dog_app

June 2, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

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 Jupyter notebook.
## Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets:
Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

```
In [2]: human_files[0:10]

Out[2]: array(['/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg',
               '/data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg',
               '/data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg',
               '/data/lfw/Donald_Carty/Donald_Carty_0001.jpg',
               '/data/lfw/Barry_Switzer/Barry_Switzer_0001.jpg',
               '/data/lfw/Jeong_Se-hyun/Jeong_Se-hyun_0003.jpg',
               '/data/lfw/Jeong_Se-hyun/Jeong_Se-hyun_0008.jpg',
               '/data/lfw/Jeong_Se-hyun/Jeong_Se-hyun_0005.jpg',
               '/data/lfw/Jeong_Se-hyun/Jeong_Se-hyun_0007.jpg',
               '/data/lfw/Jeong_Se-hyun/Jeong_Se-hyun_0004.jpg'],
              dtype='<U90')
```

## Step 1: Detect Humans

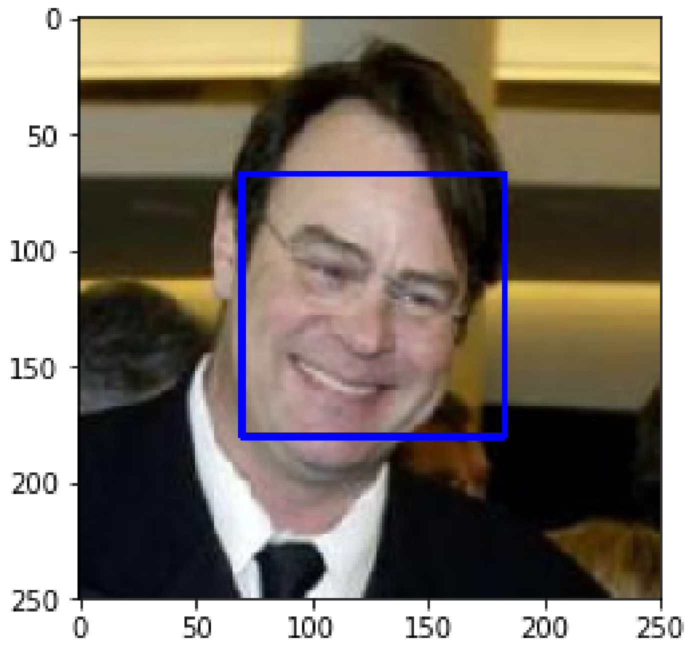
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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[0])
# 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.

1.1.1 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 [4]: # 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
```

1.1.2 (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?

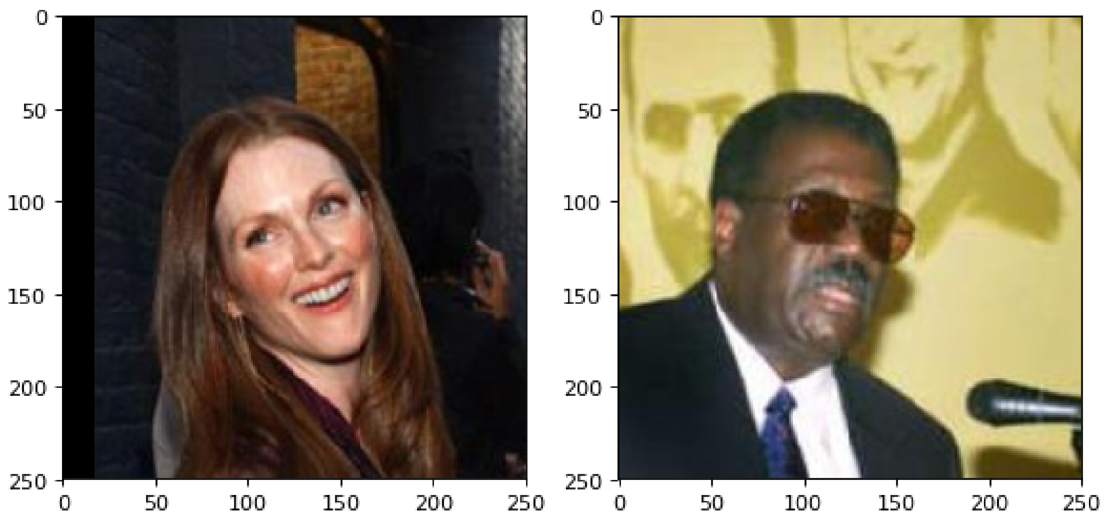
Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in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

Answer: - in 2 out of the first 100 humans pictures the face is not detected - in 17 out of the first 100 dogs pictures a human face is not detected. By looking more closely at the dog picture we can see that 4 out of the 17 pictures contains effectively a human face so we can reduce the classification error at 13 out of the 100 dogs pictures.

