dog_app

December 9, 2021

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 artificial 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 **'(IMPLEMENTATION)'** 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

In [1]:
```
########1########2########3########4########5########6########7########8
#import generic libraries
import cv2
import matplotlib.pyplot as plt
import numpy as np
import random
import seaborn as sns

from time import time
from extract_bottleneck_features import *

#sklearn
from sklearn.datasets import load_files

#keras
from keras.applications.resnet50 import ResNet50
from keras.applications.resnet50 import preprocess_input, decode_predictions
from keras.callbacks import ModelCheckpoint
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras.preprocessing.image import load_img, img_to_array
from keras.preprocessing.image import ImageDataGenerator
from keras.utils.np_utils import to_categorical

#glob
from glob import glob

#tqm
from tqdm import tqdm

#PIL
from PIL import ImageFile

#setting the random seed
random.seed(8675309)

#using matplotlib inline
%matplotlib inline
```

```
Using TensorFlow backend.
```

In [2]:
```
########1########2########3########4########5########6########7########8
# define function to load train, test, and validation datasets
def load_dataset(path):
    '''
    This function loads the dataset containing images and labels.

    Special note: this function is STRONGLY based on Udacity notebook for Dog
    Breed Classifications, for completing the Capstone Project for Data Scientist
    course. It may be used for educational purposes only!

    Inputs:
    - path (mandatory) - a string containing the path for loading the data

    Outputs:
    - dog_files
    - dog_targets
    '''
    data = load_files(path)

    dog_files = np.array(data['filenames'])
    dog_targets = to_categorical(np.array(data['target']), 133)

    return dog_files, dog_targets
```

In [3]:
```
########1########2########3########4########5########6########7########8
start = time()

# 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/*/"))]

end = time()
print('{:.4f}seconds to process'.format(end-start))

total_dog = len(dog_names)
total_image = len(np.hstack([train_files, valid_files, test_files]))
train_image = len(train_files)
valid_image = len(valid_files)
test_image = len(test_files)

# print statistics about the dataset
```

```
        print('There are {} total dog categories'.format(total_dog))
        print('There are {} total dog images'.format(total_image))
        print('-----------------')
        print('There are {} training ({:.1f}%) of dog images'\
              .format(train_image, 100 * (train_image/total_image)))
        print('There are {} validation ({:.1f}%) of dog images'\
              .format(valid_image, 100 * (valid_image/total_image)))
        print('There are {} test ({:.1f}%) of dog images'\
              .format(test_image, 100 * (test_image/total_image)))
```

```
9.4681seconds to process
There are 133 total dog categories
There are 8351 total dog images
-----------------
There are 6680 training (80.0%) of dog images
There are 835 validation (10.0%) of dog images
There are 836 test (10.0%) of dog images
```

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 [4]: ########1########2########3########4########5########6########7########8
        # 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 {} total human images'.format(len(human_files)))
```

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

Step 1: Detect Humans

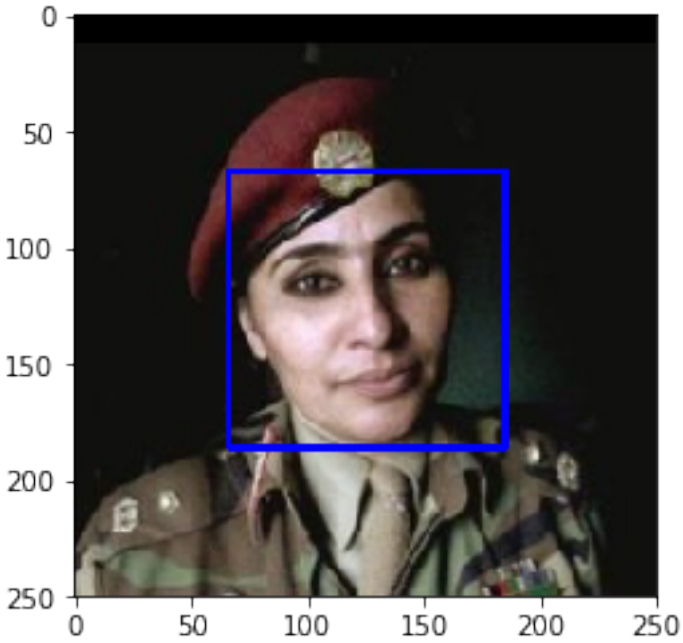
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

