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

Who's that doggy in the window?

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

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

To do this, we'll be using data from the [Kaggle dog breed identification competition (https://www.kaggle.com/c/dog-breed-identification/overview)](https://www.kaggle.com/c/dog-breed-identification/overview). It consists of a collection of 10,000+ labelled images of 120 different dog breeds.

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

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

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

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

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

Getting our workspace ready

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

1. Import tensorflow
2. Import tensorflow Hub
3. Use GPU

In [2]:

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

In [3]:

```
tf.__version__
```

Out[3]:

```
'2.8.0'
```

In [4]:

```
hub.__version__
```

Out[4]:

```
'0.12.0'
```

Getting our Data ready (Turning into Tensors)

In [5]:

```
import pandas as pd
labels = pd.read_csv("/content/drive/MyDrive/ML/labels.csv")
print(labels.describe())
print(labels.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
```

In [6]:

```
labels.head()
```

Out[6]:

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

In [7]:

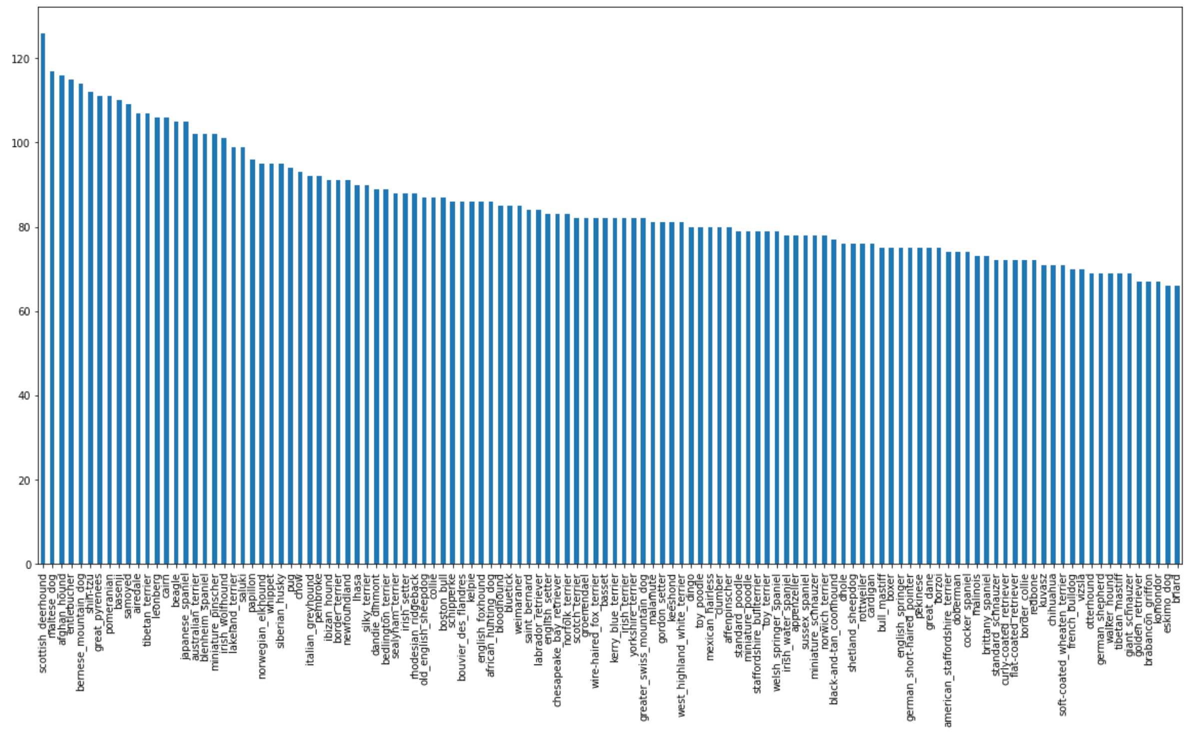
```
labels["breed"].value_counts()
```

Out[7]:

```
scottish_deerhound      126
maltese_dog             117
afghan_hound            116
entlebucher             115
bernese_mountain_dog    114
                       ...
golden_retriever         67
brabancon_griffon        67
komondor                 67
eskimo_dog               66
briard                   66
Name: breed, Length: 120, dtype: int64
```

In [8]:

```
labels["breed"].value_counts().plot.bar(figsize=(20,10));
```



Getting Images and their labels

In [9]:

```
# Create pathnames from ID

filenames = ["/content/drive/MyDrive/ML/train/" + fname +".jpg" for fname in labels["id"]]
filenames[:10]
```

Out[9]:

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

In [10]:

```
# Check whether no. of file names matches with actual image files

import os
if len(os.listdir("/content/drive/MyDrive/ML/train")) == len(filenames):
    print("No. of file names matches no. of images")
else:
    print("No. of file names does not matches no. of images")
```

```
No. of file names does not matches no. of images
```

Now, lets prepaare our labels

In [11]:

```
import numpy as np
labels2 = labels["breed"]
labels2 = np.array(labels2)
labels2
```

Out[11]:

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

In [12]:

```
len(labels2)
```

Out[12]:

```
10222
```

In [13]:

```
len(labels)
```

Out[13]:

```
10222
```

In [14]:

```
if len(labels2) == len(filenames):
  print("no. of labels matches no. of filenames")
else:
  print("no. of labels matches does not no. of filenames")
```

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

In [15]:

```
# find the unique labels
unique_breeds = np.unique(labels2)
unique_breeds
```

