
test_filenames = [test_path + fname for fname in os.listdir(test_path)]
test_filenames[:10]
```

```
['/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e4e99
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/dfa54e
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/dffba
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e3c97e
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e4c74
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e0072
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/df2d8
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e4819
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e43f6e
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/dog-breed-identification/test/e0536
```

```
len(test_filenames)
```

```
10357
```

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

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

Note: Calling `predict()` on the full model and passing it the test data batch it will take some time to run (` 1hr.).

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

```
324/324 [==============================] - 1141s 4s/step
```

```
# Save predictions (Numpy array) to csv file (for later access)
np.savetxt("/content/drive/MyDrive/Colab Notebooks/Dog Vision/preds_array.csv", test_predicti
```

```
# Load predictions (numpy array) from csv file
```

```
test_prediction = np.loadtxt("/content/drive/MyDrive/Colab Notebooks/Dog Vision/preds_array.c
```

```
test_predictions[:10]
```

```
array([[2.9447148e-04, 7.2730395e-06, 2.4468712e-08, ..., 1.0139988e-07,
        2.9968969e-10, 2.0819679e-07],
       [8.9293308e-06, 2.3130249e-06, 9.0968427e-10, ..., 6.6061068e-10,
        7.5992954e-09, 1.0041420e-09],
       [9.0683711e-11, 2.9163666e-06, 2.0281883e-10, ..., 1.2635759e-05,
        1.0733615e-08, 6.0741878e-09],
       ...,
       [1.3541052e-10, 8.4183237e-11, 4.6548519e-11, ..., 4.7809682e-08,
        6.0870824e-04, 6.6470984e-10],
       [1.2931997e-12, 4.3014568e-11, 3.4622409e-12, ..., 1.8699330e-10,
        3.3734923e-10, 8.2129980e-12],
       [5.9803529e-13, 5.6340681e-08, 1.5912713e-10, ..., 1.9726814e-07,
        5.0620439e-08, 2.3507127e-12]], dtype=float32)
```

```
test_predictions.shape
```

```
(10357, 120)
```

Preparing test dataset prediction for submission

Models predictions probability outputs in a DataFrame with an ID and a column for each different dog breed.

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

id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffor

0 rows × 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	african_hunting_
0	e4e991c432ce6d8e8ba26672ff8fb2f5	NaN	NaN	I
1	dfa54e85c1309d8a9933deedc1d775c7	NaN	NaN	I
2	dffba3aed600801a5f91c5034bec5b08	NaN	NaN	I
3	e3c97ed588b32f49c7aae65cf91f17ba	NaN	NaN	I
4	e4c743b9aaf615dd5fe162bf25f82fb5	NaN	NaN	I

5 rows × 121 columns

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

	id	affenpinscher	afghan_hound	african_hunting_
0	e4e991c432ce6d8e8ba26672ff8fb2f5	0.000294471	7.27304e-06	2.44687
1	dfa54e85c1309d8a9933deedc1d775c7	8.92933e-06	2.31302e-06	9.09684
2	dffba3aed600801a5f91c5034bec5b08	9.06837e-11	2.91637e-06	2.02819
3	e3c97ed588b32f49c7aae65cf91f17ba	0.00109781	1.48128e-09	5.74404
4	e4c743b9aaf615dd5fe162bf25f82fb5	6.14069e-10	1.5763e-09	1.69677

5 rows × 121 columns

```
# save predictions dataframe to csv for submission
preds_df.to_csv("/content/drive/MyDrive/Colab Notebooks/Dog Vision/full_model_predictions_sub
                index=False)
```

Making predictions on custom images

To make predictions on custom need to:

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

```
# Get costum images filepaths
custom_path = "/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/"
custom_image_paths = [custom_path + fname for fname in os.listdir(custom_path)]
```

```
custom_image_paths
```

```
['/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/anna-dudkova-Ce2Fz
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/max-kleinen-Jr_Dk(
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/jana-ohajdova-IqF8
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/anatoly-najmitenk(
 '/content/drive/MyDrive/Colab Notebooks/Dog Vision/custum-dog-photos/wannes-de-mol-lhU1
```

```
# turn cusom images into batch dataset
custom_data = create_data_batches(custom_image_paths, test_data=True)
custom_data
```

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

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

```
custom_preds.shape
```

```
(5, 120)
```

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

```
['airedale',
 'german_shepherd',
 'german_shepherd',
 'leonberg',
 'norwich_terrier']
```

```
# Get custom images ( the unbaychify() function won't work since ther aren't labels)
```

```
custom_images = []
# loop through unbatched data
for image in custom_data.unbatch().as_numpy_iterator():
  custom_images.append(image)


# Check custom image predictions
plt.figure(figsize=(10, 10))
for i , image in enumerate(custom_images):
  plt.subplot(1, 5, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.title(custom_pred_labels[i])
  plt.imshow(image)
```