```
In [5]: from tqdm import tqdm

        human_files_short = human_files[:100]
        dog_files_short = dog_files[:100]

        #-#-# Do NOT modify the code above this line. #-#-#
```

```
In [6]: ## Test classification on human pictures
        total_image_test = 0
        true_positive = 0
        false_negative = 0
        false_negative_imgs = []
        for human in tqdm(human_files_short):
            total_image_test += 1
            if face_detector(human):
                true_positive += 1
            else:
                false_negative += 1
                false_negative_imgs.append(human)

        print("- in {} out of the first {} human pictures the face is not detected".format(false
        n_cols = 4
        fig = plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
        for i, picture in enumerate(false_negative_imgs):
            img = cv2.imread(picture)
            # convert BGR image to RGB for plotting
            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            ax = fig.add_subplot(false_negative//n_cols + 1, n_cols, i+1)
            ax.imshow(cv_rgb)
```

```
100%|| 100/100 [00:03<00:00, 29.66it/s]


- in 2 out of the first 100 human pictures the face is not detected
```



```
In [7]: ## Test classification on dogs pictures
        total_image_test = 0
        true_negative = 0
        false_positive = 0
        false_positive_imgs = []
        for dog in tqdm(dog_files_short):
            total_image_test += 1
            if face_detector(dog):
                false_positive += 1
                false_positive_imgs.append(dog)
            else:
                true_negative += 1
        print("- in {} out of the first {} dog pictures a human face is detected".format(false_p

        n_cols = 4
        fig = plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
        for i, picture in enumerate(false_positive_imgs):
            img = cv2.imread(picture)
            # convert BGR image to RGB for plotting
            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            ax = fig.add_subplot(false_positive//n_cols + 1, n_cols, i+1)
            ax.imshow(cv_rgb)
```

```
100%|| 100/100 [00:37<00:00,  2.67it/s]


- in 17 out of the first 100 dog pictures a human face is detected
```

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 `human_files_short` and `dog_files_short`.

```
In [8]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

## Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 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.

```
In [9]: import torch
        import torchvision.models as models

        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # check if CUDA is available
        use_cuda = torch.cuda.is_available()
        device = torch.device("cuda" if use_cuda else "cpu")
        print("The code is running on: " + str(device))
        # move model to GPU if CUDA is available
        VGG16 = VGG16.to(device)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:06<00:00, 90178134.16it/s]


The code is running on: cuda
```

Given an image, this pre-trained VGG-16 model returns a prediction (derived from the 1000 possible categories in ImageNet) for the object that is contained in the image.

1.1.4 (IMPLEMENTATION) Making Predictions with a Pre-trained Model

In the next code cell, you will write a function that accepts a path to an image (such as `'dogImages/train/001.Affenpinscher/Affenpinscher_00001.jpg'`) as input and returns the index corresponding to the ImageNet class that is predicted by the pre-trained VGG-16 model. The output should always be an integer between 0 and 999, inclusive.

Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [10]: VGG16
```

```
Out[10]: VGG(
           (features): Sequential(
             (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (1): ReLU(inplace)
             (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (3): ReLU(inplace)
             (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (6): ReLU(inplace)
             (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (8): ReLU(inplace)
             (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

