dog_app_final

January 23, 2021

1 Project: Write an Algorithm for a Dog Identification

1.0.1 Why We're Here

In this notebook, we will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, the code will accept any user-supplied image as input. 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. The image below displays potential sample output of the finished project

1.0.2 The Road Ahead

We break the notebook into separate steps

- Section **??**: Import Datasets
- Section **??**: Detect Humans
- Section **??**: Detect Dogs
- Section **??**: Create a CNN to Classify Dog Breeds (from Scratch)
- Section **??**: Use a CNN to Classify Dog Breeds (using Transfer Learning)
- Section **??**: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Section **??**: Write your Algorithm
- Section **??**: Test Your Algorithm

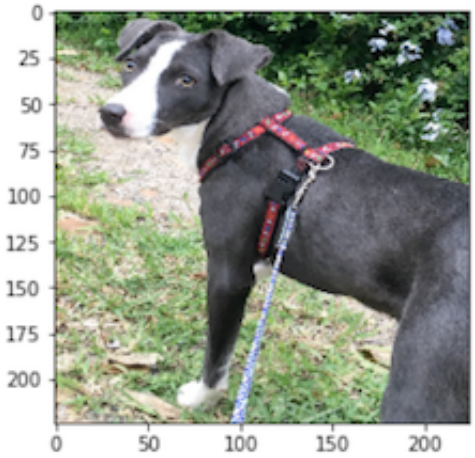
1.1 Step 0: Import Datasets

1.1.1 Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the `load_files` function from the scikit-learn library: - `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

```
In [47]: from sklearn.datasets import load_files
         from keras.utils import np_utils
         import numpy as np
         from glob import glob
         import os
```

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



Sample Dog Output

```
import matplotlib.pyplot as plt
%matplotlib inline

# define function to load train, test, and validation datasets
def load_dataset(path):

    '''
    Function to load the dataset using the path as input

    Inputs:
    path: path to the data set which is to be loaded
    '''

    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[20:-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, tes
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))
```

```
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.
```

1.1.2 Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array `human_files`.

```
In [48]: 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.
```

1.2 Step 1: Detect Humans

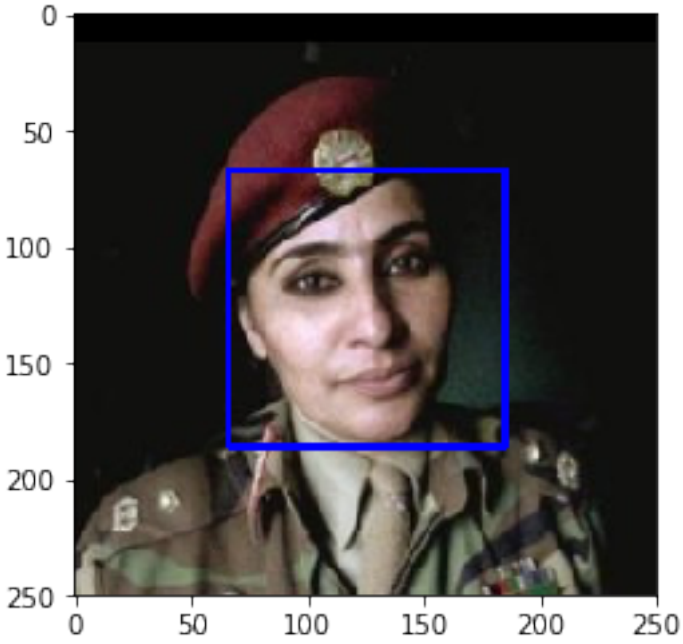
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [49]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

         # extract pre-trained face detector
         face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

         # load color (BGR) image
         img = cv2.imread(human_files[3])
         # convert BGR image to grayscale
```

```
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # find faces in image
    faces = face_cascade.detectMultiScale(gray)

    # print number of faces detected in the image
    print('Number of faces detected:', len(faces))

    # get bounding box for each detected face
    for (x,y,w,h) in faces:
        # add bounding box to color image
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # display the image, along with bounding box
    plt.imshow(cv_rgb)
    plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.2.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [50]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):

             '''
             Function to detect whether the image (located at img_path) has a human face.

             Inputs:
             img_path: path to the imaged

             Output: True if a human face is present, False if not present
             '''


             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.2.2 Assess the Human Face Detector

The code cell below test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. We will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`

```
In [51]: human_files_short = human_files[:100]
         dog_files_short = train_files[:100]

         # initialize no of humans and dogs detected to nil each
         humans_detected = 0
         dogs_detected = 0

         # get no of total files (100 for both the lists)
         total_files = len(human_files_short)
```