dog_app

October 26, 2019

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.
```

Step 1: Detect Humans

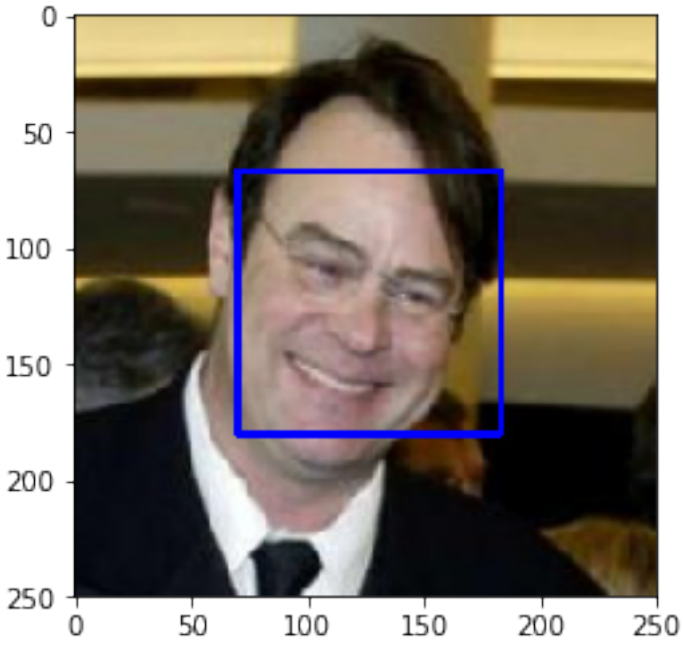
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [2]: 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 [3]: # 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: Using haarcascades face detector:
Human faces detected in the first 100 human images: 98%
Human faces detected in the first 100 dog images: 17%

```
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. #-#-#

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

        human_faces = 0 # Human faces detected in  the first 100 human images
        dog_faces = 0 # Human faces detected in the first 100 dog images

        for index in range(100):
            if face_detector(human_files_short[index]):
                human_faces += 1
            if face_detector(dog_files_short[index]):
                dog_faces += 1
        print("Human faces detected in the humans images dataset: ", human_faces, "%")
        print("Human faces detected in the dogs images dataset: ", dog_faces, "%")
```

```
Human faces detected in the humans images dataset:  98 %
Human faces detected in the dogs images dataset:  17 %
```

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


         #Alternate Approach: OPENCV LBP CASCADE CLASSIFIER
         #load cascade classifier training file for lbpcascade
         lbp_face_cascade = cv2.CascadeClassifier('lbpcascade/lbpcascade_frontalface_improved.xm

         def lbp_face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = lbp_face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

```
In [13]: lbp_human_faces = 0 # Human faces detected in  the first 100 human images
         lbp_dog_faces = 0 # Human faces detected in the first 100 dog images

         for index in range(100):
             if lbp_face_detector(human_files_short[index]):
                 lbp_human_faces += 1
             if lbp_face_detector(dog_files_short[index]):
                 lbp_dog_faces += 1
         print("Human faces detected in the humans images dataset (LBP): ", lbp_human_faces, "%"
         print("Human faces detected in the dogs images dataset: (LBP)", lbp_dog_faces, "%")
```