```
In [5]: ########1########2########3########4########5########6########7########8
        # 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
```




```
In [6]: print(img.shape)
```

```
(250, 250, 3)
```

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.

---

cv2 is **Open Source Computer Vision** Project and can be find here
cv2.imread method at GeeksforGeeks

```
In [7]: ########1########2########3########4########5########6########7########8
        def face_detector(img_path,
                          verbose=False):
            '''
            This function takes an image path and returns a True, if some face could be
            recognized.

            Special note: this function is STRONGLY based on Udacity notebook for Dog
            Breed Classifications, for completing the Capstone Project for Data Scientist
            course. It may be used for educational purposes only!

            Inputs:
            - img_path (mandatory) - (Text String)
            - verbose (optional) - if you want some verbosity under processing
              (default=False)

            Output:
            - True, is a face was recognized in the image (Boolean)
            '''
            if verbose:
                print('###function face detector started')

            start = time()
            classifier='haarcascades/haarcascade_frontalface_alt.xml'

            #you take an already trained face detector that is taken from a path
```

```
    face_cascade = cv2.CascadeClassifier(classifier)

    #originally it is a RGB color image
    img = cv2.imread(img_path)

    if verbose:
        print('*image:', img_path)

    #as we seen in the class, normally human faces were converted to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #run face detector method - for grayscale
    faces = face_cascade.detectMultiScale(gray)

    #Test function for faces
    if_face = len(faces) > 0
    num_faces = len(faces)

    #check if it is OK
    if verbose:
        print('*number of faces detected:{}, returning {}'.format(num_faces, if_face))

    end = time()

    if verbose:
        print('processing time: {:.4}s'.format(end-start))

    return if_face, num_faces
```

```
In [8]: test_face, number = face_detector(
                                img_path=human_files[3],
                                verbose=True)
```

```
###function face detector started
*image: ../../../data/lfw/Khatol_Mohammad_Zai/Khatol_Mohammad_Zai_0001.jpg
*number of faces detected:1, returning True
processing time: 0.0439s
```

1.2.4 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 [9]: human_files_short = human_files[:100]
        dog_files_short = train_files[:100]
        # Do NOT modify the code above this line.
        print('first human:', human_files_short[0])
        print('first dog:', dog_files_short[0])
```

```
first human: ../../../data/lfw/Rick_Dinse/Rick_Dinse_0002.jpg
first dog: ../../../data/dog_images/train/095.Kuvasz/Kuvasz_06442.jpg
```

Why in an Error, it prints this hellish *oh, no, Captain Goodnight!*?

- its only for a bit of **humor**, there was a game for Apple II named **Captain Goodnight and the Islands of Fear**, wich is now an Abandonware. There was a lot of humor in this game and it provided a lot of fun for me and my friends at my younger age.
- references of the game are at wiki, and when something really **weird** happened with your character, this phrase appeard just before you die!

```
In [10]: verbose=False

         human_errs = []

         for human in human_files_short:
             test_face, num_faces = face_detector(
                                        img_path=human,
                                        verbose=False)

             if test_face and num_faces == 1:
                 #print('*all right!')
                 pass
             else:
                 print('*oh, no, Captain Goodnight!')
                 human_errs.append((human, test_face, num_faces))

         for facerr in human_errs:
             print('{} is a human and was {}, recognized ({} faces)'.format(facerr[0], facerr[1]

         print('total potential mistakes: {}%'.format(len(human_errs)/len(human_files_short) * 1
```

```
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
../../../data/lfw/Padraig_Harrington/Padraig_Harrington_0002.jpg is a human and was True, recogn
../../../data/lfw/David_Ho/David_Ho_0001.jpg is a human and was True, recognized (2 faces)
../../../data/lfw/Dalai_Lama/Dalai_Lama_0002.jpg is a human and was True, recognized (2 faces)
total potential mistakes: 3.0%
```

Answer to **Question 1a**:

- well, the number is: **100%** correct, so "everything ran well, and all the humans were correctly interpreted as humans"
- so also we can say: "no **false negatives** (Type II Error) occurred"
- potential troubles could happen with some pictures, that occurred **2 faces** found
- perhaps we need to **open** these pics as see if there is really more than 1 face in each picture
- another potential trouble is that we are testing only a **small part** of our data. So perhaps our **sample** is too small...
- and finally, it also coud occur some **false positives** (Type I Error) with this data. So perhaps a dragon, or a Vulcan was falsely detected as a **Human** (or, just extrapolating, a **clock** identified as a human face (and a **true** human face not recognized in the same picture!).

