DEEP LEARNING & REINFORCEMENT LEARNING:

COURSE PROJECT:

Algorithms and Techniques for Dog Identification

IBM / COURSERA

IDDI ABDUL AZIZ

Introduction

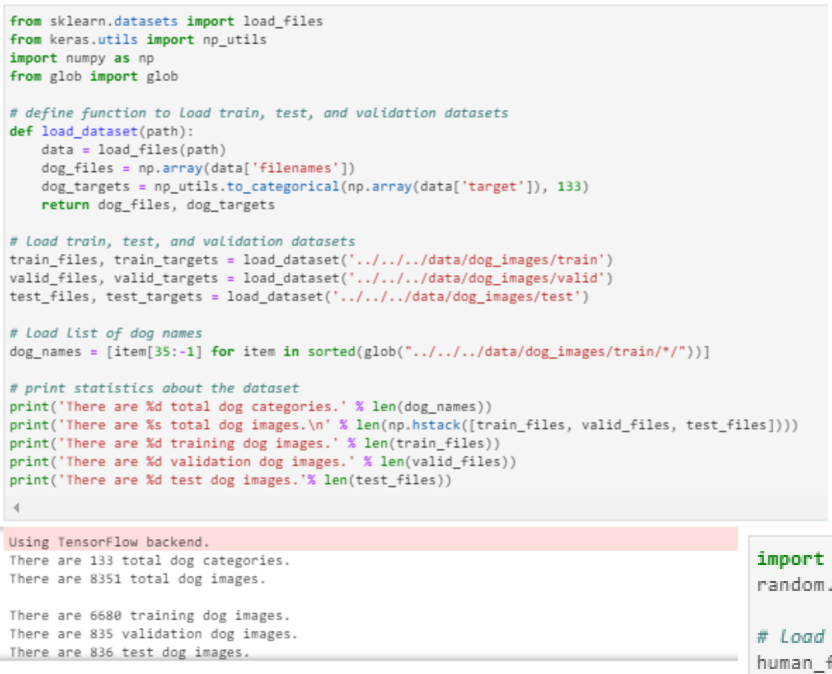
The objective of this project is to use deep learning to classify images of dogs according to their breed, developing an algorithm that could be used as part of a mobile or web app. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling.

Import Datasets

Import Dog and Human Datasets

I used the *load_files* function from scikit-learn library to import a dataset of dog images:

- train_files, valid_files, test_files - numpy arrays containing file paths to images
- train_targets, valid_targets, test_targets - numpy arrays containing onehot-encoded classification labels
- dog_names - list of string-valued dog breed names for translating labels



```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob

# define function to load train, test, and validation datasets
def load_dataset(path):
    data = load_files(path)
    dog_files = np.array(data['filenames'])
    dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
    return dog_files, dog_targets

# load train, test, and validation datasets
train_files, train_targets = load_dataset('../../../data/dog_images/train')
valid_files, valid_targets = load_dataset('../../../data/dog_images/valid')
test_files, test_targets = load_dataset('../../../data/dog_images/test')

# load list of dog names
dog_names = [item[35:-1] for item in sorted(glob("../../../data/dog_images/train/*/"))]

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_files, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

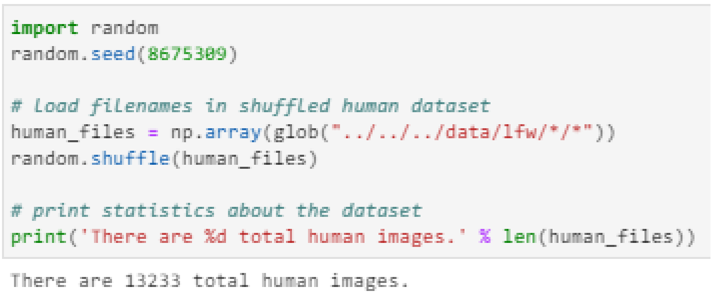
```
Using TensorFlow backend.
There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

From the code at left, we can see that:

1. There are 113 total dog categories.
2. There are 8351 total dog images.
3. There are 6680 training dog images.
4. There are 835 validation dog images.
5. There are 836 test dog images.