```
Human faces detected in the humans images dataset (LBP):  82 %
Human faces detected in the dogs images dataset: (LBP) 6 %
```

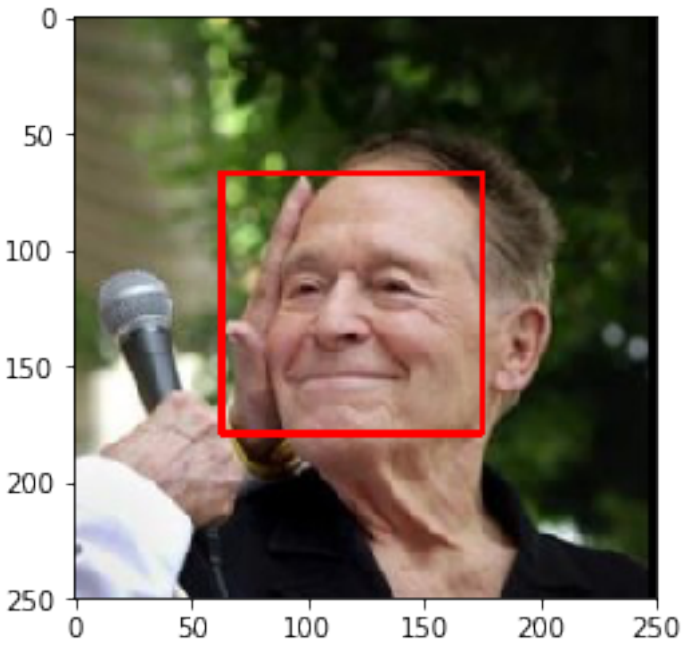
Comparing the two different cascades:

```
In [17]: def test_face_detector_on_image(img, face_detector):
             # convert BGR image to grayscale
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

             # find faces in image
             faces = face_detector.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-4),(x+w+4,y+h),(0,0,255),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()
```

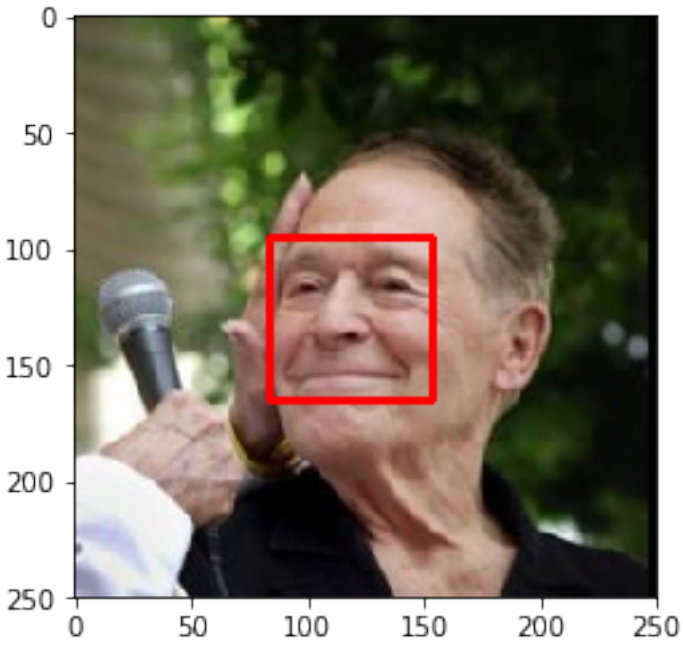
```
img = cv2.imread(human_files[107])
test_face_detector_on_image(img, face_cascade)
```

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



In [18]:
```
img = cv2.imread(human_files[107])
test_face_detector_on_image(img, lbp_face_cascade)
```

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



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 [19]: 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()

         # move model to GPU if CUDA is available
         if use_cuda:
             VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:46<00:00, 11937071.15it/s]
```

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 [23]: from PIL import Image, ImageFile

         import torchvision.transforms as transforms

         ImageFile.LOAD_TRUNCATED_IMAGES = True

         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
             ## Return the *index* of the predicted class for that image

             img = Image.open(img_path).convert('RGB')
             img_transform = transforms.Compose([transforms.Resize(size=(224,224)), transforms.T
             img = img_transform(img)[:3,:,:].unsqueeze(0)
             use_cuda = torch.cuda.is_available()

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

```
    pred = VGG16(img)
    return torch.max(pred,1)[1].item()
```

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 [26]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             predict = VGG16_predict(img_path)
             return predict >= 151 and predict <= 268
```

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:
Dogs percentage detected in the humans images dataset: 0%
Dogs percentage detected in the dogs images dataset: 97%