In [11]:
```
verbose=False

dog_errs = []

for dog in dog_files_short:
    test_face, num_faces = face_detector(
                                img_path=dog,
                                verbose=False)

    if test_face: #test for false positives
        print('*oh, no, Captain Goodnight!')
        dog_errs.append((dog, test_face, num_faces))

for facerr in dog_errs:
    print('{} is a dog and was {} intepreted as a human ({} faces)'.format(facerr[0], f

print('total potential mistakes: {}%'.format(len(dog_errs)/len(dog_files_short) * 100))
```

```
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
*oh, no, Captain Goodnight!
../../../data/dog_images/train/095.Kuvasz/Kuvasz_06442.jpg is a dog and was True intepreted as a
../../../data/dog_images/train/099.Lhasa_apso/Lhasa_apso_06646.jpg is a dog and was True intepre
../../../data/dog_images/train/009.American_water_spaniel/American_water_spaniel_00628.jpg is a
../../../data/dog_images/train/057.Dalmatian/Dalmatian_04023.jpg is a dog and was True intepretе
../../../data/dog_images/train/096.Labrador_retriever/Labrador_retriever_06474.jpg is a dog and
```

```
../../../data/dog_images/train/106.Newfoundland/Newfoundland_06989.jpg is a dog and was True int
../../../data/dog_images/train/117.Pekingese/Pekingese_07559.jpg is a dog and was True inteprete
../../../data/dog_images/train/039.Bull_terrier/Bull_terrier_02805.jpg is a dog and was True int
../../../data/dog_images/train/097.Lakeland_terrier/Lakeland_terrier_06516.jpg is a dog and was
../../../data/dog_images/train/024.Bichon_frise/Bichon_frise_01771.jpg is a dog and was True int
../../../data/dog_images/train/084.Icelandic_sheepdog/Icelandic_sheepdog_05705.jpg is a dog and
total potential mistakes: 11.0%
```

Answer to **Question 1b**:

- well, 11 dogs were incorrectly classified as humans in a sample of 100, so 11% of **error**
- that is a thing to worry about, because they are not just errors. They are Type I errors (**false positives**). And this is normally the **worse** type of error!
- so this Perceptron is not so relliable in terms of **false positives**

Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having unneccessarily frustrated users!). In your opinion, is this a reasonable expectation to pose on the user? If not, can you think of a way to detect humans in images that does not necessitate an image with a clearly presented face?

Answer:

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on each of the datasets.

In **Extra Classes** about Perceptrons, we learned at Lesson 3, Class 22/23 that is **not** necessary to be really alligned to be interpreted as True. And this is named **translation invariance**.

So, we could intensely modify the presence algorithm for better recognition of "bad" pictures.

Only on word, somethimes, just saying to the user to stay really **alligned** for the picture is a kind of an invitation to **not stay alligned**! So, people loves to try to break rules, or to cheat an AI (and this is basically the main theme of the movie "The Terminator" (1984).

So if, we could make something just to improve the Accuracy for translated people in a picture, it should be wonderful!

Back into the 22/23 classes, what we can do (and it is not so complicated) is to make **Image Augmentation**. Just take the original dataset and do some artificial translations and retrain it with this new augmented dataset.

Extra note: I am not quite sure that we really need to do an Image Augmentation for **translation** in this case. As we already have an algorithm to detect a **face window**, the remaining of the picture will not be interpreted by our Perceptron!

Source: Udacity Extra Classes, Perceptrons, Lesson 3, Class 22/23 (slightly modified, for fitting our new data parameters)

In [12]:
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.
```