# dog_app

June 24, 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 [2]: #Data wrangling and plotting
        import matplotlib.pyplot as plt
        import numpy as np
        import random
        import re
        import pandas as pd
        from sklearn.metrics import classification_report
        from sklearn.metrics import confusion_matrix

        #Dataset loading
        from sklearn.datasets import load_files

        #Image preprocessing
        import cv2
        from PIL import ImageFile

        #Find all pathnames matching a specified pattern
        from glob import glob

        #progress meter for loops
        from tqdm import tqdm

        #Keras Classes and functions for:
        # 1)input preprocessing
        # 2)building, training and evaluating Neural Networks and CNN performance
        # 3)loading state of the art pretrained models and architectures for transfer-learning
        from keras.applications.resnet50 import ResNet50,preprocess_input, decode_predictions
        from keras.callbacks import ModelCheckpoint
        from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D, AveragePooling2D
        from keras.layers import Dropout, Flatten, Dense
        from keras.models import Sequential
        from keras.preprocessing import image
        from keras.utils import np_utils

        #Availables functions for extracting bottlenech features of popular pretrained models
        from extract_bottleneck_features import *

        %matplotlib inline
```

```
Using TensorFlow backend.
```

```
In [3]: # 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, test
        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))
```

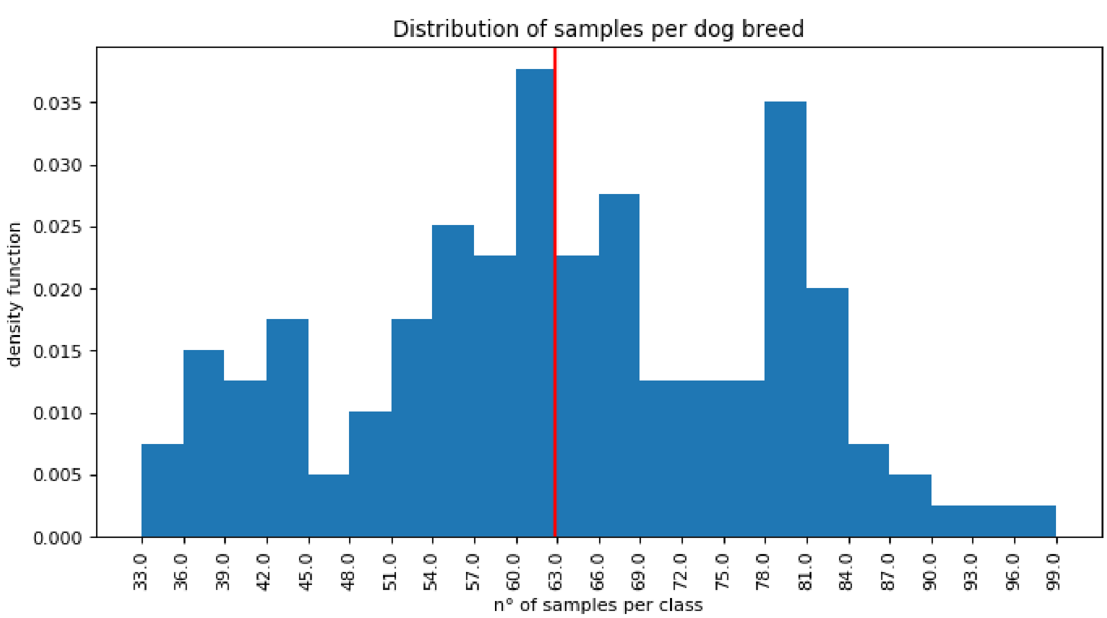
```
There are 133 total dog categories.
There are 8351 total dog images.

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

```
In [4]: #Dog breed names in clean format

        def breed_list(dog_names):
            '''
            Function that extracts and cleans the dog breeds names
            contained in list dog_names
            Input: dog names list from step 0
            Output: list with cleaned dog's breeds names
            '''
            reg=re.compile(r'\d{3}.*')
            breeds=[]

            for i in dog_names:
                breed=reg.search(i)
                raw=breed.group()[4:].replace("_"," ")
                breeds.append(raw)
            return breeds
```

```
breeds=breed_list(dog_names)
```