```
In [27]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detection(images):
             detected_count = 0
             for img in images:
                 if dog_detector(img):
                     detected_count += 1
             return detected_count, len(images)
         dog_faces_in_humans = dog_detection(human_files_short)
         dog_faces_in_dogs = dog_detection(dog_files_short)
         print("Dogs percentage detected in the humans images dataset: ", dog_faces_in_humans[0]
         print("Dogs percentage detected in the dogs images dataset: ", dog_faces_in_dogs[0]/dog

Dogs percentage detected in the humans images dataset:  0.0 %
Dogs percentage detected in the dogs images dataset:  0.97 %
```

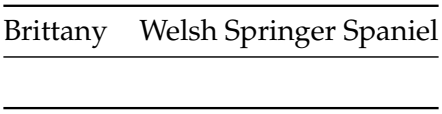
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

```
In [28]: ### (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)

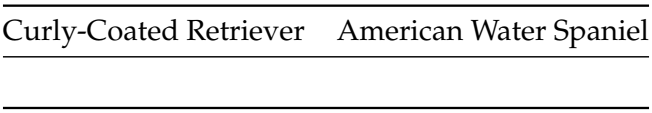
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of 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.



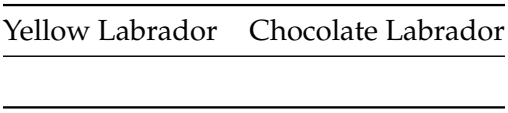
Brittany Welsh Springer Spaniel

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 [39]:
```
from PIL import Image, ImageFile

import torchvision.transforms as transforms

import os
from torchvision import datasets

ImageFile.LOAD_TRUNCATED_IMAGES = True

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
# Samples per batch
batch_size = 20
# Subprocesses used
num_workers = 0

# Image transforms
transformations = {
    'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.RandomRotation(5),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0
    'valid': transforms.Compose([transforms.Resize(size=(248)),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0
    'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0
}

# Directories
train_dir = '/data/dog_images/train/'
validation_dir = '/data/dog_images/valid/'
test_dir = '/data/dog_images/test/'

# Data Loaders
train_data = datasets.ImageFolder(train_dir, transform=transformations['train'])
validation_data = datasets.ImageFolder(validation_dir, transform=transformations['valid
test_data = datasets.ImageFolder(test_dir, transform=transformations['test'])

train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=True)
validation_loader = torch.utils.data.DataLoader(validation_data,
                                           batch_size=batch_size,
```

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

    loaders_scratch = {
        'train': train_loader,
        'valid': validation_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:

1. All the images are resized to 224 for input into the model following the VGG16 paper which is a size that works well. The images for training have a random crop while the test data has a center crop. The evaluation data its just resized to 224.
2. Yes, I decided to augment the data set to avoid overfitting. For the training images I used: random rotation, random horizontal flip and random crop of size of the input. All the images are normalized though.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [40]: import torch.nn as nn
         import torch.nn.functional as F

         # 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

                 # Convolutional layers
                 self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, stride=2, padding=1)

                 # Max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
```

```
        # Dropout layer
        self.dropout = nn.Dropout(0.3)

        # Linear layer (For classification)
        # 133 dog classes are present
        self.fc1 = nn.Linear(128*7*7, 784)
        self.fc2 = nn.Linear(784, 133)

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

        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        # Preparation for linear layer
        x = x.view(-1, 7*7*128)

        # Classification
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(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.cuda()
```

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

Answer:

1. I wanted to extract as many important features from the images as possible, hence I chose to use 3 convolutional layers and a max pooling layer between the 3 layers. As the size of the kernels is 3, the stranding of the convolutional layers and the padding is 1, the convolutional layers will reduce the dimenssions of the image by a half.

2. The pooling layers will discard spatial information that is irrelevant to the classifier. Those layers will reduce the size of the input by a half as well.

3. In the linear layer for the classifier, I used a dropout layer so that I could avoid overfitting. I used 2 layers, the last of them will output the probability for each of the 133 dog breed classes.

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 [41]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.05)
```

