dog_app

May 23, 2019

1 Data Scientist Nanodegree

1.1 Convolutional Neural Networks

1.2 Project: Write an Algorithm for a Dog Identification App

This notebook walks you through one of the most popular Udacity projects across machine learning and artifical intellegence nanodegree programs. The goal is to classify images of dogs according to their breed.

If you are looking for a more guided capstone project related to deep learning and convolutional neural networks, this might be just it. Notice that even if you follow the notebook to creating your classifier, you must still create a blog post or deploy an application to fulfill the requirements of the capstone project.

Also notice, you may be able to use only parts of this notebook (for example certain coding portions or the data) without completing all parts and still meet all requirements of the capstone project.

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTA-TION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this IPython notebook.
Step 0: Import Datasets

1.2.1 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

```python
In [13]: 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[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.


In [21]: # save dog names to disk
         import csv
         csvfile='/saved_models/dog_names.csv'

         filename='saved_models/dog_names.csv'
         with open(filename, "w") as outfile:
```

```
        for entries in dog_names:
            outfile.write(entries)
            outfile.write("\n")
```

1.2.2   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 [13]: 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.
```

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 [14]: 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))
```
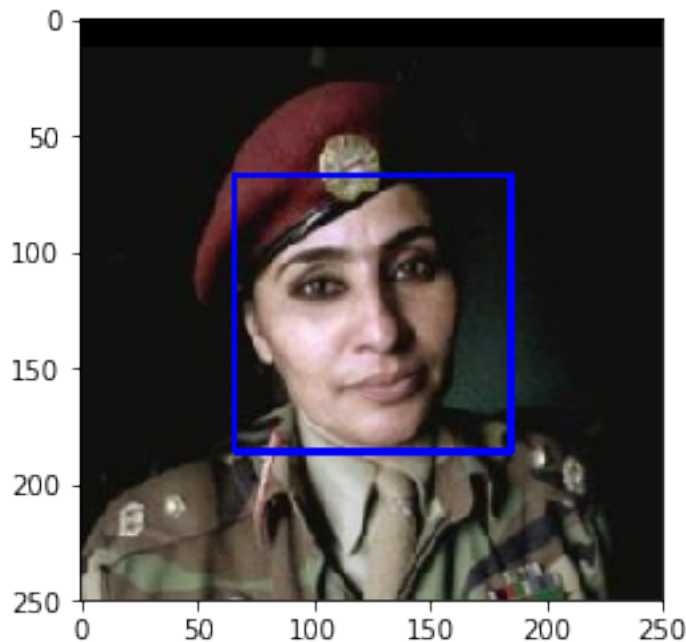
```
        # get bounding box for each detected face
        for (x,y,w,h) in faces:
            # add bounding box to color image
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

        # 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.

1.2.3 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 [15]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.2.4 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to 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. You 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`.
    Answer:

```
In [16]: human_files_short = human_files[:100]
         dog_files_short = train_files[:100]
         # Do NOT modify the code above this line.

         ## TODO: Test the performance of the face_detector algorithm
         ## on the images in human_files_short and dog_files_short.
```

```
In [17]: def assess_classifier(classifier, human_data=human_files_short, dog_data=dog_files_shor
             """
             Assesses the performance of a dog/human classifier

             Input:
             classifier...classifier to be tested
             human_data...input data
             dog_data...input data

             Output:
             out_mat...matrix containing result
             """
             # initialize output matrix with zeros
             # col 1: results for humans
             # col 2: results for dogs
             out_mat=np.zeros((len(human_data),2))
             for i in range(0,len(human_data)):
                 # humans
                 out_mat[i,0]=classifier(human_data[i])
                 #print(i)
                 #print(out_mat[i,0])
                 # dogs
```