Import Human Dataset



```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("../../../data/lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

```
There are 13233 total human images.
```

From the code cell, I imported a dataset of human images, where the files paths are stored in the numpy array *human_files*.

From the code, we can see that there are 13,233 total human images.

Detect Humans

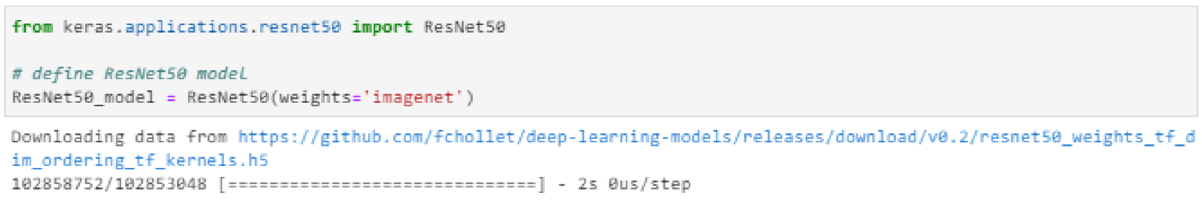
I use the OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. Before using any of the face detectors, I converted the images to grayscale, as that's the standard procedure. I then use *detectMultiScale* function to executes the classifier stored in *face_cascade* and takes the grayscale image as a parameter.

The face detector finds human faces in 11% of the first 100 dog photos; that's meaning identified 11 images as containing faces (instead of dog faces) in it. The face detector finds human faces in 100%

of the first 100 human photos; that's meaning the algorithm '*Face_detector*' work done on human faces.

Detect Dogs

In this section, I used a pre-trained ResNet-50 model to detect dogs in images. I first downloaded the ResNet-50 model, along with weights that have been trained on ImageNet. ImageNet is a very large, very popular dataset used for image classification and other vision tasks, it contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.



```
from keras.applications.resnet50 import ResNet50

# define ResNet50 model
ResNet50_model = ResNet50(weights='imagenet')
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/resnet50_weights_tf_d
im_ordering_tf_kernels.h5
102858752/102853048 [==============================] - 2s 0us/step
```

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which is also referred as a 4D tensor) as input, with shape

(nb_samples, rows, columns, channels),

where nb_samples correspond to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively.

The path_to_tensor function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

(1, 224 ,224, 3).

The paths_to_tensor function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

(nb_samples, 224, 224, 3).

Here, nb_samples is the number of samples, or number of images, in the supplied array of image paths. It is best to think of nb_samples as the number of 3D tensors (where each 3D tensor corresponds to a different image) in the dataset!

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as [103.939,116.779,123.68] and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function preprocess_input. The pre-process input can access here here.

Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the predict method, which returns an array whose i-th entry is the model's predicted probability that the image belongs to the i-th ImageNet category. This is implemented in the ResNet50_predict_labels function below. By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary.

```
from keras.applications.resnet50 import preprocess_input, decode_predictions

def ResNet50_predict_labels(img_path):
    # returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))
```

Write a Dog Detector

While looking at the dictionary, I noticed that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from 'Chihuahua' to 'Mexican hairless'. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, I need only check if the ResNet50_predict_labels function above returns a value between 151 and 268 (inclusive).

I use these ideas to complete the dog_detector function below, which returns True if a dog is detected in an image (and False if not).

```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

After running the code;

1. The dog detector finds human faces in 0% of the human photo.
2. The dog detector finds human faces in 100% of the human photos

   That's the algorithm *'dog_detector'* work done on both human faces and dogs.

```
human_faces = [dog_detector(human_file) for human_file in human_files_short]
percentage_human = 100 * (np.sum(human_faces)/len(human_faces))

dog_faces = [dog_detector(dog_file) for dog_file in dog_files_short]
percentage_dog = 100 * (np.sum(dog_faces)/len(dog_faces))

print('There are %.1f%% images of the first 100 human_files that have a detected dog face.' % percentage_human)
print('There are %.1f%% images of the first 100 dog_files that have a detected dog face.' % percentage_dog)
```

```
There are 0.0% images of the first 100 human_files that have a detected dog face.
There are 100.0% images of the first 100 dog_files that have a detected dog face.
```

Creating a CNN to Classify Dog Breeds

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, I will create a CNN that classifies dog breeds.