Out[15]:

```
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)
```

In [16]:

```
len(unique_breeds)
```

Out[16]:

```
120
```

In [17]:

```
# Turning every label into boolean array
boolean_labels = [label == unique_breeds for label in labels2]
boolean_labels[:2]
```

Out[17]:

```
[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])]
```

In [18]:

```
len(boolean_labels)
```

Out[18]:

```
10222
```

Creating our own validation set

In [19]:

```
# Setup x and y
x = filenames
y = boolean_labels
```

we are going to start experimenting with ~1000 samples and increase as needed

In [20]:

```
# set up no. of images to use for experimenting
NUM_IMAGES = 1000 #@param {type:"slider", min:1000, max:10000, step:1000}
```

In [21]:

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

# Split into training and validation sets of 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)
```

Out[21]:

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

In [22]:

```
x_train[:1], y_train[:1]
```

Out[22]:

```
(['/content/drive/MyDrive/ML/train/00bee065dcec471f26394855c5c2f3de.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])])
```

Preprocessing images(Turning images into Tensors)

To Preprocess images into tensors we're going to write function which does few things :

1. Take an image filepath as input.
2. Use tensorflow to read the file and save it to a variable `image` .
3. Turn our `image` (a jpeg) into Tensors.

4. Normalize out `image` (Convert color channel from 0-255 to 0-1)
5. Resize the `image` to a shape of (224,224).
6. Return the modified `image`.

Importing an Image

In [23]:

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

Out[23]:

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

(Hieght, Widht, Color channel(RGB)), Value is between 0 and 255

In [24]:

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

Out[24]:

```
<tf.Tensor: shape=(2, 350, 3), dtype=uint8, numpy=
array([[[ 89, 137,  87],
        [ 76, 124,  74],
        [ 63, 111,  59],
        ...,
        [ 76, 134,  86],
        [ 76, 134,  86],
        [ 76, 134,  86]],

       [[ 72, 119,  73],
        [ 67, 114,  68],
        [ 63, 111,  63],
        ...,
        [ 75, 131,  84],
        [ 74, 132,  84],
        [ 74, 131,  86]]], dtype=uint8)>
```

Writing a function to turn images into Tensors

In [25]:

```
# Define image size
IMG_SIZE = 224

# Create a function for preprocessing image
def process_image(image_path, img_size = IMG_SIZE):
  """
      Takes a image file path and turns it into Tensor
  """
  # Read an image file
  image = tf.io.read_file(image_path)
  # Turn image into numerical tensor with 3 color channel
  image = tf.image.decode_jpeg(image, channels=3)
  # Convert the color channel values from 0-255 to 0-1 values (Normalization)
  image = tf.image.convert_image_dtype(image, tf.float32)
  # Resize the image
  image = tf.image.resize(image, size=[IMG_SIZE, IMG_SIZE])


  return image
```

Turning our data into batches.

Why turn our data into batches??

If we try to process entire 10000+ images in one go, then all might not fit into memory.

So that's why we do about 32 (batch size) images at a time. We can manually set batch size if needed.

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

In [26]:

```
# Create a simple function to return a tuple(image, label)

def get_image_label(image_path, label):
  """ Takes an image file path name and associated label,
     processes the image and returns the tuple of (image, label)"""
  image = process_image(image_path)
  return image, label
```

In [27]:

```
# Demo of above function
(process_image(x[42]), tf.constant(y[42]))
```

Out[27]:

```
(<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 way to turn our data into tuples of Tensors, let's make a fucntion to turn all our data (x & y) into batches.

In [94]:

```
# Define batch size
BATCH_SIZE = 32

# Create a function to turn our data into batches

def create_data_batches(x, y=None, batch_size=BATCH_SIZE, valid_data = False, test_data = F
  """
  Creates batches of data out of image (x) and label (y) pairs.
  Shuffles the data if it's training data but does'nt shuffle if it's validation data.
  Also accepts test data as input (no labels).
  """
  # If data is test dataset , we will not have labels
  if test_data:
    print("Creating test data batches...")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(x))) # only filepaths
    data_batch = data.map(process_image).batch(BATCH_SIZE)
    return data_batch

  # If dataset is valid set then we don't need to shuffle it
  elif valid_data:
    print("Creating validation data batches......")
    data = tf.data.Dataset.from_tensor_slices((tf.constant(x),
                                               tf.constant(y)))
    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
    data = data.shuffle(buffer_size=len(x))

    # Create (image, label) tuple
    data = data.map(get_image_label)

    data_batch = data.batch(BATCH_SIZE)

  return data_batch
```

In [29]:

```
train_data = create_data_batches(x_train, y_train)
valid_data = create_data_batches(x_val, y_val, valid_data=True)
```

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

In [30]:

```
# Checkout different attributes of our data batches
train_data.element_spec, valid_data.element_spec
# None = Batch size
```

Out[30]:

```
((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 our data batches

In [31]:

```
import matplotlib.pyplot as plt

def show_25_images(images, labels):
  """
  Displays a plot of 25 images and their labels from data batches
  """
  plt.figure(figsize=(10,10))
  for i in range(25):
    # Create subplots
    ax = plt.subplot(5, 5, i+1)
    plt.imshow(images[i])
    plt.title(unique_breeds[labels[i].argmax()])
    plt.axis("off")
```

In [32]:

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

In [33]:

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

Out[33]:

```
(32, 32)
```

In [34]:

```
show_25_images(train_images, train_labels)
```

In [35]:

```
val_images, val_label = next(valid_data.as_numpy_iterator())
show_25_images(val_images, val_label)
```

Building a model

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

- Input shape - Turn our images into form of Tensors.

- Output shape - Turn image labels into Tensors.
- The URL of the model we want to use from Tensorflow hub.

In [36]:

```
# Setup input shape
INPUT_SHAPE = [None, IMG_SIZE, IMG_SIZE, 3] # batch, height, width, color channel.