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 [44]: 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

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

                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()

                     # forward pass
```

```
            output = model(data)

            # calculating the loss
            loss = criterion(output, target)

            # backprop
            loss.backward()
            optimizer.step()


            # updating the training loss
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

        ######################
        # validate the model #
        ######################
        model.eval()
        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 loss

            # forward pass
            output = model(data)

            # calculating the  loss
            loss = criterion(output, target)

            # updating the average validation loss
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch,
            train_loss,
            valid_loss
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
            valid_loss_min,
            valid_loss))
            valid_loss_min = valid_loss

    # return trained model
```

```
            return model
```

In [45]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.576357         Validation Loss: 4.415347
Validation loss decreased (inf --> 4.415347).  Saving model ...
Epoch: 2         Training Loss: 4.504117         Validation Loss: 4.311998
Validation loss decreased (4.415347 --> 4.311998).  Saving model ...
Epoch: 3         Training Loss: 4.445042         Validation Loss: 4.262124
Validation loss decreased (4.311998 --> 4.262124).  Saving model ...
Epoch: 4         Training Loss: 4.378969         Validation Loss: 4.172400
Validation loss decreased (4.262124 --> 4.172400).  Saving model ...
Epoch: 5         Training Loss: 4.316105         Validation Loss: 4.122733
Validation loss decreased (4.172400 --> 4.122733).  Saving model ...
Epoch: 6         Training Loss: 4.248947         Validation Loss: 4.101139
Validation loss decreased (4.122733 --> 4.101139).  Saving model ...
Epoch: 7         Training Loss: 4.175209         Validation Loss: 3.969471
Validation loss decreased (4.101139 --> 3.969471).  Saving model ...
Epoch: 8         Training Loss: 4.157277         Validation Loss: 3.875675
Validation loss decreased (3.969471 --> 3.875675).  Saving model ...
Epoch: 9         Training Loss: 4.117508         Validation Loss: 3.906092
Epoch: 10         Training Loss: 4.046240         Validation Loss: 3.824624
Validation loss decreased (3.875675 --> 3.824624).  Saving model ...
Epoch: 11         Training Loss: 3.999779         Validation Loss: 3.852731
Epoch: 12         Training Loss: 3.953512         Validation Loss: 3.846833
Epoch: 13         Training Loss: 3.929067         Validation Loss: 3.811232
Validation loss decreased (3.824624 --> 3.811232).  Saving model ...
Epoch: 14         Training Loss: 3.866349         Validation Loss: 3.775447
Validation loss decreased (3.811232 --> 3.775447).  Saving model ...
Epoch: 15         Training Loss: 3.842497         Validation Loss: 3.670903
Validation loss decreased (3.775447 --> 3.670903).  Saving model ...
Epoch: 16         Training Loss: 3.792710         Validation Loss: 3.657252
Validation loss decreased (3.670903 --> 3.657252).  Saving model ...
Epoch: 17         Training Loss: 3.751626         Validation Loss: 3.678188
Epoch: 18         Training Loss: 3.705757         Validation Loss: 3.642530
Validation loss decreased (3.657252 --> 3.642530).  Saving model ...
Epoch: 19         Training Loss: 3.674816         Validation Loss: 3.633603
Validation loss decreased (3.642530 --> 3.633603).  Saving model ...
Epoch: 20         Training Loss: 3.657227         Validation Loss: 3.658544
```

In [46]:
```
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

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 [47]: 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))

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