In [11]:
```
import torchvision.transforms as transforms
from collections import OrderedDict
import torch.nn.functional as F
from PIL import Image
import torch.nn as nn

# Normalize the image (VGG is trained on normalized image)
loader = transforms.Compose([
    transforms.Resize((256,256)),  # scale imported image
    transforms.CenterCrop(224),
    transforms.ToTensor(), # transform it into a torch tensor
    transforms.Normalize((0.485, 0.456, 0.406),
                         (0.229, 0.224, 0.225))]) # normalize

def image_loader(image_name):
        image = Image.open(image_name)
        # fake batch dimension required to fit network's input dimensions
        image = loader(image).unsqueeze(0)
        return image.to(device, torch.float)

    def VGG16_predict(img_path):
        '''
        Use pre-trained VGG-16 model to obtain index corresponding to
        predicted ImageNet class for image at specified path

        Args:
            img_path: path to an image

        Returns:
            Index corresponding to VGG-16 model's prediction
        '''

        ## TODO: Complete the function.
        ## Load and pre-process an image from the given img_path
        image = image_loader( img_path )
        image = image.to(device)
        VGG16.eval()
        with torch.no_grad():
            logits = VGG16(image)

        preds = torch.topk(logits.cpu(), k=1)
            #print(probabilities.size())

        ## Return the *index* of the predicted class for that image
        return (preds[0].numpy()[0][0], preds[1].numpy()[0][0]) # predicted class index
```

```
In [12]: import json

         # Load class names
         labels_map = json.load(open("imagenet_class_index.json", "r"))
         labels_map = [labels_map[str(i)] for i in range(1000)]

         # Test on person image
         print("---- human file ----")
         for human in human_files_short[0:10]:
             prob, key = VGG16_predict(human)
             print("{:d} : {:s}".format(key,labels_map[key][1]))

         # Test on dog image
         print("---- dogs file ----")
         for dog in dog_files_short[0:10]:
             prob, key = VGG16_predict(dog)
             print("{:d} : {:s}".format(key,labels_map[key][1]))
```

```
---- human file ----
906 : Windsor_tie
456 : bow
400 : academic_gown
834 : suit
768 : rugby_ball
683 : oboe
834 : suit
400 : academic_gown
834 : suit
906 : Windsor_tie
---- dogs file ----
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
243 : bull_mastiff
246 : Great_Dane
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice 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 VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

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).

```
In [13]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             prob, key = VGG16_predict(img_path)
             isDog = key < 269 and key > 150
             return isDog, (prob, key) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

- in 0 out of the first 100 human pictures a dog is detected

- in 0 out of the first 100 dog pictures a dog is not detected

VGG performance are awesome!

```
In [14]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

In [15]: ## Test classification on human pictures
         total_image_test = 0
         true_positive = 0
         false_negative = 0
         false_negative_imgs = []
         for human in tqdm(human_files_short):
             total_image_test += 1
             is_dog, classification = dog_detector(human)
             if not is_dog:
                 true_positive += 1
             else:
                 false_negative += 1
                 false_negative_imgs.append((human, labels_map[classification[1]]))

         print("- in {} out of the first {} human pictures a dog is detected".format(false_negat
         n_cols = 4
         fig = plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
         for i, data in enumerate(false_negative_imgs):
             img = cv2.imread(data[0])
             # convert BGR image to RGB for plotting
             cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

             #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             ax = fig.add_subplot(false_negative//n_cols + 1, n_cols, i+1)
             ax.set_title(data[1])
             ax.imshow(cv_rgb)

100%|| 100/100 [00:03<00:00, 28.88it/s]

- in 0 out of the first 100 human pictures a dog is detected


<matplotlib.figure.Figure at 0x7fce206e0860>

In [16]: ## Test classification on dogs pictures
         total_image_test = 0
         true_negative = 0
         false_positive = 0
```

```
        false_positive_imgs = []
        for dog in tqdm(dog_files_short):
            total_image_test += 1
            is_dog, classification = dog_detector(dog)
            if not is_dog:
                false_positive += 1
                print(classification[1])
                false_positive_imgs.append((dog, labels_map[classification[1]]))
            else:
                true_negative += 1
        print("- in {} out of the first {} dog pictures a dog is not detected".format(false_pos

        n_cols = 4
        fig = plt.figure(figsize=(18, 16), dpi= 80, facecolor='w', edgecolor='k')
        for i, data in enumerate(false_positive_imgs):
            img = cv2.imread(data[0])
            # convert BGR image to RGB for plotting
            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            #gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            ax = fig.add_subplot(false_positive//n_cols + 1, n_cols, i+1)
            ax.set_title(data[1])
            ax.imshow(cv_rgb)
```

```
100%|| 100/100 [00:04<00:00, 23.79it/s]

- in 0 out of the first 100 dog pictures a dog is not detected
```

```
<matplotlib.figure.Figure at 0x7fce684cb908>
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [17]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

## Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You

must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [18]: import os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # check if CUDA is available
```

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")

batch_size = 128
num_workers = 0
img_size = 224 # 224
n_epochs = 1

### TODO: Write data loaders for training, validation, and test sets
normalization = transforms.Normalize((0.485, 0.456, 0.406),
                         (0.229, 0.224, 0.225)) # normalize

train_transform = transforms.Compose([
                                     #transforms.ColorJitter(brightness=50, contrast=5
                                     transforms.RandomResizedCrop(img_size),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     normalization])

test_transform = transforms.Compose([transforms.Resize(img_size+16),
                                     transforms.CenterCrop(img_size),
                                     transforms.ToTensor(),
                                     normalization])

# Define the pat of the images
main_dir  = './dogImages/'
train_dir = os.path.join(main_dir, 'train')
test_dir  = os.path.join(main_dir, 'test')
valid_dir = os.path.join(main_dir, 'valid')