# Setup output shape
OUTPUT_SHAPE = len(unique_breeds)

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

Now let's put our input, output and model into Keras deep learning model.

Creating a function which will:

- Takes input shape, output shape and the model we have chosen as parameters.
- Defines the layers in karas model in sequential fashion.
- Evaluates and improve the model.(Compile the model)
- Builds the model.
- Returns the model.

In [37]:

```
# Create a function which builds keras model.
def create_model(input_shape = INPUT_SHAPE, output_shape=OUTPUT_SHAPE, model_url = MODEL_UR
  print("Building model with ", MODEL_URL)

  # Setup the model layer
  model = tf.keras.Sequential([
                               hub.KerasLayer(MODEL_URL), # Layer 1(input layer)
                               tf.keras.layers.Dense(units=OUTPUT_SHAPE,
                                                     activation="softmax") # Layer 2 (outpu
])

   # Compile the model
  model.compile(
   loss=tf.keras.losses.CategoricalCrossentropy(),
   optimizer = tf.keras.optimizers.Adam(),
   metrics= ["accuracy"]
             )

  # Build a model
  model.build(INPUT_SHAPE)
  return model
```

In [38]:

```
model = create_model()
model.summary()
```

```
Building model with  https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/
classification/5 (https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/cla
ssification/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
_________________________________________________________________
```

Creating callbacks

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

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

TensorBoard callback

To setup TensorBoard callback we need to do 3 things:

1. Load the TensorBoard 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 our model's training logs with the `%tensorboard` magic function.

In [39]:

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

In [40]:

```
import datetime

# Creating a fuction to build a TensorBoard callback
def create_tensorboard_callback():
  logdir = os.path.join("/content/drive/MyDrive/ML/logs",
                        datetime.datetime.now().strftime("%y%m%d-%H%M%S"))
  return tf.keras.callbacks.TensorBoard(logdir)
```

Early stopping callback

Early stopping helps our model from overfitting by stoping training if a certain evaluation matric stops improving.

In [41]:

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

Training a model (on subset of data)

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

In [42]:

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

Let's create a function which trains a model

- Creating 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, no. of epochs to train, and callbacks we'd like to use.
- Return the model.

In [43]:

```
def train_model():
  """
  trains and returns a trained model
  """
  # Create a model
  model = create_model()

  # Create a TensorBoard session everytime we train a model
  tensorboard = create_tensorboard_callback()

  # Fit the model to the data
  model.fit(x=train_data,
            epochs=NUM_EPOCHS,
            validation_data=valid_data,
            validation_freq = 1,
            callbacks = [tensorboard, early_stopping])
  return model
```

In [44]:

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

```
Building model with  https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/
classification/5 (https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/cla
ssification/5)
Epoch 1/100
25/25 [==============================] - 137s 4s/step - loss: 4.7448 - accur
acy: 0.0862 - val_loss: 3.6234 - val_accuracy: 0.1750
Epoch 2/100
25/25 [==============================] - 5s 185ms/step - loss: 1.7106 - accu
racy: 0.6662 - val_loss: 2.2184 - val_accuracy: 0.4650
Epoch 3/100
25/25 [==============================] - 5s 183ms/step - loss: 0.5814 - accu
racy: 0.9350 - val_loss: 1.6958 - val_accuracy: 0.5750
Epoch 4/100
25/25 [==============================] - 5s 198ms/step - loss: 0.2605 - accu
racy: 0.9887 - val_loss: 1.4920 - val_accuracy: 0.6350
Epoch 5/100
25/25 [==============================] - 5s 196ms/step - loss: 0.1483 - accu
racy: 0.9937 - val_loss: 1.4050 - val_accuracy: 0.6500
Epoch 6/100
25/25 [==============================] - 5s 193ms/step - loss: 0.1009 - accu
racy: 1.0000 - val_loss: 1.3558 - val_accuracy: 0.6600
Epoch 7/100
25/25 [==============================] - 5s 196ms/step - loss: 0.0761 - accu
racy: 1.0000 - val_loss: 1.3189 - val_accuracy: 0.6650
Epoch 8/100
25/25 [==============================] - 5s 185ms/step - loss: 0.0598 - accu
racy: 1.0000 - val_loss: 1.2932 - val_accuracy: 0.6700
Epoch 9/100
25/25 [==============================] - 5s 182ms/step - loss: 0.0493 - accu
racy: 1.0000 - val_loss: 1.2734 - val_accuracy: 0.6700
```

Checking the TensorBoard logs

In [45]:

```
%tensorboard --logdir /content/drive/MyDrive/ML/logs
```

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

In [46]:

```
valid_data
```

Out[46]:

```
<BatchDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.f
loat32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))
>
```

Making predictions

In [47]:

```
predictions = model.predict(valid_data, verbose=1)
predictions
```

```
7/7 [==============================] - 2s 166ms/step
```