Below are high-level steps to build CNN to classify images:

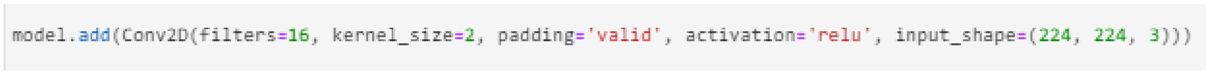
1. Create convolutional layers by applying kernel or feature maps
2. Apply Max pool for translational invariance
3. Flatten the inputs
4. Create a Fully connected neural network
5. Train the model
6. Predict the output

Firstly, I initialise the neural network for building CNN.

```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()
#The model type that we will be using is Sequential.
#Sequential is the easiest way to build a model in Keras.
#It allows you to build a model layer by layer.
```

This model type that we I be using is Sequential. Sequential is the easiest way t build a model in Keras. It allows you to build a model layer by layer.



```
model.add(Conv2D(filters=16, kernel_size=2, padding='valid', activation='relu', input_shape=(224, 224, 3)))
```

I used the *'add()'* function to add layers to our model. My first layer is Conv2D layers, convolution layers that will deal with our input images, which are seen as 2-dimensional matrices.

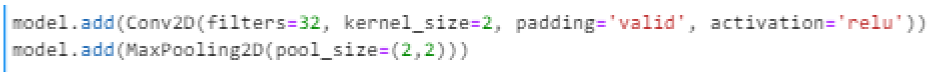
16 in the first layer is the number of nodes, I want to feature maps. Kernel size is the filter matrix for our convolution. That be, a kernel size of 2 means will have a 2x2 filter matrix or feature detector.

Padding is used on the convolutional layers to ensure the height and width of the output feature maps matches the inputs.

The activation function I used for the layers is ReLU Rectifier Linear Unit which help with non-linearity in the neural network. The first layer also takes in an input shape, this is the shape of each input image; 224, 224, 3.

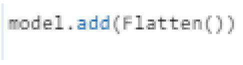
I applied max pooling for translational invariance. Translational invariance is when we change the input by a small amount, the outputs do not change. Max pooling reduces the number of cells.

Pooling helps detect features like colours, edges etc. For max pooling, we use the pool_size of 2 matrix for all 32 feature maps.



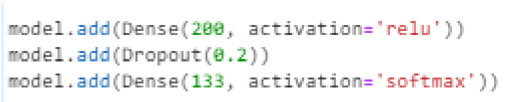
```
model.add(Conv2D(filters=32, kernel_size=2, padding='valid', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
```

Following by two layers is Conv2D layers, their number of filters 32,64 is the number of nodes, ith the kernel of (2,2). When then apply the max pooling to the convolutional layers.



```
model.add(Flatten())
```

When done with 'Flatten' layer. Flatten serves as connection between the convolutional and dense layers, step is to flatten all the inputs. The flattened data will be input to the fully connected neural network.

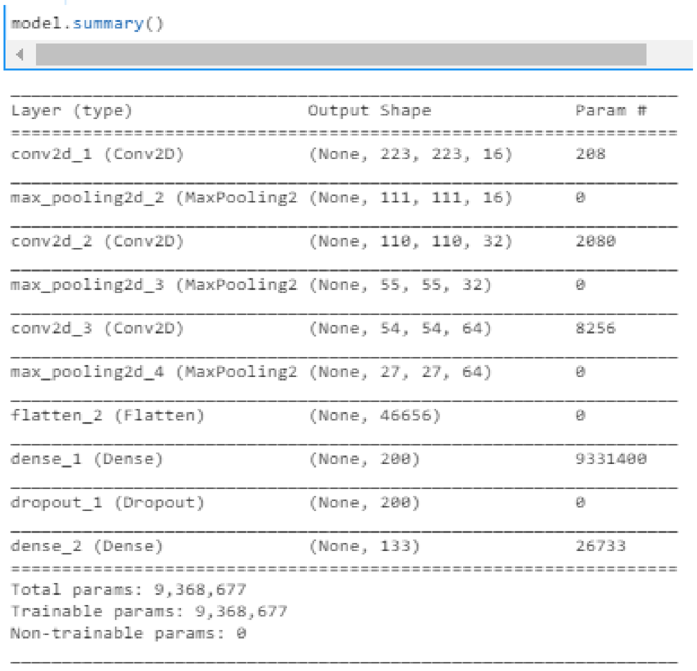


```
model.add(Dense(200, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(133, activation='softmax'))
```