In [5]:
```
#Main statistics and number of samples by breed distribution

from matplotlib.pyplot import figure

figure(figsize=(10,5), dpi=80)

samples=np.vstack([train_targets, valid_targets, test_targets])

datos=pd.Series(samples.sum(axis=0))
datos.index=breeds
print("Avg. number of samples per breed: {}".format(np.around(datos.mean(),1),color='r')
print("Min. number of samples per breed: {}".format(datos.min()))
print("Max. number of samples per breed: {}".format(datos.max()))

bins = np.arange(datos.min(), datos.max()+6, 3)
plt.hist(datos,bins=bins,density=True)  # arguments are passed to np.histogram
plt.title("Distribution of samples per dog breed")
plt.xlabel("nř of samples per class")
plt.ylabel("density function")
plt.xticks(bins, np.around(bins,0), rotation='vertical')
plt.axvline(np.around(datos.mean(),1),linewidth=2, color='r')
plt.show()
```

```
Avg. number of samples per breed: 62.8
Min. number of samples per breed: 33.0
Max. number of samples per breed: 96.0
```



```
In [6]: #save data for posterior excel writing

        frame = { 'Dog_breed': datos.index, 'n_samples': datos.values }
        train_dc=pd.DataFrame(frame)
        train_dc=train_dc.sort_values(by=['n_samples'],ascending=False).reset_index(drop=True)
```

## 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 [7]: 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.
```

```
In [8]: human_files.size
```

```
Out[8]: 13233
```

---

## Step 1: Detect Humans

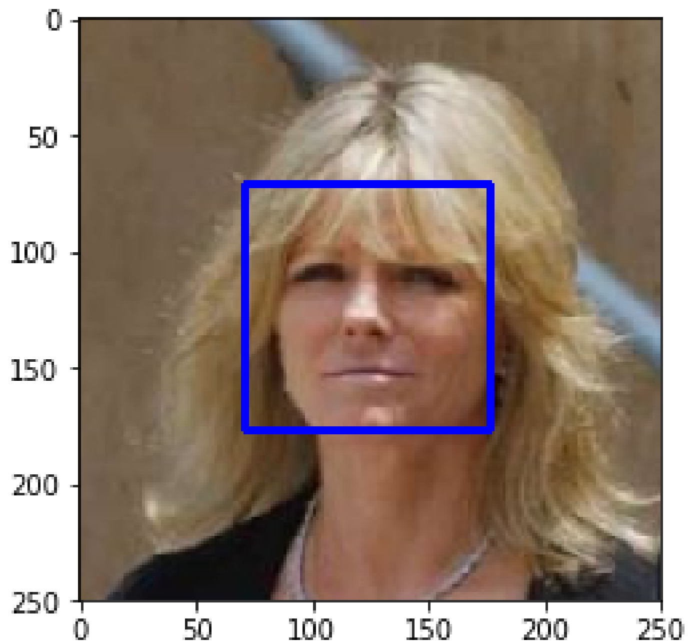
We use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

```
In [8]: # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[80])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

```python
# 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.

### 2.0.1 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 [9]: # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # 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
```

### 2.0.2 (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 [10]: 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.
         def face_percentage(images_path):
             '''
             Function that asses face_detector's perfomance
             by a set of images paths.
             Input: image files paths
             Output: percentage of the images with at least one face detected
             '''

             n_face_image=0
             for path in images_path:
                 if face_detector(path)==True:
                     n_face_image+=1

             return n_face_image/len(images_path)

         print("Percentage of images in human_files_short with face detected: {} %".format(face_
```

```
print("Percentage of images in dog_files_short with face detected: {} %".format(face_pe
```

```
Percentage of images in human_files_short with face detected: 100.0 %
Percentage of images in dog_files_short with face detected: 11.0 %
```

**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.

The predictive capability of a model is mainly reduced to the task that it was train for and the characteristics of the data set used for that purpose. As saw in the previous question, a model can be used for a task different that it's ineherent one, but probably with a lower performance compare to a model specifically trained for the task. The used Haar cascade classifier was built for identifying distinctive frontal face features and cause a blurred face or an image that lacks from a clear view of it doesnt't provide the features the model learnt to recognise, it is no wonder that the model will not behave as expected. Therefore it is perfectly valid to warn the user of that limitation.

Nonetheless, it's possible to add the capability to recognise a face without a clear view by training another model that consideres a significant proportion of images that meets the last characteristic. A model with this data will probably learn to recognize others distinctive features, like the shape of a human head, a nose from the side, etc...

```
In [7]: ## (Optional) TODO: Report the performance of another
        ## face detection algorithm on the LFW dataset
        ### Feel free to use as many code cells as needed.
```

---

## Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories. Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

```
In [11]: # define ResNet50 model
         ResNet50_model = ResNet50(weights='imagenet')
```

```
Downloading data from https://github.com/fchollet/deep-learning-models/releases/download/v0.2/re
102858752/102853048 [==============================] - 1s 0us/step
```