Out[47]:

```
array([[3.68647953e-03, 3.29027898e-05, 9.73858114e-05, ...,
        1.97379733e-04, 2.47712778e-05, 2.44005444e-03],
       [3.14045884e-03, 2.83804885e-03, 1.94703266e-02, ...,
        5.60831628e-04, 6.15788298e-03, 9.70339534e-05],
       [7.60305602e-06, 4.03997292e-05, 6.48998321e-05, ...,
        7.95382191e-04, 4.89898521e-05, 2.04039519e-04],
       ...,
       [2.35884709e-05, 1.06828476e-04, 4.45570804e-05, ...,
        1.94761105e-05, 1.01176076e-04, 9.15998244e-05],
       [2.14959215e-03, 1.49096071e-04, 1.67374732e-04, ...,
        2.56708066e-04, 1.00156227e-04, 1.64738130e-02],
       [1.73756955e-04, 3.77618344e-05, 6.66299311e-04, ...,
        2.30563316e-03, 3.85021628e-03, 2.66933930e-04]], dtype=float32)
```

In [48]:

```
index = 42
print(predictions[index])
print(f"Max value : {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])]}")
```

```
[1.32686400e-04 1.04028128e-04 6.30661089e-05 3.73838819e-04
 1.26218016e-03 1.15187191e-04 5.27475087e-04 2.17620120e-03
 7.69787375e-03 6.69648126e-02 7.88318139e-05 3.37397978e-05
 1.29143125e-03 2.79874750e-03 2.52222089e-04 1.00734911e-03
 1.12848691e-04 5.30809397e-04 6.33701507e-04 6.80103386e-03
 8.87547067e-05 4.90600651e-04 2.51234324e-05 1.63808960e-04
 3.32808099e-03 5.77416722e-05 5.80950073e-05 1.95284287e-04
 2.74279068e-04 3.52292845e-05 5.82407883e-05 4.25756123e-04
 1.08046355e-04 1.63998091e-04 7.04070189e-05 1.18516640e-04
 1.01031379e-04 1.47979474e-03 1.50120992e-04 1.74586073e-01
 4.11863613e-04 2.03474792e-05 7.40757724e-03 2.57977044e-05
 1.54777517e-04 1.23024336e-04 3.25265457e-04 1.67969358e-03
 5.42076523e-05 2.10965649e-04 2.45431729e-04 4.72492044e-04
 8.73763638e-04 4.60181665e-03 5.66133494e-05 7.94771186e-04
 3.04986723e-04 2.96462895e-05 7.52007918e-06 2.63973652e-05
 2.76397186e-04 1.00323767e-03 7.59737886e-05 5.43397400e-05
 1.65215155e-04 7.64621072e-05 7.99155459e-05 1.40765085e-04
 1.64350684e-04 4.30519140e-05 7.51464104e-05 2.24576943e-04
 3.63724452e-04 4.52392414e-04 9.16139179e-05 3.61337326e-04
 2.32767517e-04 1.51351225e-04 4.36230875e-05 3.22382461e-04
 1.25892466e-05 1.41592565e-04 2.39153480e-04 7.21439195e-04
 1.77808630e-03 1.51458502e-04 1.28340296e-04 8.42180725e-06
 3.42117892e-05 8.62658781e-04 4.51138709e-04 3.33141979e-05
 1.75982446e-03 1.97185465e-04 1.09741914e-05 1.39776239e-04
 2.90858825e-05 1.20599158e-04 3.68484616e-05 1.33394627e-04
 8.92168682e-05 6.37568955e-05 1.93439817e-04 2.05651450e-04
 2.60475150e-04 3.89227353e-05 7.71489809e-04 3.52016796e-05
 2.11633684e-04 3.00607731e-04 1.41197335e-04 2.87668919e-03
 1.34193129e-03 6.85863554e-01 1.39036390e-04 1.60163268e-03
 7.98768306e-05 2.27499640e-05 6.40118378e-04 9.70125140e-04]
Max value : 0.6858635544776917
Sum : 1.0
Max index : 113
Predicted label : walker_hound
```

Note Prediction probabilities are also known as confidence levels.

In [49]:

```
# Turn prediction probabilities into labels
def get_pred_label(prediction_probabilities):
  "Turn an array of prediction probabilities into labels"
  return unique_breeds[np.argmax(prediction_probabilities)]
```

In [50]:

```
get_pred = get_pred_label(predictions[81])
get_pred
```

Out[50]:

```
'dingo'
```

We'll have to unbatch our validation data set to make predictions on validation labels

In [51]:

```
def unbatchify(data):
  images = []
  labels = []
  # Loop through ubatch data
  for image, label in data.unbatch().as_numpy_iterator():
    images.append(image)
    labels.append(unique_breeds[np.argmax(label)])
  return images, labels
```

In [52]:

```
val_images, val_labels = unbatchify(valid_data)
val_images[0], val_labels[0]
```