# Create dataset based on ImageNet config
train_data = datasets.ImageFolder(train_dir, transform=train_transform)
test_data  = datasets.ImageFolder(test_dir,  transform=test_transform)
valid_data = datasets.ImageFolder(valid_dir, transform=test_transform)

# Create dataloader
train_indices = [i for i in range(300)]
train_loader = torch.utils.data.DataLoader(train_data,
                                          batch_size=batch_size,
                                          shuffle=True,
                                          num_workers=num_workers)


test_loader  = torch.utils.data.DataLoader(test_data,
                                          batch_size=batch_size,
                                          shuffle=False,
                                          num_workers=num_workers)
```

```
valid_loader = torch.utils.data.DataLoader(valid_data,
                                           batch_size=batch_size,
                                           shuffle=False,
                                           num_workers=num_workers)

loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: - I choose a size of 224 because by visually inspecting the resulting images they are sufficiently precise to enable the classification of dog breeds. Nevertheless this resolution it is commonly used in classification tasks. Another important characteristic is that 224 can be divided by 2ˆ5 and this is perfect since I wanted to build a network with 5 max_pool layers. - Mostly the image of dogs are taken with a dog standing on the ground and therefore I only applied a random horizontal flip. I was planning to add a ColorJitter after the first test but as a matter of fact after the first epochs I saw that this was not needed since the achived performances were more than enugh

1.1.8 (IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. Use the template in the code cell below.

```
In [19]: import torch.nn as nn
         import torch.nn.functional as F

         num_classes = 133 # total classes

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.max_pool = torch.nn.MaxPool2d(2, stride=2)
                 self.dropout = torch.nn.Dropout(p=0.5)

                 #self.conv1 = torch.nn.Conv2d(3, 64, kernel_size=(3,3), padding=1)
                 #self.conv2 = torch.nn.Conv2d(64, 128, kernel_size=(3,3), padding=1)
                 #self.conv3 = torch.nn.Conv2d(128, 256, kernel_size=(3,3), padding=1)
                 #self.conv4 = torch.nn.Conv2d(256, 512, kernel_size=(3,3), padding=1)
                 #self.conv5 = torch.nn.Conv2d(512, 512, kernel_size=(3,3), padding=1)
```

```
        #self.fc1 = torch.nn.Linear(2*7*7*512, 4096 )
        #self.fc2 = torch.nn.Linear(4096, 4096)
        #self.fc3 = torch.nn.Linear(4096, num_classes)

        self.conv1 = torch.nn.Conv2d(3, 32, kernel_size=(3,3), padding=1)
        self.conv2 = torch.nn.Conv2d(32, 64, kernel_size=(3,3), padding=1)
        self.conv3 = torch.nn.Conv2d(64, 128, kernel_size=(3,3), padding=1)
        self.conv4 = torch.nn.Conv2d(128, 256, kernel_size=(3,3), padding=1)
        self.conv5 = torch.nn.Conv2d(256, 256, kernel_size=(3,3), padding=1)



        self.fc1 = torch.nn.Linear(7*7*256, 1024 )
        self.fc2 = torch.nn.Linear(1024, 512 )
        self.fc3 = torch.nn.Linear(512, num_classes )





    def forward(self, x):
        ## Define forward behavior

        x = F.relu(self.conv1(x))
        x = self.max_pool(x)

        x = F.relu(self.conv2(x))
        x = self.max_pool(x)

        x = F.relu(self.conv3(x))
        x = self.max_pool(x)

        x = F.relu(self.conv4(x))
        x = self.max_pool(x)

        x = F.relu(self.conv5(x))
        x = self.max_pool(x)

        x = x.view(x.shape[0], -1)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

#-#-# You so NOT have to modify the code below this line. #-#-#
```

```
# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch = model_scratch.cuda()
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step.

Answer: I started from the famouse VGG architecture and I tried to reduce the number of layers. In the first part I squeese the input tensor with 5 convolutional layers and 5 max pooling layers. Starting from a tensor with size 224x224x3 I end up with a tensor of size 7x7x256. At this point I flattern this tensor and I give the resulting feature as an input to the classification module. The last part is composed by 3 dense fully connected layers separated by dropout unit that are inserted to help the network on training a greater amount of its weights and obtaining a more robust result.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [20]: import torch.optim as optim

         ### TODO: select loss function
         criterion_scratch = nn.CrossEntropyLoss()


         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01, momentum=0.9)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [21]: import numpy as np
         train_losses = []
         valid_losses = []
         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):
                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0
```

```
correct = 0
total = 0
###################
# train the model #
###################
model.train()
for batch_idx, (data, target) in enumerate(loaders['train']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## find the loss and update the model parameters accordingly
    output = model(data)
    loss = criterion(output, target)

    optimizer.zero_grad() # reset gradients
    loss.backward() # accumulate gradents
    optimizer.step()

    ## record the average training loss, using something like
    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

######################
# validate the model #
######################
model.eval()
with torch.no_grad():
    for batch_idx, (data, target) in enumerate(loaders['valid']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## update the average validation los
        output = model(data)
        loss = criterion(output, target)
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_l

        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().
        total += data.size(0)

# print training/validation statistics
print('Epoch: {}/{}'.format(epoch, n_epochs))
print('\t|-> Training Loss: {:.6f}'.format(train_loss))
print('\t|-> Validation Loss: {:.6f}'.format(valid_loss))
train_losses.append(train_loss)
valid_losses.append(valid_loss)
print('\t|-> Validation Accuracy: %2d%% (%2d/%2d)' % (
                            100. * correct / total, correct, total))

                ## TODO: save the model if validation loss has decreased
                if valid_loss < valid_loss_min:
                    valid_loss_min = valid_loss
                    torch.save(model.state_dict(), save_path)
                    print('\t|-> Saving model new best validation Loss: {:.6f}'.format(
                        valid_loss
                        ))
            # return trained model
            return model
```

```
In [22]: # train the model
         from workspace_utils import active_session
         import os
         checkpoint_file = 'model_scratch.pt'
         resume = True

         if os.path.exists(checkpoint_file): # load the model if already exist
             if resume:
                 model_scratch.load_state_dict(torch.load(checkpoint_file))
                 n_epochs = 1

         with active_session():
             model_scratch = train(n_epochs, loaders_scratch, model_scratch, optimizer_scratch,
                                   criterion_scratch, use_cuda, checkpoint_file)

         # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load(checkpoint_file))
```

```
Epoch: 1/1
        |-> Training Loss: 1.451095
        |-> Validation Loss: 1.761979
        |-> Validation Accuracy: 54% (454/835)
        |-> Saving model new best validation Loss: 1.761979
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [23]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
             test_loss = 0.
             correct = 0.
             total = 0.
```

```
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['test']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the loss
            loss = criterion(output, target)
            # update average test loss
            test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
            # convert output probabilities to predicted class
            pred = output.data.max(1, keepdim=True)[1]
            # compare predictions to true label
            correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
            total += data.size(0)

        print('Test Loss: {:.6f}\n'.format(test_loss))

        print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
            100. * correct / total, correct, total))
```

```
In [24]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))

         # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 1.757238


Test Accuracy: 56% (474/836)
```

## Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [25]: ## TODO: Specify data loaders
         import os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # check if CUDA is available
         use_cuda = torch.cuda.is_available()
         device = torch.device("cuda" if use_cuda else "cpu")

         batch_size = 128
         num_workers = 0
         img_size = 224
         n_epochs = 100

         ### TODO: Write data loaders for training, validation, and test sets
         normalization = transforms.Normalize((0.485, 0.456, 0.406),
                                  (0.229, 0.224, 0.225)) # normalize

         train_transform = transforms.Compose([
                                             #transforms.ColorJitter(brightness=50, contrast=50
                                             transforms.RandomResizedCrop(img_size),
                                             transforms.RandomHorizontalFlip(),
                                             transforms.ToTensor(),
                                             normalization])

         test_transform = transforms.Compose([transforms.Resize(img_size+16),
                                             transforms.CenterCrop(img_size),
                                             transforms.ToTensor(),
                                             normalization])

         # Define the pat of the images
         main_dir  = './dogImages/'
         train_dir = os.path.join(main_dir, 'train')
         test_dir  = os.path.join(main_dir, 'test')
         valid_dir = os.path.join(main_dir, 'valid')

         # Create dataset based on ImageNet config
         train_data = datasets.ImageFolder(train_dir, transform=train_transform)
         test_data  = datasets.ImageFolder(test_dir,  transform=test_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=test_transform)

         # Create dataloader
         train_indices = [i for i in range(300)]
         train_loader = torch.utils.data.DataLoader(train_data,
                                                  batch_size=batch_size,
                                                  shuffle=True,
                                                  num_workers=num_workers)


test_loader  = torch.utils.data.DataLoader(test_data,
                                                  batch_size=batch_size,
                                                  shuffle=False,
                                                  num_workers=num_workers)


valid_loader = torch.utils.data.DataLoader(valid_data,
                                                  batch_size=batch_size,
                                                  shuffle=False,
                                                  num_workers=num_workers)

loaders_transfer = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [26]: import torchvision.models as models
         import torch.nn as nn
         num_classes = 133 # total classes