Dense is the layer type I used in for the input layer. Dense is a standard layer type that is used in many cases for neural networks.

I used Dropout rate of 20% to prevent overfitting.

The activation is 'SoftMax'. SoftMax makes the output sum up to 1 so the output can be interpreted as probabilities. The model will then make its prediction based on which option has the highest probability.



```
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208       
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 111, 111, 16)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080      
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 55, 55, 32)        0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256      
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 27, 27, 64)        0         
_________________________________________________________________
flatten_2 (Flatten)          (None, 46656)             0         
_________________________________________________________________
dense_1 (Dense)              (None, 200)               9331400   
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               26733     
=================================================================
Total params: 9,368,677
Trainable params: 9,368,677
Non-trainable params: 0
_________________________________________________________________
```

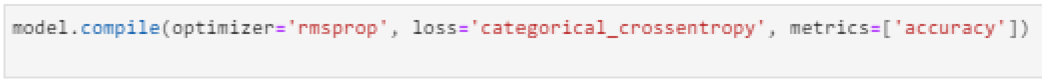
From the code, we realised;

Total params: 9,368.677

Trainable params: 9,268.677

Non-trainable params: 0

Now I need compile the model. Compiling the model takes three parameters: Optimizer, loss and metrics.



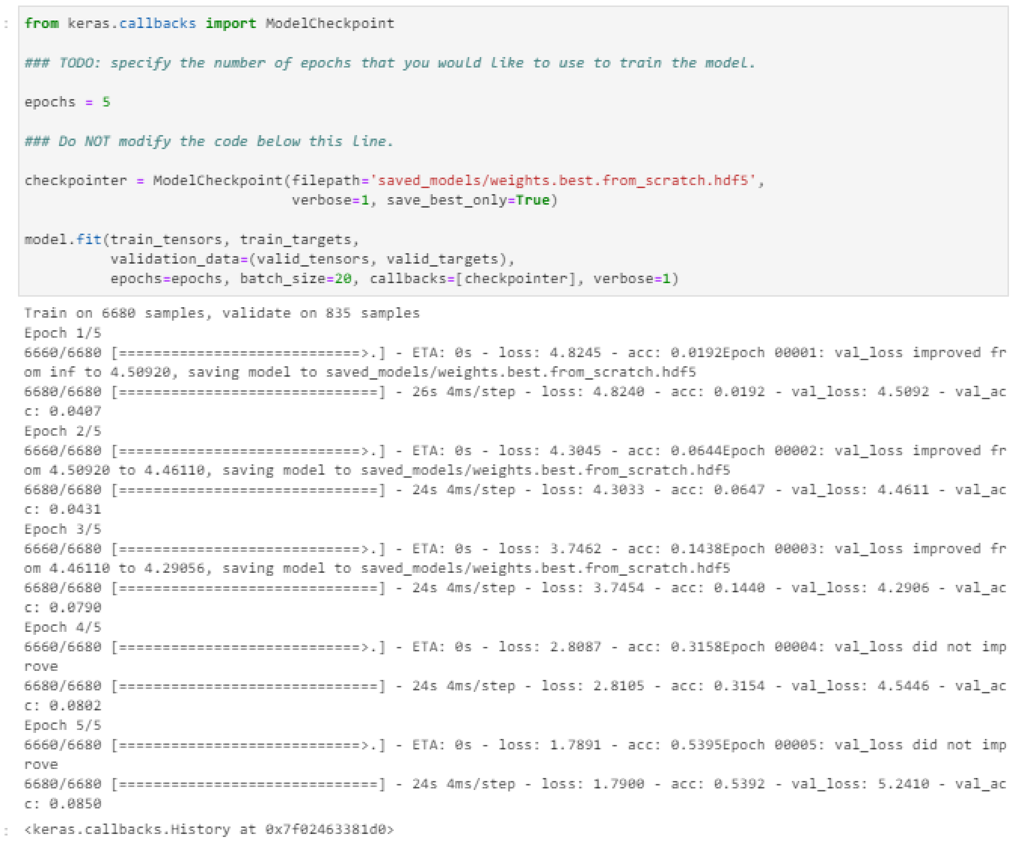
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])
```

The optimizer controls the learning rate. We will be using ‘rmsprop’ as our optimizer. The learning rate determines how fast the optimal weights for the model are calculated. A smaller learning rate may lead to more accurate weights (up to a certain point), but the time it takes to compute the weights will be longer.

We will use ‘categorical_crossentropy’ for **our loss** function. This is the most common choice for classification. A lower score indicates that the model is performing better.

To make things even easier to interpret, we will use the ‘accuracy’ **metric** to see the accuracy score on the validation set when we train the model.

When trained for 5 epochs, the model achieves a test accuracy of 6.1005%.