```
Test Loss: 3.769286


Test Accuracy: 12% (103/836)
```

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 [48]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

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 [49]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)

         for param in model_transfer.parameters():
             param.requires_grad = False

         n_inputs = model_transfer.fc.in_features
         last_layer = nn.Linear(n_inputs, 133, bias=True)
         model_transfer.fc = last_layer
         last_layer_params = model_transfer.fc.parameters()

         for param in last_layer_params:
             param.requires_grad = True

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:03<00:00, 31116369.45it/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:

1. I choose resnet for model transfer as its performance on image classification its very good.
2. Then I replaced the last layer with a fully connected layer at the end so it could output the probabilities of one of the 133 breeds.
3. This architecture is pretty decent since resnet already has excellent feature detection. We just have to reduce the number of output classes accordingly.

4. This approach of transfer learning is very nice, as we prevent the rest of the network from training. Hence, we save time and compute resources.

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 [50]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.05)
```

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 [52]: # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 2.844607         Validation Loss: 1.095418
Validation loss decreased (inf --> 1.095418).  Saving model ...
Epoch: 2         Training Loss: 1.527436         Validation Loss: 0.784561
Validation loss decreased (1.095418 --> 0.784561).  Saving model ...
Epoch: 3         Training Loss: 1.203023         Validation Loss: 0.597667
Validation loss decreased (0.784561 --> 0.597667).  Saving model ...
Epoch: 4         Training Loss: 1.096464         Validation Loss: 0.497281
Validation loss decreased (0.597667 --> 0.497281).  Saving model ...
Epoch: 5         Training Loss: 1.032519         Validation Loss: 0.511495
Epoch: 6         Training Loss: 0.960511         Validation Loss: 0.476890
Validation loss decreased (0.497281 --> 0.476890).  Saving model ...
Epoch: 7         Training Loss: 0.924555         Validation Loss: 0.467276
Validation loss decreased (0.476890 --> 0.467276).  Saving model ...
Epoch: 8         Training Loss: 0.901580         Validation Loss: 0.435740
Validation loss decreased (0.467276 --> 0.435740).  Saving model ...
Epoch: 9         Training Loss: 0.893078         Validation Loss: 0.439226
Epoch: 10         Training Loss: 0.862404         Validation Loss: 0.397476
Validation loss decreased (0.435740 --> 0.397476).  Saving model ...
Epoch: 11         Training Loss: 0.844269         Validation Loss: 0.404880
Epoch: 12         Training Loss: 0.827698         Validation Loss: 0.392090
Validation loss decreased (0.397476 --> 0.392090).  Saving model ...
Epoch: 13         Training Loss: 0.835733         Validation Loss: 0.405750
Epoch: 14         Training Loss: 0.788498         Validation Loss: 0.380274
Validation loss decreased (0.392090 --> 0.380274).  Saving model ...
Epoch: 15         Training Loss: 0.808735         Validation Loss: 0.416256
Epoch: 16         Training Loss: 0.795071         Validation Loss: 0.395907
```

```
Epoch: 17         Training Loss: 0.771711         Validation Loss: 0.419597
Epoch: 18         Training Loss: 0.784641         Validation Loss: 0.398800
Epoch: 19         Training Loss: 0.752647         Validation Loss: 0.356555
Validation loss decreased (0.380274 --> 0.356555).  Saving model ...
Epoch: 20         Training Loss: 0.740679         Validation Loss: 0.394802
```

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

```
Test Loss: 0.451178


Test Accuracy: 85% (715/836)
```

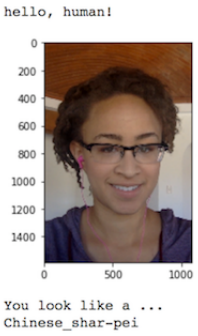
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [55]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

         # 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 train_data.classes]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path).convert('RGB')
             data_transform = transforms.Compose([transforms.Resize(size=(224,224)),
                                                  transforms.ToTensor()])
             img = data_transform(img)
             img = img.unsqueeze(0)
             if use_cuda:
                 img = img.cuda()
             pred = model_transfer(img)
             return class_names[torch.max(pred,1)[1].item()]
```