         ## TODO: Specify model architecture
         # Load the pretrained model from pytorch

         model_transfer = models.resnet18(pretrained=True)

         # Freeze training for all layers
         for param in model_transfer.parameters():
             param.requires_grad = False

         n_input = model_transfer.fc.in_features

         model_transfer.fc = nn.Linear(n_input, num_classes)

         if use_cuda:
             model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet18-5c106cde.pth" to /root/.torch/models/
```

```
100%|| 46827520/46827520 [00:00<00:00, 85889992.49it/s]
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer: ResNet is a network architecture that achive great performance in the classification task. I choose the resnet18 for the sake of keeping limited the dimension of the network! Nevertheless I think that the 1000 classes on which this ResNet was trained require the netork to learn a set of feature that can be beneficial even for the classificaton of the dog breed therefore I decided to freeze the parameters of the convolutional part of the netowrk and only retrain the final classifier. The final classifier is composed by only one fully connected network that takes as input a vector of 512 features and I modify it to have an logits ouput (one for each dog breed). The relative small size of the final fully connected classifier guarantee a limited number of parameters to tune helping in reducing the overfitting problem and in speeding up the process.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [27]: import torch.optim as optim

         # specify loss function (categorical cross-entropy)
         criterion_transfer = nn.CrossEntropyLoss()

         # specify optimizer (stochastic gradient descent) and learning rate = 0.001
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [28]: # Check layer with active gradient
         for name, param in model_transfer.named_parameters():
             if param.requires_grad:
                 print("\t=> ", name)

        =>  fc.weight
        =>  fc.bias
```

```
In [29]: # train the model
         from workspace_utils import active_session
         import os
         checkpoint_file = 'model_transfer.pt'
         resume = True

         if os.path.exists(checkpoint_file): # load the model if already exist
```

```
    if resume:
        model_transfer.load_state_dict(torch.load(checkpoint_file))
        n_epochs = 1


with active_session():
    model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transf

# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load(checkpoint_file))
```

```
Epoch: 1/1
        |-> Training Loss: 1.725454
        |-> Validation Loss: 1.296292
        |-> Validation Accuracy: 75% (627/835)
        |-> Saving model new best validation Loss: 1.296292
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [30]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

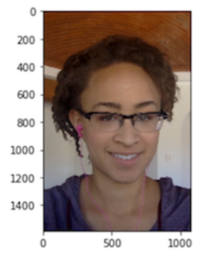
```
Test Loss: 1.292471


Test Accuracy: 77% (652/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [40]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         from PIL import Image

         checkpoint_file = 'model_transfer.pt'
         model_transfer.load_state_dict(torch.load(checkpoint_file, map_location='cpu'))

         breed_transform = test_transform
         breed_model = model_transfer

         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset
```

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

```
def image_loader1(image_name):
    image = Image.open(image_name)
    # fake batch dimension required to fit network's input dimensions
    image = breed_transform(image).unsqueeze(0)
    return image.to(device, torch.float)

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    image = image_loader1(img_path)
    breed_model.eval()
    with torch.no_grad():
        logits = breed_model(image)
    idx = torch.argmax(logits)
    return class_names[idx]
```

## Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [41]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
```

```
    is_dog, _ = dog_detector(img_path)
    dog_breed = predict_breed_transfer(img_path)
    if is_dog:
        print("wof wof wof! \n (eng Hello {}!)".format(dog_breed))
    else:
        is_human = face_detector(img_path)
        if is_human:
            print("Hello human")
            print("if you were a dog you'd be a {}".format(dog_breed))
        else:
            print("I can not understand if you are a dog or a human!")