```
from keras.callbacks import ModelCheckpoint

### TODO: specify the number of epochs that you would like to use to train the model.

epochs = 5

### Do NOT modify the code below this line.

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.hdf5',
                               verbose=1, save_best_only=True)

model.fit(train_tensors, train_targets,
          validation_data=(valid_tensors, valid_targets),
          epochs=epochs, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8245 - acc: 0.0192Epoch 00001: val_loss improved fr
om inf to 4.50920, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 26s 4ms/step - loss: 4.8240 - acc: 0.0192 - val_loss: 4.5092 - val_ac
c: 0.0407
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.3045 - acc: 0.0644Epoch 00002: val_loss improved fr
om 4.50920 to 4.46110, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 24s 4ms/step - loss: 4.3033 - acc: 0.0647 - val_loss: 4.4611 - val_ac
c: 0.0431
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 3.7462 - acc: 0.1438Epoch 00003: val_loss improved fr
om 4.46110 to 4.29056, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 24s 4ms/step - loss: 3.7454 - acc: 0.1440 - val_loss: 4.2906 - val_ac
c: 0.0790
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 2.8087 - acc: 0.3158Epoch 00004: val_loss did not imp
rove
6680/6680 [==============================] - 24s 4ms/step - loss: 2.8105 - acc: 0.3154 - val_loss: 4.5446 - val_ac
c: 0.0802
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 1.7891 - acc: 0.5395Epoch 00005: val_loss did not imp
rove
6680/6680 [==============================] - 24s 4ms/step - loss: 1.7900 - acc: 0.5392 - val_loss: 5.2410 - val_ac
c: 0.0850