Out[52]:

```
(array([[[0.29599646, 0.43284872, 0.3056691 ],
         [0.26635826, 0.32996926, 0.22846507],
         [0.31428418, 0.2770141 , 0.22934894],
         ...,
         [0.77614343, 0.82320225, 0.8101595 ],
         [0.81291157, 0.8285351 , 0.8406944 ],
         [0.8209297 , 0.8263737 , 0.8423668 ]],

        [[0.2344871 , 0.31603682, 0.19543913],
         [0.3414841 , 0.36560842, 0.27241898],
         [0.45016077, 0.40117094, 0.33964607],
         ...,
         [0.7663987 , 0.8134138 , 0.81350833],
         [0.7304248 , 0.75012016, 0.76590735],
         [0.74518913, 0.76002574, 0.7830809 ]],

        [[0.30157745, 0.3082587 , 0.21018331],
         [0.2905954 , 0.27066195, 0.18401104],
         [0.4138316 , 0.36170745, 0.2964005 ],
         ...,
         [0.79871625, 0.8418535 , 0.8606443 ],
         [0.7957738 , 0.82859945, 0.8605655 ],
         [0.75181633, 0.77904975, 0.8155256 ]],

        ...,

        [[0.9746779 , 0.9878955 , 0.9342279 ],
         [0.99153054, 0.99772066, 0.9427856 ],
         [0.98925114, 0.9792082 , 0.9137934 ],
         ...,
         [0.0987601 , 0.0987601 , 0.0987601 ],
         [0.05703771, 0.05703771, 0.05703771],
         [0.03600177, 0.03600177, 0.03600177]],

        [[0.98197854, 0.9820659 , 0.9379411 ],
         [0.9811992 , 0.97015417, 0.9125648 ],
         [0.9722316 , 0.93666023, 0.8697186 ],
         ...,
         [0.09682598, 0.09682598, 0.09682598],
         [0.07196062, 0.07196062, 0.07196062],
         [0.0361607 , 0.0361607 , 0.0361607 ]],

        [[0.97279435, 0.9545954 , 0.92389745],
         [0.963602  , 0.93199134, 0.88407487],
         [0.9627158 , 0.9125331 , 0.8460338 ],
         ...,
         [0.08394483, 0.08394483, 0.08394483],
         [0.0886985 , 0.0886985 , 0.0886985 ],
         [0.04514172, 0.04514172, 0.04514172]]], dtype=float32), 'cairn')
```

In [53]:

```
def plot_pred(prediction_probabilities, labels, images, n=1):
  "Views the prediction, ground truth and image for sample n"
  pred_prob, truth_label, image = prediction_probabilities[n], labels[n], images[n]

  pred_label = get_pred_label(pred_prob)

  plt.imshow(image)
  plt.xticks([])
  plt.yticks([])

  if pred_label == truth_label:
    color = "green"
  else:
    color = "red"

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

In [54]:

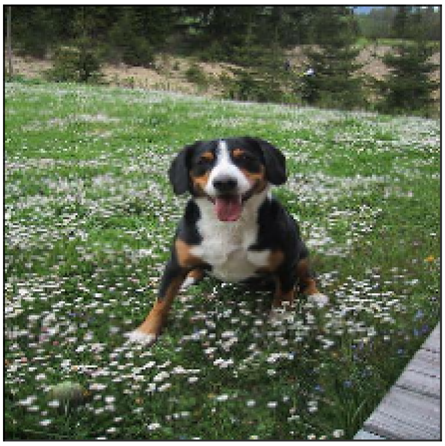
```
plot_pred(prediction_probabilities=predictions,
          labels=val_labels,
          images=val_images,
          n=16)
```

irish_setter 93% irish_setter

In [55]:

```
plot_pred(prediction_probabilities=predictions,
          labels=val_labels,
          images=val_images,
          n=97)
```

greater_swiss_mountain_dog 60% entlebucher

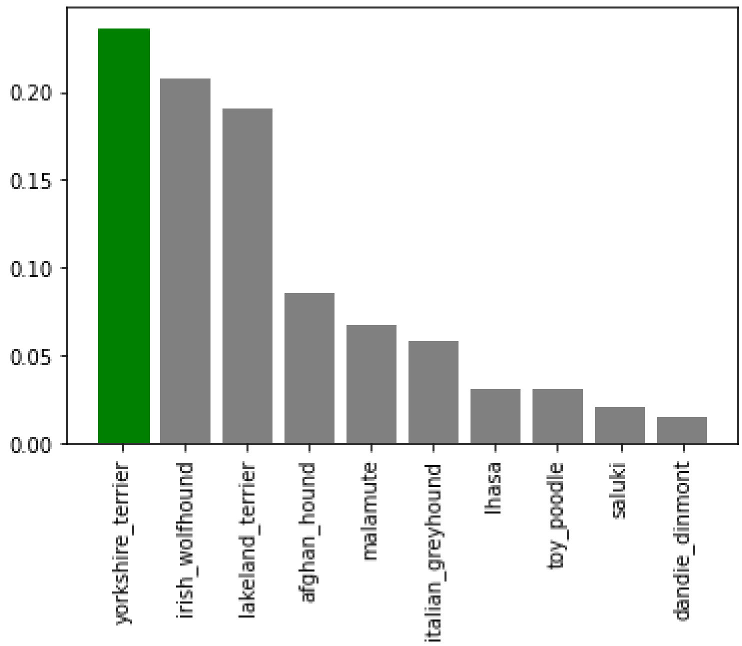


In [56]:

```
def plot_pred_conf(prediction_probabilities, labels, n =1):
  pred_prob, true_label = prediction_probabilities[n], labels[n]

  pred_label = get_pred_label(pred_prob)

  top_10_pred_indexes = pred_prob.argsort()[-10:][::-1]

  top_10_pred_values = pred_prob[top_10_pred_indexes]

  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")