    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    print(file)
```

## Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

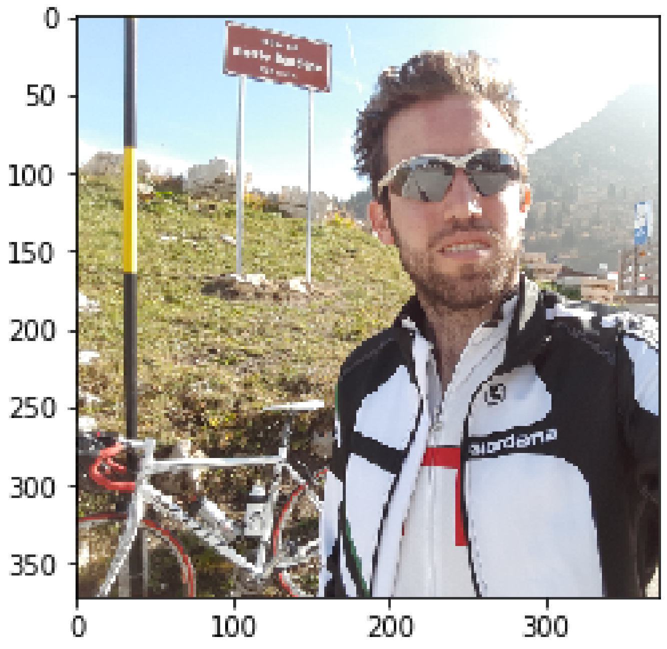
Overall I obtained better result than I was expecting expecially from my custom designed model. Even after 150 epoch the custom network was continuously improving without overfitting so if at the endo of the course I will have some remaining budget of GPU time I would love to try to exploit more the potentiality of my architecture. Nevertheless at the moment I only use a very basic data augmentation and could be interesting to see if increasing the data augmentation can help in achieving even better results. Nevertheless increasing the ~7000 dogs images in the training set will surly help in obtaining better results.

In summary: - train for more ephocs - better data augmentation - increase dataset size

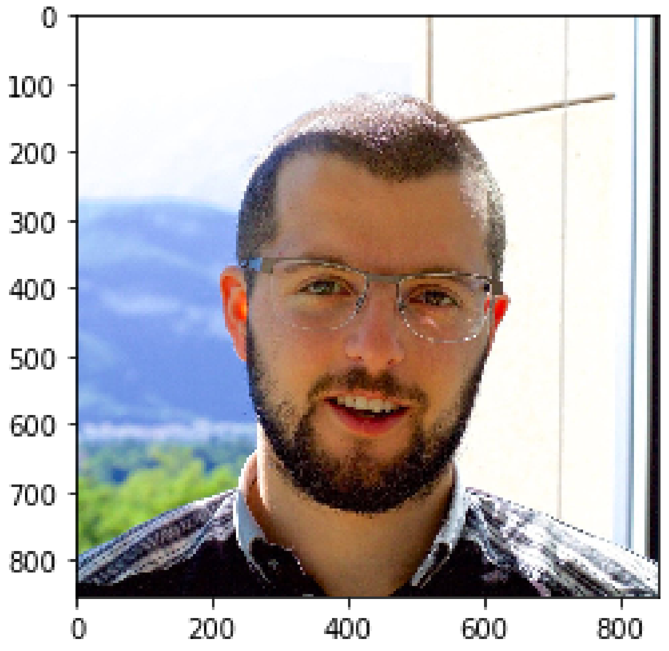
```
In [42]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         from os import listdir
         from os.path import isfile, join
```

```python
imgs_path = "./images/custom"
image_files = [f for f in listdir(imgs_path) if isfile(join(imgs_path, f))]

for file in image_files:
    if file[-4:] == ".jpg":
        run_app(join(imgs_path,file))
        print("\n\n")
```

```
I can not understand if you are a dog or a human!
```



```
Gionata.jpg



Hello human
if you were a dog you'd be a Kerry blue terrier
```

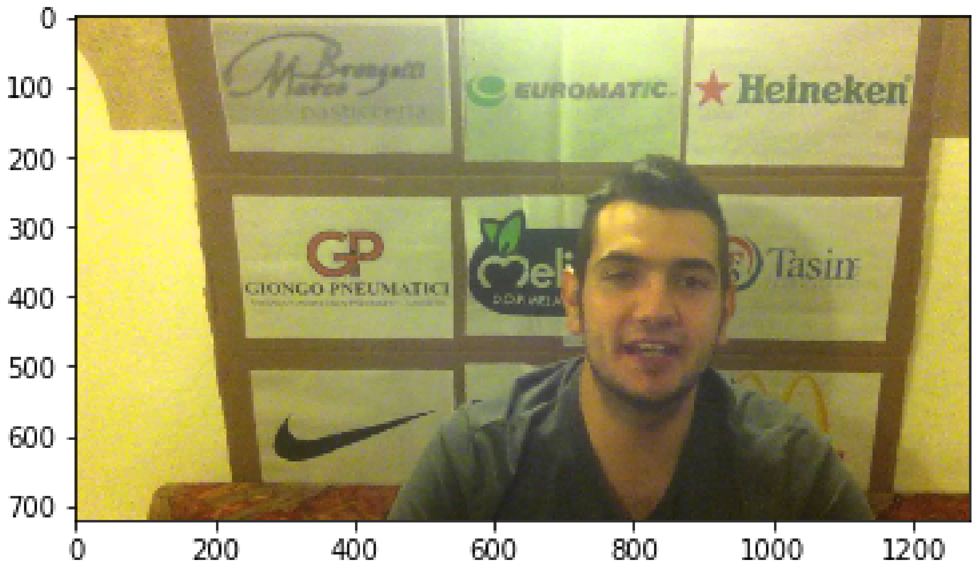