<keras.callbacks.History at 0x7f02463381d0>
```

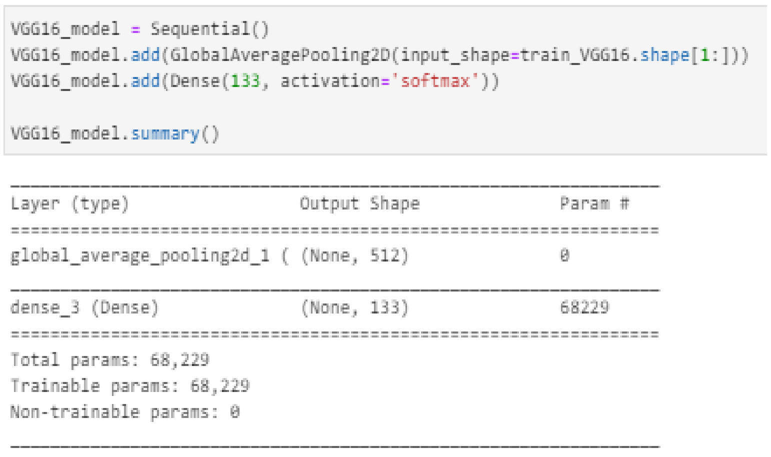
I had want to add more epochs in the training process but here in this situation we can’t add more cause more parameters means longer training**.** One Epoch is when an ENTIRE dataset is passed forward and backward through the neural network only ONCE.
Why I use more than one Epoch? I know it doesn’t make sense in the starting that — passing the entire dataset through a neural network is not enough. And we need to pass the full dataset multiple times to the same neural network. But keep in mind that we are using a limited dataset and to optimise the learning and the graph we are using Gradient Descent which is an iterative process. So, updating the weights with single pass or one epoch is not enough.
One epoch leads to underfitting of the curve.

As the number of epochs increases, more number of times the weight are changed in the neural network and the curve goes from underfitting to optimal to overfitting curve

So, what is the right numbers of epochs? Unfortunately, there is no right answer to this question. The answer is different for different datasets but you can say that the numbers of epochs is related to how diverse your data is… just an example - Do you have only black cats in your dataset or is it much more diverse dataset?

Use a CNN to Classify Dog Breeds (Using Transfer Learning)

I use a CNN to Classify Dog Breeds from pre-trained VGG-16 model, using VGG-16 bottleneck features. With test accuracy: 40.9091%.



```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_1 ( (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

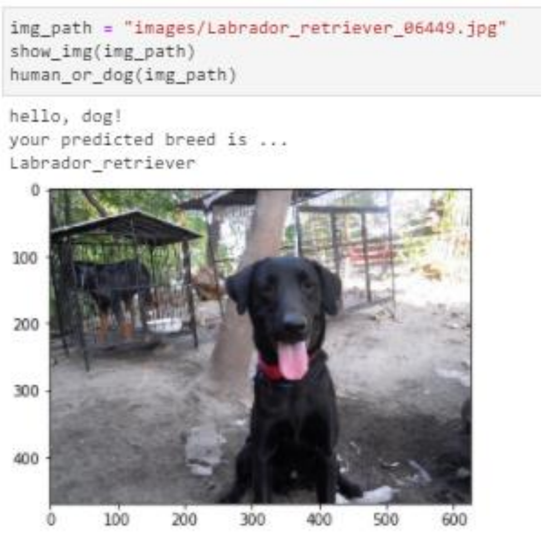
The model uses the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model.

I only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with SoftMax.

Results

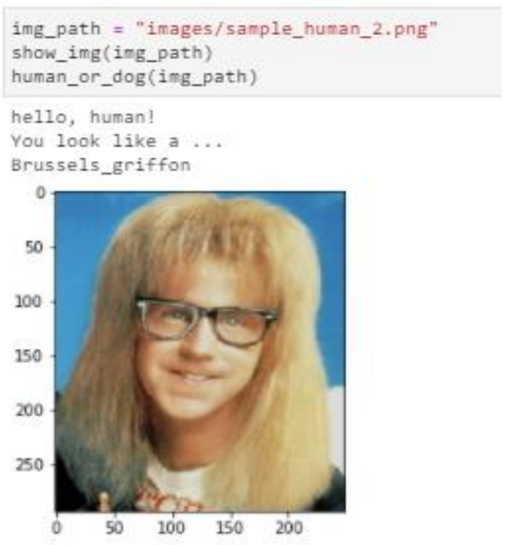
I used the ‘transfer learning – ResNet50 model’ to implement an algorithm for a Dog identification application. The user provides an image, and the algorithm first detects whether the image is human or dog. If it is a dog, it predicts the breed. If it is a human, it returns the resembling dog breed. Where the model produced the test accuracy of around 80%.

Here are examples of the algorithms:



```
img_path = "images/Labrador_retriever_06449.jpg"
show_img(img_path)
human_or_dog(img_path)
```

```
hello, dog!
your predicted breed is ...
Labrador_retriever
```



```
img_path = "images/sample_human_2.png"
show_img(img_path)
human_or_dog(img_path)
```

```
hello, human!
You look like a ...
Brussels_griffon
```