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

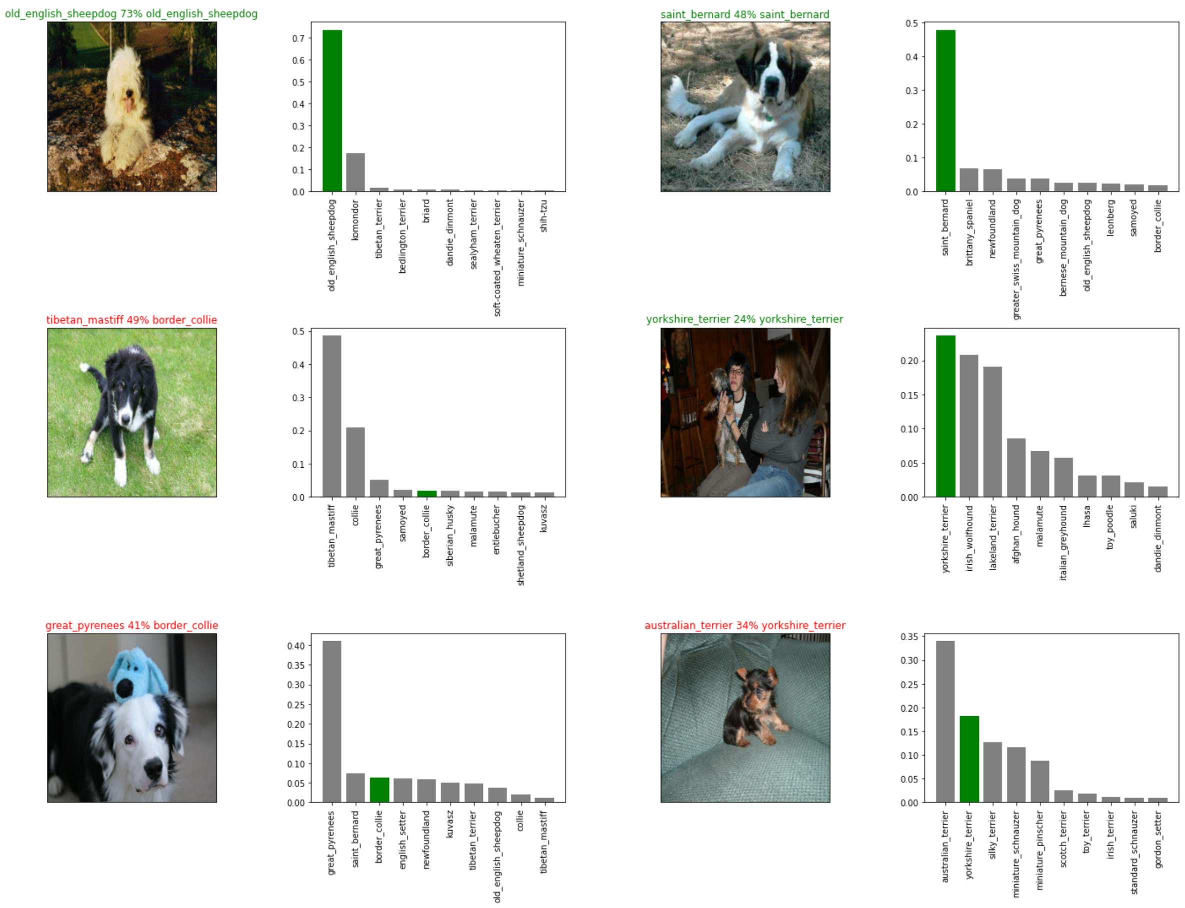
In [57]:

```
plot_pred_conf(prediction_probabilities  = predictions,
               labels = val_labels,
               n=10)
```



In [58]:

```
# Let's check a few predictions and their different values
i_multiplier = 7
num_rows = 3
num_cols = 2
num_images = num_rows*num_cols
plt.figure(figsize=(5*2*num_cols, 5*num_rows))
for i in range(num_images):
  plt.subplot(num_rows, 2*num_cols, 2*i+1)
  plot_pred(prediction_probabilities=predictions,
            labels=val_labels,
            images=val_images,
            n=i+i_multiplier)
  plt.subplot(num_rows, 2*num_cols, 2*i+2)
  plot_pred_conf(prediction_probabilities=predictions,
                labels=val_labels,
                n=i+i_multiplier)
plt.tight_layout(h_pad=1.0)
plt.show()
```



Saving and reloading a model

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

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

In [59]:

```
def save_model(model, suffix=None):
  """
  Saves a given model in a models directory and appends a suffix (str)
  for clarity and reuse.
  """
  # Create model directory with current time
  modeldir = os.path.join("/content/drive/MyDrive/ML/model",
                          datetime.datetime.now().strftime("%Y%m%d-%H%M%s"))
  model_path = modeldir + "-" + suffix + ".h5" # save format of model
  print(f"Saving model to: {model_path}...")
  model.save(model_path)
  return model_path
```

In [60]:

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

```
Saving model to: /content/drive/MyDrive/ML/model/20220309-05581646805527-100
0-images-model.h5...
```

Out[60]:

```
'/content/drive/MyDrive/ML/model/20220309-05581646805527-1000-images-model.h
5'
```

In [61]:

```
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
```

In [62]:

```
# Load our model trained on 1000 images
model_1000_images = load_model('/content/drive/MyDrive/ML/model/20220308-09521646733164-100
```

```
Loading saved model from: /content/drive/MyDrive/ML/model/20220308-095216467
33164-1000-images-model.h5
```

In [63]:

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

```
7/7 [==============================] - 1s 121ms/step - loss: 1.2734 - accura
cy: 0.6700
```

Out[63]:

```
[1.2734367847442627, 0.6700000166893005]
```

In [64]:

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

```
7/7 [==============================] - 2s 119ms/step - loss: 1.3570 - accura
cy: 0.6500
```