```
Valerio.jpg
```

```
wof wof wof!
 (eng Hello Affenpinscher!)
```

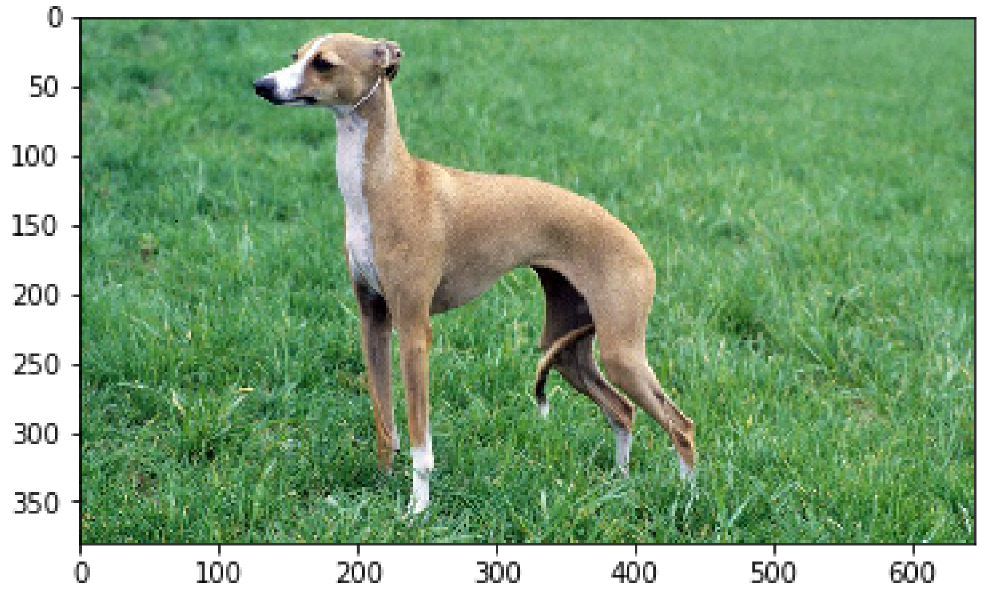
```
Affenpinscher.jpg
```

```
Hello human
if you were a dog you'd be a Chihuahua
```



```
Paolo.jpg
```

```
wof wof wof!
 (eng Hello Italian greyhound!)
```

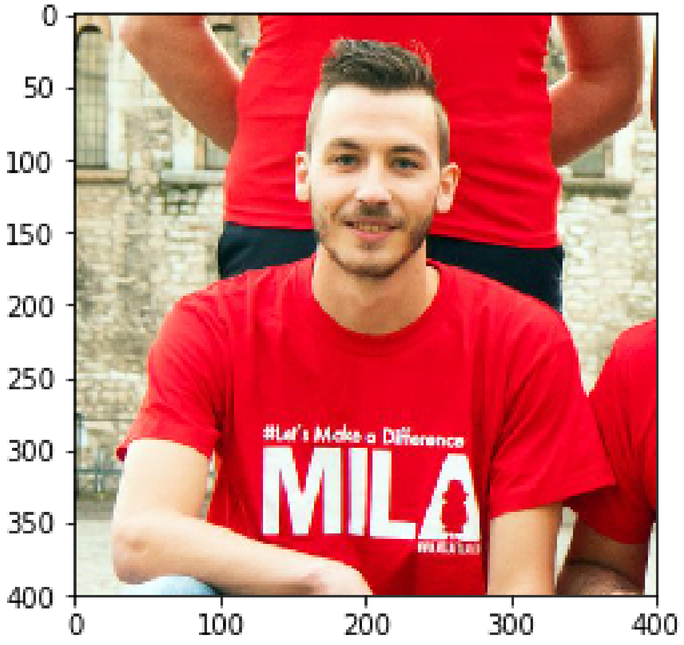


```
ItalianGreyhound.jpg
```

```
wof wof wof!
 (eng Hello Pekingese!)
```

```
Pekingese.jpg
```

```
Hello human
if you were a dog you'd be a Dogue de bordeaux
```



```
Alessandro.jpg
```

```
In [ ]:
```