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



Sample Human Output

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 [57]:
```
### 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
    if dog_detector(img_path):
        print('Hi Doggo!')
        print('Your breed is most likely: ')
        print(predict_breed_transfer(img_path))
        img = cv2.imread(img_path)
        # 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()

    elif face_detector(img_path):
        print('Greetings Human!')
        print('You face resembles a: ')
        print(predict_breed_transfer(img_path))
        img = cv2.imread(img_path)
        # 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()

        else:
            print('Sorry! I could not detect human or dog in the image.')
```

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.

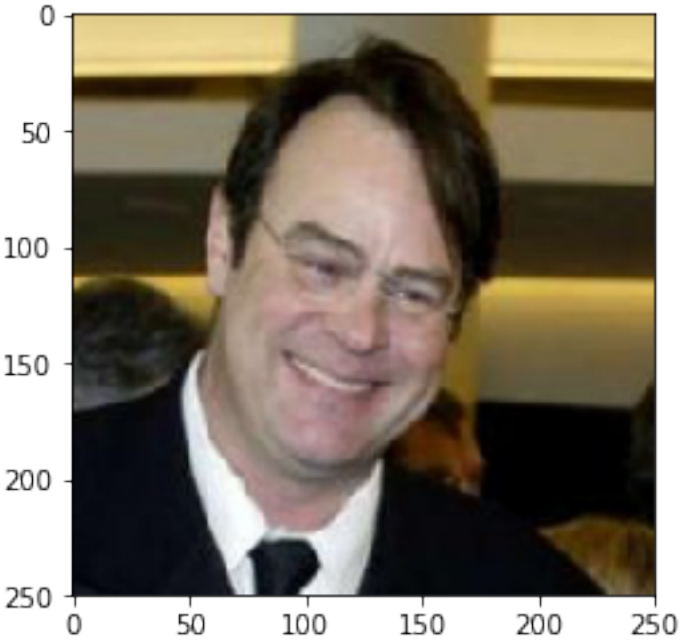
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

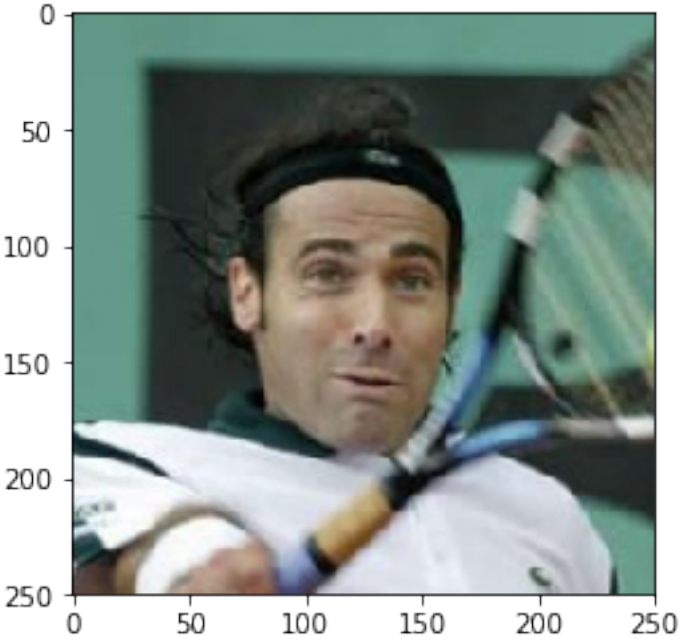
1. The output is pretty decent but there is scope of improvement. From what can be seen, a better accuracy is needed in differentiating between similar lookin dog breeds, otherwise breed classification gets messed up.
2. Handle better the case when there are multiple dogs and/or humans in the image and provide classification for all of them.
3. We can also return the top N (i.e top 3) predicted classes and their probabilities, not just one class so give user a better idea about the breed.

```
In [58]: ## 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.

         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```

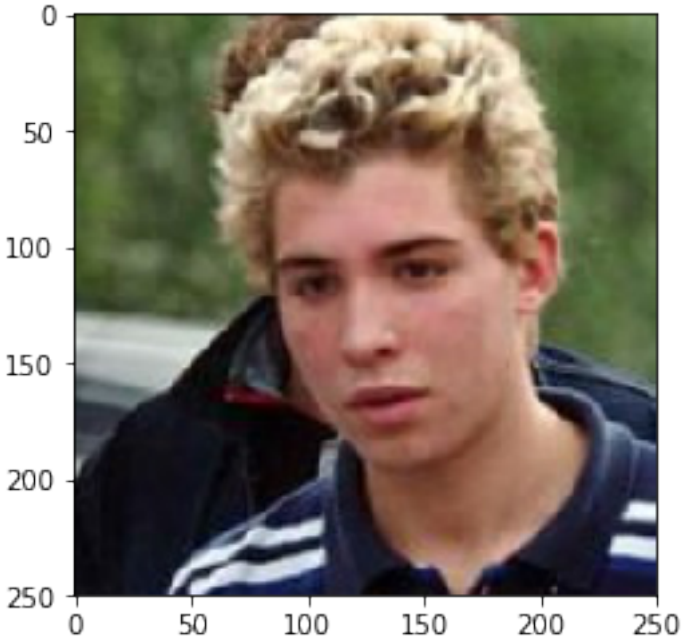
```
Greetings Human!
You face resembles a:
Bull terrier
```