Out[64]:

```
[1.3569968938827515, 0.6499999761581421]
```

Training model on full data

In [65]:

```
full_data = create_data_batches(x, y)
```

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

In [66]:

```
full_data
```

Out[66]:

```
<BatchDataset element_spec=(TensorSpec(shape=(None, 224, 224, 3), dtype=tf.f
loat32, name=None), TensorSpec(shape=(None, 120), dtype=tf.bool, name=None))
>
```

In [67]:

```
# Create a full model

full_model=create_model()
```

Building model with https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/5 (https://tfhub.dev/google/imagenet/mobilenet_v2_130_224/classification/5)

In [68]:

```
# Create full model callbacks

full_model_tensorboard = create_tensorboard_callback()

full_model_early_stopping = tf.keras.callbacks.EarlyStopping(monitor="accuracy", patience=3
```

In [69]:

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

```
Epoch 1/100
320/320 [==============================] - 57s 164ms/step - loss: 1.3694 - a
ccuracy: 0.6651
Epoch 2/100
320/320 [==============================] - 53s 166ms/step - loss: 0.3991 - a
ccuracy: 0.8860
Epoch 3/100
320/320 [==============================] - 52s 161ms/step - loss: 0.2372 - a
ccuracy: 0.9365
Epoch 4/100
320/320 [==============================] - 52s 163ms/step - loss: 0.1548 - a
ccuracy: 0.9626
Epoch 5/100
320/320 [==============================] - 53s 164ms/step - loss: 0.1084 - a
ccuracy: 0.9770
Epoch 6/100
320/320 [==============================] - 50s 156ms/step - loss: 0.0771 - a
ccuracy: 0.9861
Epoch 7/100
320/320 [==============================] - 54s 169ms/step - loss: 0.0599 - a
ccuracy: 0.9907
Epoch 8/100
320/320 [==============================] - 54s 168ms/step - loss: 0.0466 - a
ccuracy: 0.9938
Epoch 9/100
320/320 [==============================] - 56s 175ms/step - loss: 0.0378 - a
ccuracy: 0.9961
Epoch 10/100
320/320 [==============================] - 55s 171ms/step - loss: 0.0309 - a
ccuracy: 0.9977
Epoch 11/100
320/320 [==============================] - 55s 171ms/step - loss: 0.0270 - a
ccuracy: 0.9976
Epoch 12/100
320/320 [==============================] - 53s 167ms/step - loss: 0.0242 - a
ccuracy: 0.9977
Epoch 13/100
320/320 [==============================] - 55s 171ms/step - loss: 0.0200 - a
ccuracy: 0.9989
Epoch 14/100
320/320 [==============================] - 67s 210ms/step - loss: 0.0172 - a
ccuracy: 0.9989
Epoch 15/100
320/320 [==============================] - 54s 168ms/step - loss: 0.0162 - a
ccuracy: 0.9984
Epoch 16/100
320/320 [==============================] - 54s 168ms/step - loss: 0.0144 - a
ccuracy: 0.9989
```

Out[69]:

```
<keras.callbacks.History at 0x7fe01b1a3310>
```

In [90]:

```
save_model(full_model, suffix="full-model")
```

```
Saving model to: /content/drive/MyDrive/ML/model/20220309-06421646808132-ful
l-model.h5...
```

Out[90]:

```
'/content/drive/MyDrive/ML/model/20220309-06421646808132-full-model.h5'
```

In [91]:

```
loaded_full_model = load_model('/content/drive/MyDrive/ML/model/20220309-06421646808132-ful
```

```
Loading saved model from: /content/drive/MyDrive/ML/model/20220309-064216468
08132-full-model.h5
```

Making predictions on test dataset

Since our model has been trained on images in the form of Tensor batches, to make predictions on the test data, we'll have to get it into the same format.

Luckily we created `create_data_batches()` earlier which can take a list of filenames as input and convert them into Tensor batches.

To make predictions on the test data, we'll:

- Get the test image filenames.
- Convert the filenames into test data batches using create_data_batches() and setting the test_data parameter to True (since there are no labels with the test images).
- Make a predictions array by passing the test data batches to the predict() function.

In [102]:

```
# Load test image filenames
test_path = "/content/drive/MyDrive/ML/test/"
test_filenames = [test_path + fname for fname in os.listdir(test_path)]
test_filenames[:10]
```

Out[102]:

```
['/content/drive/MyDrive/ML/test/e2a9a7580a1424bc6531b2b7375338db.jpg',
 '/content/drive/MyDrive/ML/test/e5de4eec61d00ee4834ff0153f90ed41.jpg',
 '/content/drive/MyDrive/ML/test/e64b15ca154304104fe95ded7338858e.jpg',
 '/content/drive/MyDrive/ML/test/dd3c80cee38d165aaf48083f4a4a0071.jpg',
 '/content/drive/MyDrive/ML/test/df01edf92d38b334f78bd85460304801.jpg',
 '/content/drive/MyDrive/ML/test/e683ba5a138de0fbb7bb1523862b43f2.jpg',
 '/content/drive/MyDrive/ML/test/e743bea73da2c0dab99ccdbc697b1ac8.jpg',
 '/content/drive/MyDrive/ML/test/e2628b6bde028b5eb593616128728907.jpg',
 '/content/drive/MyDrive/ML/test/e265af5e8f446888c6e7ec31f803d63e.jpg',
 '/content/drive/MyDrive/ML/test/dd703c7beeaf5cba5533d5f42b608f2e.jpg']
```

In [103]:

```
len(test_filenames)
```

Out[103]:

```
10357
```

In [104]:

```
test_data = create_data_batches(test_filenames, test_data=True)
```