```
Greetings Human!
You face resembles a:
American staffordshire terrier
```

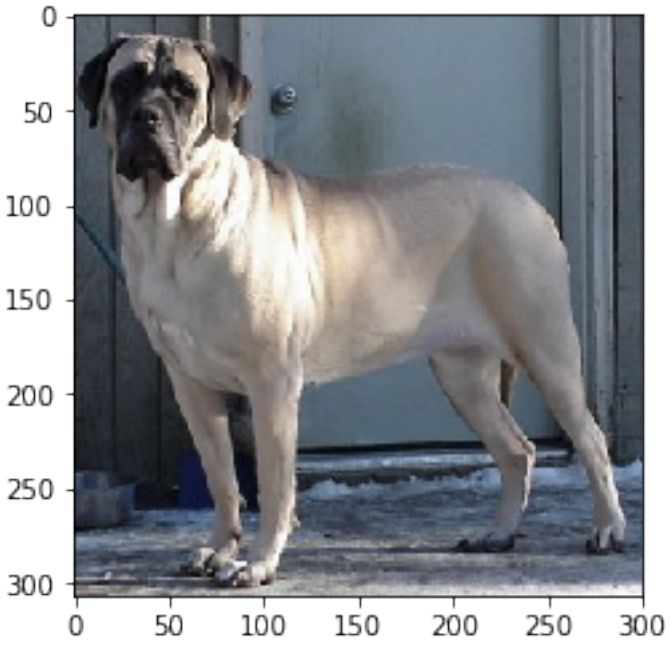


```
Greetings Human!
You face resembles a:
Bichon frise
```

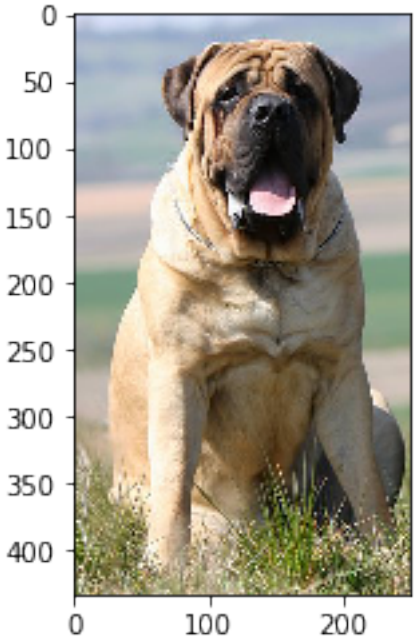