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

In [105]:

```
test_data
```

Out[105]:

```
<BatchDataset element_spec=TensorSpec(shape=(None, 224, 224, 3), dtype=tf.fl
oat32, name=None)>
```

In [106]:

```
test_predictions = loaded_full_model.predict(test_data,
                                              verbose=1)
```

```
324/324 [==============================] - 1312s 4s/step
```

In [107]:

```
# Save predictions (Numpy array) to a csv
np.savetxt("/content/drive/MyDrive/ML/preds_array.csv", test_predictions, delimiter=",")
```

In [110]:

```
# Load predictions from a saved csv
test_predictions = np.loadtxt("/content/drive/MyDrive/ML/preds_array.csv", delimiter=",")
```

In [111]:

```
test_predictions[:10]
```

Out[111]:

```
array([[1.08898703e-07, 8.13571660e-07, 2.32910111e-06, ...,
        6.83813050e-06, 1.00849311e-05, 3.20487243e-06],
       [1.28516590e-11, 2.04853364e-03, 6.60769706e-09, ...,
        4.77530193e-06, 1.36379896e-09, 6.51470941e-07],
       [3.99432043e-09, 4.38627236e-14, 2.22435759e-09, ...,
        1.58351099e-07, 1.09960530e-08, 4.82859093e-07],
       ...,
       [1.09537304e-08, 1.60778786e-08, 4.87111329e-08, ...,
        1.21411300e-04, 9.36579170e-07, 1.29548353e-05],
       [8.70448886e-08, 1.33869491e-06, 1.67285696e-09, ...,
        4.66091912e-08, 2.43241841e-04, 9.94002676e-07],
       [3.41673273e-10, 3.84930132e-07, 9.56923984e-10, ...,
        1.97258814e-08, 5.11051531e-12, 2.82903423e-10]])
```

In [113]:

```
test_predictions.shape
```

Out[113]:

```
(10357, 120)
```

Preparing test dataset predictions for Kaggle

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

- Create a pandas DataFrame with an ID column as well as a column for each dog breed.
- Add data to the ID column by extracting the test image ID's from their filepaths.
- Add data (the prediction probabilities) to each of the dog breed columns using the unique_breeds list and the test_predictions list.
- Export the DataFrame as a CSV to submit it to Kaggle.

In [115]:

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

Out[115]:

id	affenpinscher	afghan_hound	african_hunting_dog	airedale	american_staffordshire_terrier

0 rows × 121 columns

In [119]:

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

Out[119]:

	id	affenpinscher	afghan_hound	african_hunting_dog	aired
0	e2a9a7580a1424bc6531b2b7375338db	NaN	NaN	NaN	N
1	e5de4eec61d00ee4834ff0153f90ed41	NaN	NaN	NaN	N
2	e64b15ca154304104fe95ded7338858e	NaN	NaN	NaN	N
3	dd3c80cee38d165aaf48083f4a4a0071	NaN	NaN	NaN	N
4	df01edf92d38b334f78bd85460304801	NaN	NaN	NaN	N

5 rows × 121 columns

In [120]:

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

Out[120]:

	id	affenpinscher	afghan_hound	african_hunting_dog	air
0	e2a9a7580a1424bc6531b2b7375338db	1.088987e-07	8.135717e-07	2.329101e-06	5.277
1	e5de4eec61d00ee4834ff0153f90ed41	1.285166e-11	2.048534e-03	6.607697e-09	3.816
2	e64b15ca154304104fe95ded7338858e	3.994320e-09	4.386272e-14	2.224358e-09	2.036
3	dd3c80cee38d165aaf48083f4a4a0071	9.385129e-06	6.233477e-10	1.370689e-11	1.844
4	df01edf92d38b334f78bd85460304801	1.847914e-05	8.686377e-06	8.260436e-06	5.756

5 rows × 121 columns

In [121]:

```
preds_df.to_csv("/content/drive/MyDrive/ML/full_submission_1_mobilienetV2.csv",
                 index=False)
```

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

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

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

To do so, we'll:

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

In [ ]:

```
# Get custom image filepaths
custom_path = "drive/My Drive/Data/dogs/"
custom_image_paths = [custom_path + fname for fname in os.listdir(custom_path)]
# Turn custom image into batch (set to test data because there are no labels)
custom_data = create_data_batches(custom_image_paths, test_data=True)
Creating test data batches...
# Make predictions on the custom data
custom_preds = loaded_full_model.predict(custom_data)
Now we've got some predictions arrays, let's convert them to labels and compare them with e

# Get custom image prediction labels
custom_pred_labels = [get_pred_label(custom_preds[i]) for i in range(len(custom_preds))]
custom_pred_labels
['golden_retriever', 'labrador_retriever', 'lakeland_terrier']
# Get custom images (our unbatchify() function won't work since there aren't labels)
custom_images = []
# Loop through unbatched data
for image in custom_data.unbatch().as_numpy_iterator():
  custom_images.append(image)
# Check custom image predictions
plt.figure(figsize=(10, 10))
for i, image in enumerate(custom_images):
  plt.subplot(1, 3, i+1)
  plt.xticks([])
  plt.yticks([])
  plt.title(custom_pred_labels[i])
  plt.imshow(image)
```

What's next?

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

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

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

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

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

"How to improve a TensorFlow 2.x image classification model?" "TensorFlow 2.x image classification best practices" "Transfer learning for image classification with TensorFlow 2.x" And when you see an example you think might be beyond your reach (because it looks too complicated), remember, if in doubt, run the code. Try and reproduce what you see. This is the best way to get hands-on and build your own knowledge.

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