```
Hi Doggo!
Your breed is most likely:
Chinese shar-pei
```


```
Hi Doggo!
Your breed is most likely:
Mastiff
```



```
Hi Doggo!
Your breed is most likely:
Chinese shar-pei
```

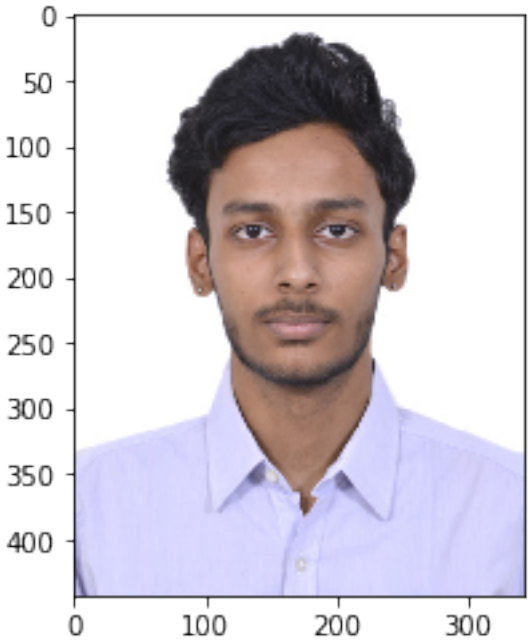



```
In [59]: run_app('images/Labrador_retriever_06449.jpg')
```

```
Hi Doggo!
Your breed is most likely:
Plott
```

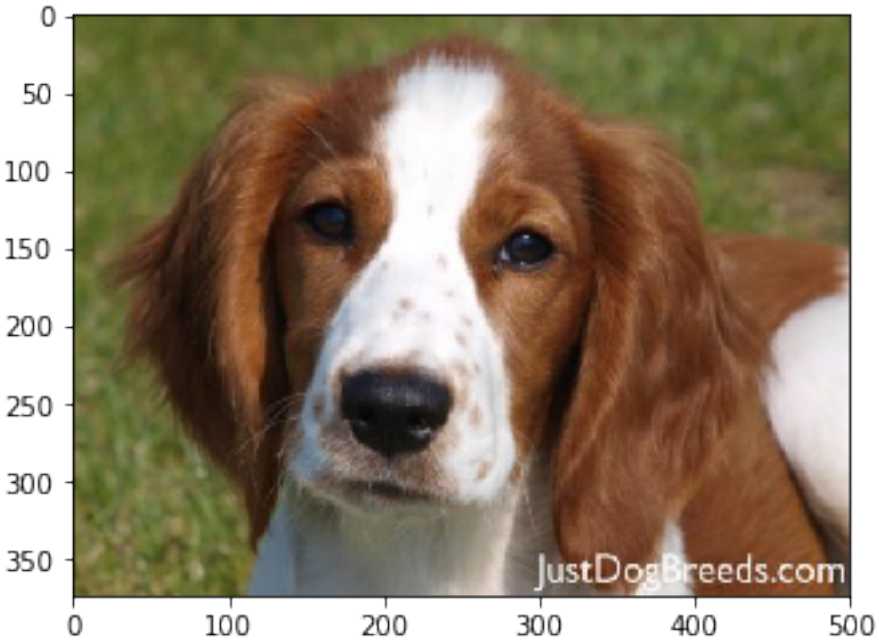

```
In [62]: run_app('images/ravi.JPG')
```

```
Greetings Human!
You face resembles a:
Cocker spaniel
```



In [63]: run_app('images/Welsh_springer_spaniel_08203.jpg')

```
Hi Doggo!
Your breed is most likely:
Basset hound
```



In [ ]: