dog_app

April 30, 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.
```

Step 1: Detect Humans

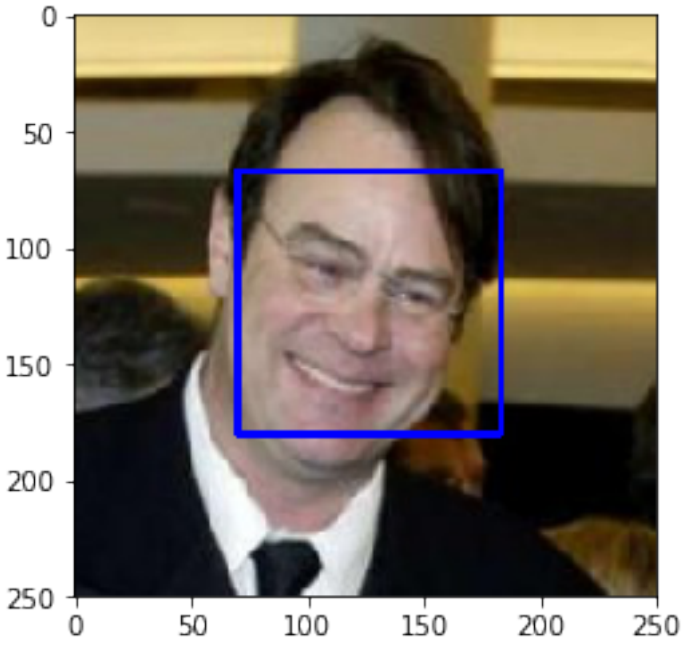
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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: - Percentage of human images with a detected human face is: 98 - Percentage of dog images with a detected human face is: 17

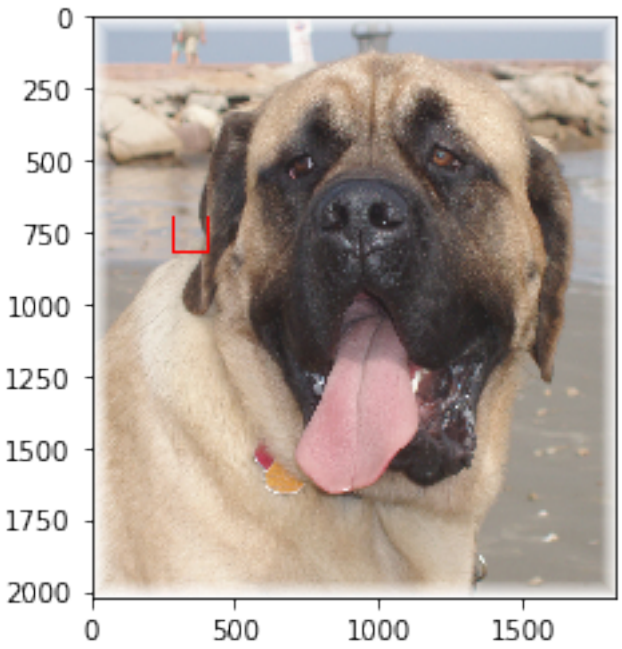
```
In [4]: from tqdm import tqdm

        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.
        detected_human = 0
        detected_dog = 0
        for i in range(len(human_files_short)):
            if face_detector(human_files_short[i]):
                detected_human +=1
            if face_detector(dog_files_short[i]):
                detected_dog +=1

        print("Percentage of human images with a detected human face is:", detected_human)
        print("Percentage of dog images with a detected human face is:", detected_dog)
```

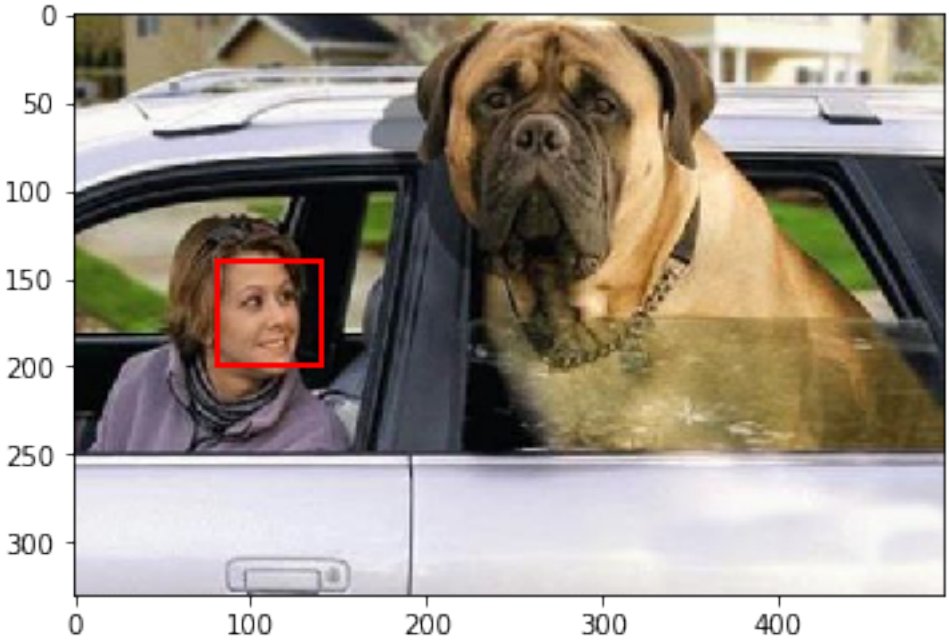
```
Percentage of human images with a detected human face is: 98
Percentage of dog images with a detected human face is: 17
```

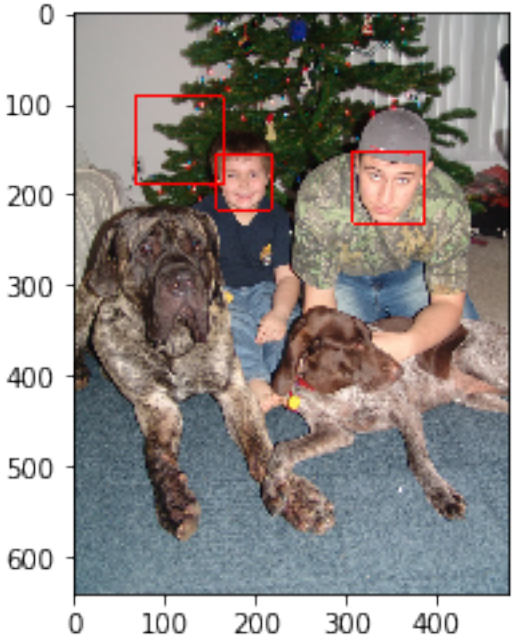
We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide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

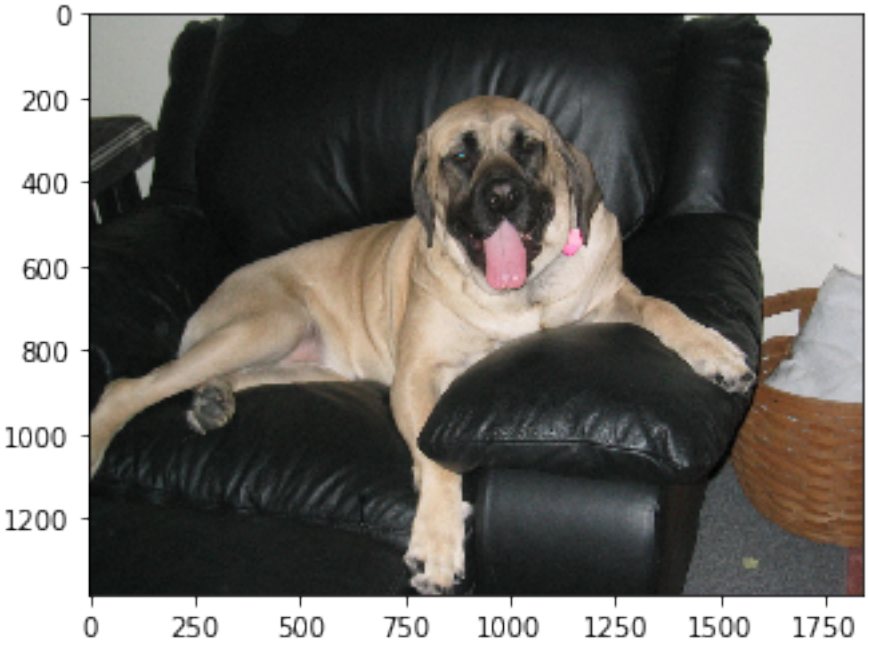
In [5]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
def face_detector_show(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # display the image, along with bounding box
    for (x,y,w,h) in faces:
        # add bounding box to color image
        cv2.rectangle(cv_rgb,(x,y),(x+w,y+h),(255,0,0),2)
    return len(faces) > 0, cv_rgb
```

In [6]:
```
# Lets check dog images with a detected human face
for i in range(len(human_files_short)):
    detected, img = face_detector_show(dog_files_short[i])
    if detected:
        plt.imshow(img)
        plt.show()
```

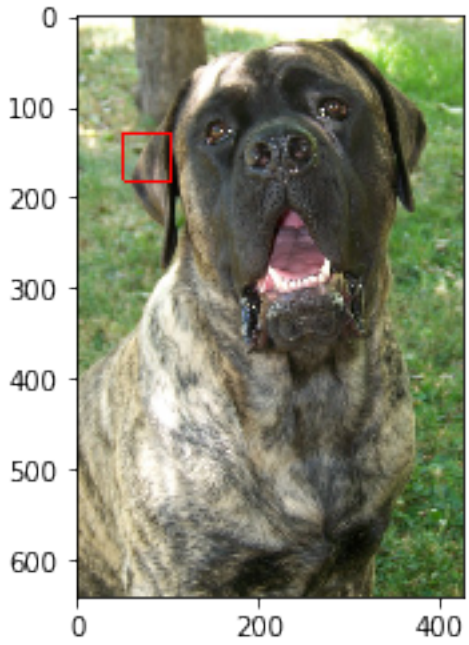


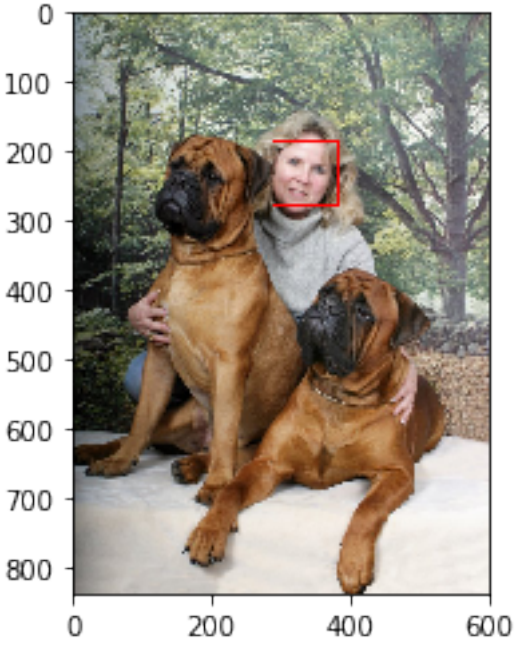

0
50
100
150
200
250
300
0
100
200
300
400

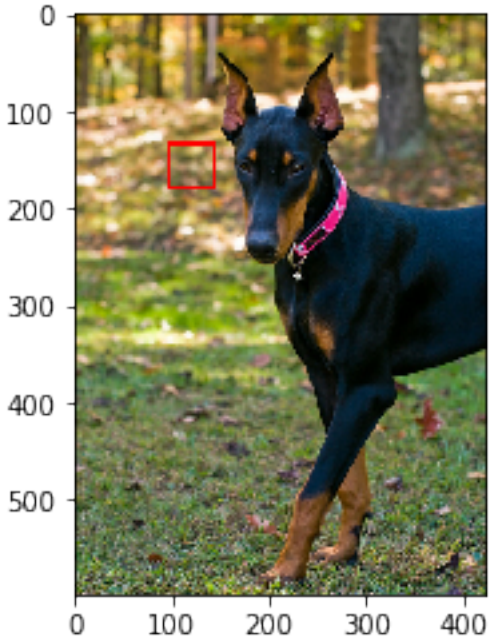

0
100
200
300
400
500
600
0
100
200
300
400


0
200
400
600
800
1000
1200
0
250
500
750
1000
1250
1500
1750


0
100
200
300
400
500
600
700
0
200
400
600


0
100
200
300
400
500
600
0
200
400

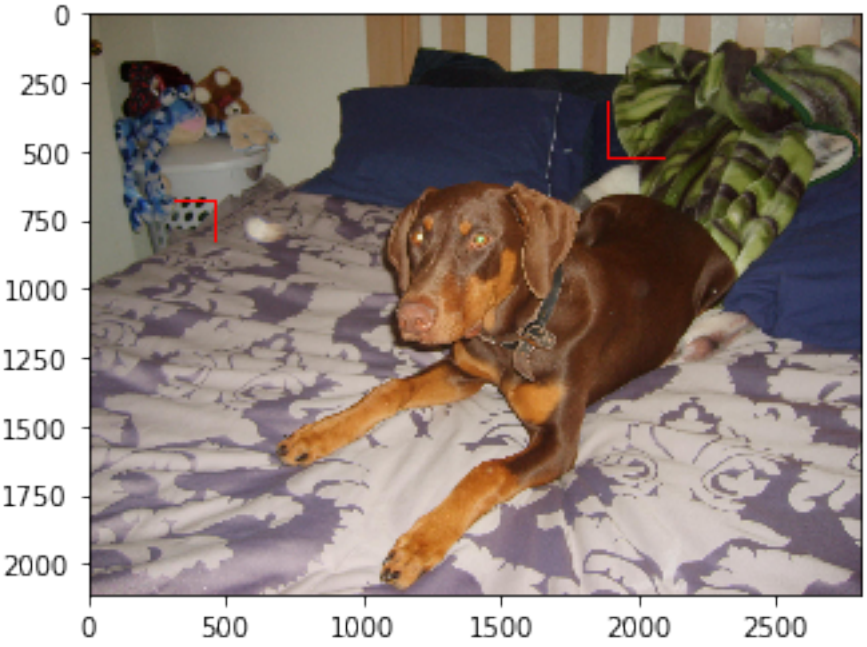

0
100
200
300
400
500
600
700
800
0
200
400
600


0
100
200
300
400
500
0
100
200
300
400

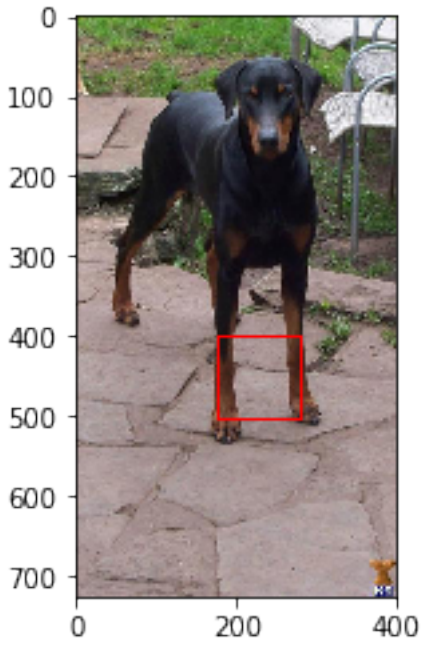

0
200
400
600
800
1000
1200
1400
1600
0
500
1000
1500
2000
2500


0
500
1000
1500
2000
2500
0
500
1000
1500
2000
2500
3000
3500

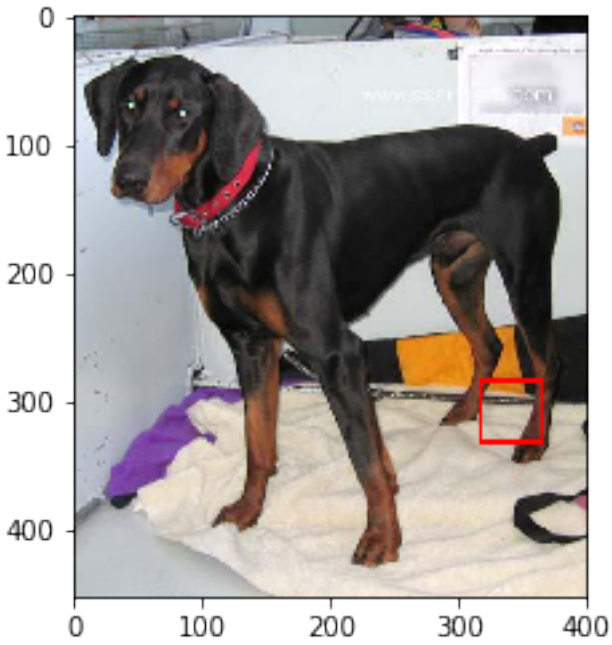

0
200
400
600
800
1000
0
200
400
600

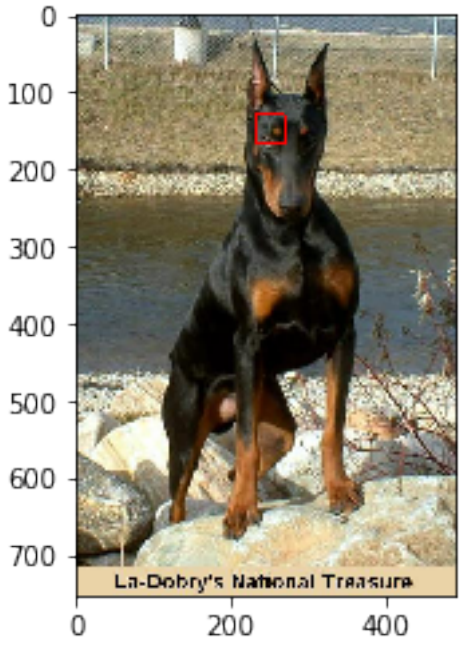

0
250
500
750
1000
1250
1500
1750
2000
0
500
1000
1500
2000
2500


0
200
400
600
800
1000
1200
1400
1600
0
250
500
750
1000


0
100
200
300
400
500
600
700
0
200
400

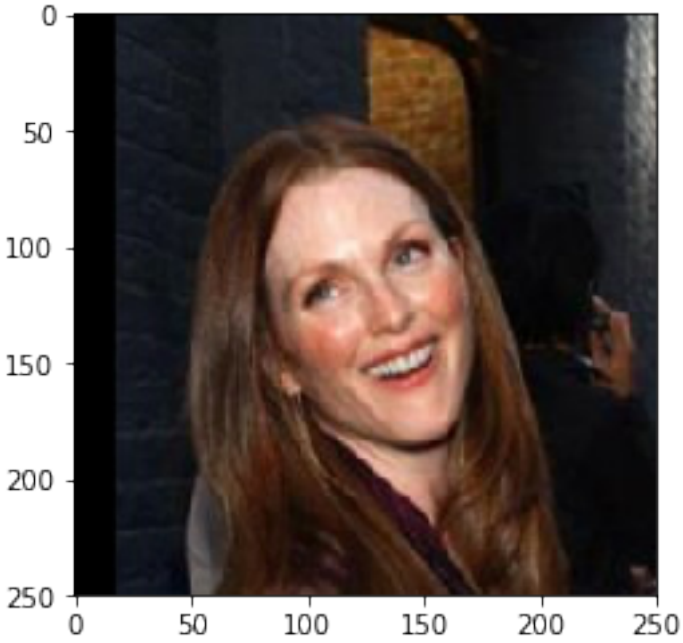

0
200
400
600
800
1000
1200
1400
1600
0
500
1000

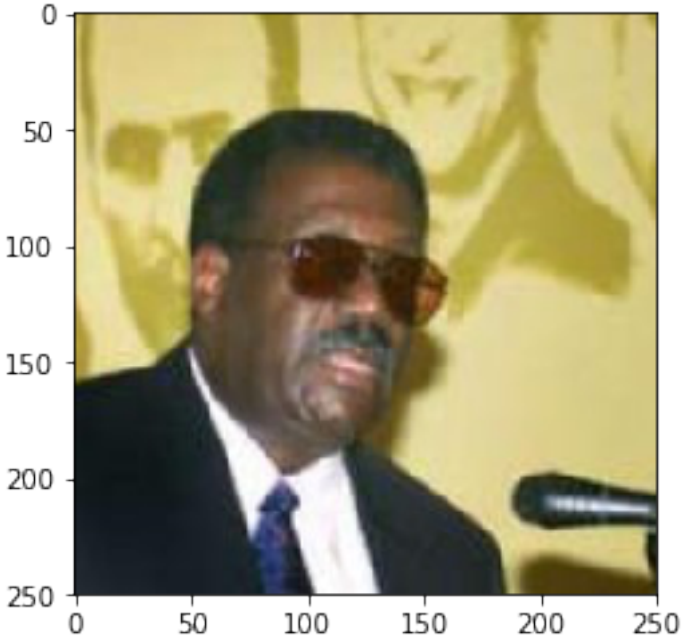





```
In [7]: # Lets check human images which are not detected as human face
        for i in range(len(human_files_short)):
            detected, img = face_detector_show(human_files_short[i])
```

```
if not detected:
    plt.imshow(img)
    plt.show()
```





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 [8]: 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 [00:05<00:00, 94189299.16it/s]
```

```
In [9]: VGG16
```

```
Out[9]: 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)
  )
)
```

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]: from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import torchvision.transforms as transforms

         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
        image = Image.open(img_path).convert('RGB')

        in_transform = transforms.Compose([
                        transforms.Resize((224, 224)),
                        transforms.ToTensor(),
                        transforms.Normalize((0.5, 0.5, 0.5),
                                             (0.5, 0.5, 0.5))])

        img_processed = in_transform(image).unsqueeze(0)

        if use_cuda:
            img_processed = img_processed.to('cuda')

        VGG16.eval()
        output = VGG16(img_processed)
        _, top_class = torch.max(output, 1)

        return top_class # predicted class index

    print(VGG16_predict(dog_files_short[0]))
```

```
tensor([ 243], device='cuda:0')
```

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 [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
```

```
    ## TODO: Complete the function.
    returned_class = VGG16_predict(img_path)
    if 151 <= returned_class <= 268:
        return True
    else:
        return False      # 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: - Percentage of dog images from human files is: 0 - Percentage of dog images from dog files is: 100

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         detected_dog_from_human_files = 0
         detected_dog_from_dog_files = 0
         for i in range(len(human_files_short)):
             if dog_detector(human_files_short[i]):
                 detected_dog_from_human_files +=1
             if dog_detector(dog_files_short[i]):
                 detected_dog_from_dog_files +=1

         print("Percentage of dog images from human files is:", detected_dog_from_human_files)
         print("Percentage of dog images from dog files is:", detected_dog_from_dog_files)
```

```
Percentage of dog images from human files is: 0
Percentage of dog images from dog files is: 100
```

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`.

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 [13]: !ls /data/dog_images

test  train  valid
```

```
In [14]: !pwd

/home/workspace/dog_project
```

```
In [15]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         dir_test = os.path.join("/data/dog_images", "test")
```

```
dir_valid = os.path.join("/data/dog_images", "valid")
dir_train = os.path.join("/data/dog_images", "train")
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.Resize(225),
                                       transforms.CenterCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5,),(0.5,))])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5,),(0.5,))])
train_data = datasets.ImageFolder(dir_train, transform=train_transforms)
valid_data = datasets.ImageFolder(dir_valid, transform=valid_transforms)
test_data = datasets.ImageFolder(dir_test, transform=valid_transforms)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=40, shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=40)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=40)

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

In [16]:
```
train_data.classes
```

Out[16]:
```
['001.Affenpinscher',
 '002.Afghan_hound',
 '003.Airedale_terrier',
 '004.Akita',
 '005.Alaskan_malamute',
 '006.American_eskimo_dog',
 '007.American_foxhound',
 '008.American_staffordshire_terrier',
 '009.American_water_spaniel',
 '010.Anatolian_shepherd_dog',
 '011.Australian_cattle_dog',
 '012.Australian_shepherd',
 '013.Australian_terrier',
 '014.Basenji',
 '015.Basset_hound',
 '016.Beagle',
 '017.Bearded_collie',
 '018.Beauceron',
 '019.Bedlington_terrier',
```

```
'020.Belgian_malinois',
'021.Belgian_sheepdog',
'022.Belgian_tervuren',
'023.Bernese_mountain_dog',
'024.Bichon_frise',
'025.Black_and_tan_coonhound',
'026.Black_russian_terrier',
'027.Bloodhound',
'028.Bluetick_coonhound',
'029.Border_collie',
'030.Border_terrier',
'031.Borzoi',
'032.Boston_terrier',
'033.Bouvier_des_flandres',
'034.Boxer',
'035.Boykin_spaniel',
'036.Briard',
'037.Brittany',
'038.Brussels_griffon',
'039.Bull_terrier',
'040.Bulldog',
'041.Bullmastiff',
'042.Cairn_terrier',
'043.Canaan_dog',
'044.Cane_corso',
'045.Cardigan_welsh_corgi',
'046.Cavalier_king_charles_spaniel',
'047.Chesapeake_bay_retriever',
'048.Chihuahua',
'049.Chinese_crested',
'050.Chinese_shar-pei',
'051.Chow_chow',
'052.Clumber_spaniel',
'053.Cocker_spaniel',
'054.Collie',
'055.Curly-coated_retriever',
'056.Dachshund',
'057.Dalmatian',
'058.Dandie_dinmont_terrier',
'059.Doberman_pinscher',
'060.Dogue_de_bordeaux',
'061.English_cocker_spaniel',
'062.English_setter',
'063.English_springer_spaniel',
'064.English_toy_spaniel',
'065.Entlebucher_mountain_dog',
'066.Field_spaniel',
'067.Finnish_spitz',
```

```
'068.Flat-coated_retriever',
'069.French_bulldog',
'070.German_pinscher',
'071.German_shepherd_dog',
'072.German_shorthaired_pointer',
'073.German_wirehaired_pointer',
'074.Giant_schnauzer',
'075.Glen_of_imaal_terrier',
'076.Golden_retriever',
'077.Gordon_setter',
'078.Great_dane',
'079.Great_pyrenees',
'080.Greater_swiss_mountain_dog',
'081.Greyhound',
'082.Havanese',
'083.Ibizan_hound',
'084.Icelandic_sheepdog',
'085.Irish_red_and_white_setter',
'086.Irish_setter',
'087.Irish_terrier',
'088.Irish_water_spaniel',
'089.Irish_wolfhound',
'090.Italian_greyhound',
'091.Japanese_chin',
'092.Keeshond',
'093.Kerry_blue_terrier',
'094.Komondor',
'095.Kuvasz',
'096.Labrador_retriever',
'097.Lakeland_terrier',
'098.Leonberger',
'099.Lhasa_apso',
'100.Lowchen',
'101.Maltese',
'102.Manchester_terrier',
'103.Mastiff',
'104.Miniature_schnauzer',
'105.Neapolitan_mastiff',
'106.Newfoundland',
'107.Norfolk_terrier',
'108.Norwegian_buhund',
'109.Norwegian_elkhound',
'110.Norwegian_lundehund',
'111.Norwich_terrier',
'112.Nova_scotia_duck_tolling_retriever',
'113.Old_english_sheepdog',
'114.Otterhound',
'115.Papillon',
```

```
 '116.Parson_russell_terrier',
 '117.Pekingese',
 '118.Pembroke_welsh_corgi',
 '119.Petit_basset_griffon_vendeen',
 '120.Pharaoh_hound',
 '121.Plott',
 '122.Pointer',
 '123.Pomeranian',
 '124.Poodle',
 '125.Portuguese_water_dog',
 '126.Saint_bernard',
 '127.Silky_terrier',
 '128.Smooth_fox_terrier',
 '129.Tibetan_mastiff',
 '130.Welsh_springer_spaniel',
 '131.Wirehaired_pointing_griffon',
 '132.Xoloitzcuintli',
 '133.Yorkshire_terrier']
```

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:First a resize of 225x225 is applied to normalize the training set image, where most of the images will be shrinked, and some few stretched. After a random crop is applied with a size of 224x224. A size of 224 was chosen, similar to what VGG authors use, to try to keep the most information detail possible in the images, while going through multiple pooling layers to reduce its size.

Random flip and rotation is applied to enhance the training process. No translation was applied to avoid decentering the pictures from dog faces

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [17]: 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
                 hidden_1 = 1024
                 hidden_2 = 512
                 hidden_3 = 64
```

```
        self.conv11 = nn.Conv2d(3, 16, 3, padding=1)
        self.conv12 = nn.Conv2d(16, 16, 3, padding=1)
        self.conv13 = nn.Conv2d(16, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 32, 3, padding=1)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
        self.maxpool = nn.MaxPool2d(2,2)

        self.bn_1 = nn.BatchNorm2d(32)
        self.bn_2 = nn.BatchNorm2d(32)
        self.bn_3 = nn.BatchNorm2d(64)
        self.bn_4 = nn.BatchNorm2d(128)
        self.bn_5 = nn.BatchNorm2d(256)

        self.fc_1 = nn.Linear(256*7*7, hidden_1)
        self.fc_2 = nn.Linear(hidden_1, hidden_2)
        self.fc_3 = nn.Linear(hidden_2, hidden_3)
        self.fc_4 = nn.Linear(hidden_3, 133)

        self.dropout12 = nn.Dropout2d(p = 0.5)
        self.bn1_1 = nn.BatchNorm1d(hidden_1)
        self.bn1_2 = nn.BatchNorm1d(hidden_2)
        self.bn1_3 = nn.BatchNorm1d(hidden_3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv11(x))
        x = F.relu(self.conv12(x))
        x = self.bn_1(self.maxpool(F.relu(self.conv13(x))))
        x = self.dropout12(self.bn_2(self.maxpool(F.relu(self.conv2(x)))))
        x = self.bn_3(self.maxpool(F.relu(self.conv3(x))))
        x = self.bn_4(self.maxpool(F.relu(self.conv4(x))))
        x = self.dropout12(self.bn_5(self.maxpool(F.relu(self.conv5(x)))))

        x = x.view(-1, 256*7*7)
        x = self.bn1_1(F.relu(self.fc_1(x)))
        x = self.bn1_2(F.relu(self.fc_2(x)))
        x = self.bn1_3(F.relu(self.fc_3(x)))
        x = F.log_softmax(self.fc_4(x), dim=1)
        return x

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv11(x))
        x = F.relu(self.conv12(x))
        x = self.bn_1(self.maxpool(F.relu(self.conv13(x))))
```

```
                x = self.dropout12(self.bn_2(self.maxpool(F.relu(self.conv2(x)))))
                x = self.bn_3(self.maxpool(F.relu(self.conv3(x))))
                x = self.bn_4(self.maxpool(F.relu(self.conv4(x))))
                x = self.dropout12(self.bn_5(self.maxpool(F.relu(self.conv5(x)))))

                x = x.view(-1, 256*7*7)
                x = self.bn1_1(F.relu(self.fc_1(x)))
                x = self.bn1_2(F.relu(self.fc_2(x)))
                x = self.bn1_3(F.relu(self.fc_3(x)))
                x = F.log_softmax(self.fc_4(x), dim=1)
                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()
```

```
In [18]: model_scratch
```

```
Out[18]: Net(
           (conv11): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv13): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           (bn_1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
           (bn_2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
           (bn_3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tru
           (bn_4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
           (bn_5): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
           (fc_1): Linear(in_features=12544, out_features=1024, bias=True)
           (fc_2): Linear(in_features=1024, out_features=512, bias=True)
           (fc_3): Linear(in_features=512, out_features=64, bias=True)
           (fc_4): Linear(in_features=64, out_features=133, bias=True)
           (dropout12): Dropout2d(p=0.5)
           (bn1_1): BatchNorm1d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
           (bn1_2): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
           (bn1_3): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=Tr
         )
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reason-

ing at each step.

Answer: - The first convolutional layer is defined to take 224 x 224 image having 3 input chanels as input and output 16 activation map and then Passed through a relu activation function. The second convolution layer is defined to take 16 activation maps as input and output 16 activation map and then Passed through a relu activation. For the first two convolution layers, Maxpooling layer was not used to retain the image spartial dimensions. - After that convoluttion layers are defined such a way that the dimension of the image is decreased and the activation map is increased and then convolution layers are passed through the batch normalisation to eliminate randomness and dropout is applied in some of the layers to prevent overfitting. - The output of last convolution layers are flattened and then passed throgh three fully connected layers and passed throgh relu activation function and batch normalisation. - The output of above mentioned fully connected layers are passed through fully connected layer having Log_softmax activation function for classification.

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 [19]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

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 [20]: 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()
            optimizer.zero_grad()
            logp = model.forward(data)
            loss = criterion(logp, target)
            loss.backward()
            optimizer.step()
            ## 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_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()
            logp = model.forward(data)
            loss = criterion(logp, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss))
            ## update the average validation 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)
            valid_loss_min = valid_loss
            print('........initiating save mode')
    # return trained model
    return model


# train the model
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.807509         Validation Loss: 4.687277
```

```
...initiating save mode
Epoch: 2         Training Loss: 4.471549         Validation Loss: 4.404094
...initiating save mode
Epoch: 3         Training Loss: 4.311878         Validation Loss: 4.426095
Epoch: 4         Training Loss: 4.193388         Validation Loss: 4.248815
...initiating save mode
Epoch: 5         Training Loss: 4.075240         Validation Loss: 4.127416
...initiating save mode
Epoch: 6         Training Loss: 3.998126         Validation Loss: 4.019804
...initiating save mode
Epoch: 7         Training Loss: 3.889716         Validation Loss: 4.318954
Epoch: 8         Training Loss: 3.815179         Validation Loss: 3.881942
...initiating save mode
Epoch: 9         Training Loss: 3.740013         Validation Loss: 3.858064
...initiating save mode
Epoch: 10         Training Loss: 3.647617         Validation Loss: 3.897732
Epoch: 11         Training Loss: 3.564791         Validation Loss: 3.701886
...initiating save mode
Epoch: 12         Training Loss: 3.450893         Validation Loss: 3.886198
Epoch: 13         Training Loss: 3.383608         Validation Loss: 3.572875
...initiating save mode
Epoch: 14         Training Loss: 3.253147         Validation Loss: 3.596771
Epoch: 15         Training Loss: 3.168374         Validation Loss: 3.429441
...initiating save mode
Epoch: 16         Training Loss: 3.052519         Validation Loss: 3.562726
Epoch: 17         Training Loss: 2.943792         Validation Loss: 3.418638
...initiating save mode
Epoch: 18         Training Loss: 2.837054         Validation Loss: 3.274237
...initiating save mode
Epoch: 19         Training Loss: 2.744966         Validation Loss: 3.198236
...initiating save mode
Epoch: 20         Training Loss: 2.621045         Validation Loss: 3.225266
Epoch: 21         Training Loss: 2.522975         Validation Loss: 3.151161
...initiating save mode
Epoch: 22         Training Loss: 2.412591         Validation Loss: 3.069539
...initiating save mode
Epoch: 23         Training Loss: 2.310749         Validation Loss: 2.995447
...initiating save mode
Epoch: 24         Training Loss: 2.181733         Validation Loss: 3.067174
Epoch: 25         Training Loss: 2.071960         Validation Loss: 3.049443
```

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 [21]: 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: 2.984463


Test Accuracy: 27% (231/836)
```

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 [22]: ## TODO: Specify data loaders

         dir_test = os.path.join("/data/dog_images", "test")
         dir_valid = os.path.join("/data/dog_images", "valid")
         dir_train = os.path.join("/data/dog_images", "train")
         train_transforms_dog = transforms.Compose([transforms.RandomRotation(30),
                                                    transforms.Resize(225),
                                                    transforms.CenterCrop(224),
                                                    transforms.RandomHorizontalFlip(),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize((0.5,),(0.5,))])

         valid_transforms_dog = transforms.Compose([transforms.Resize(255),
                                                    transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize((0.5,),(0.5,))])
         train_data_dog = datasets.ImageFolder(dir_train, transform=train_transforms_dog)
         valid_data_dog = datasets.ImageFolder(dir_valid, transform=valid_transforms_dog)
         test_data_dog = datasets.ImageFolder(dir_test, transform=valid_transforms_dog)

         train_loader_dog = torch.utils.data.DataLoader(train_data_dog, batch_size=40, shuffle=T
         valid_loader_dog = torch.utils.data.DataLoader(valid_data_dog, batch_size=40)
         test_loader_dog = torch.utils.data.DataLoader(test_data_dog, batch_size=40)

         loaders_transfer={'train':train_loader_dog,
                           'valid': valid_loader_dog,
                           'test': test_loader_dog}
```

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

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)
         for params in model_transfer.parameters():
             params.require_grad = True
         model_transfer.fc = nn.Linear(2048, 133)

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 72648100.20it/s]
```

```
In [24]: model_transfer
```

```
Out[24]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
             (2): Bottleneck(
               (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
             )
           )
           (layer2): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
    (relu): ReLU(inplace)
    (downsample): Sequential(
```

```
    (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  )
)
(1): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(2): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(3): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(4): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(5): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
```

```
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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: - Resnet50 is used because it is relatively fast to train - Resnet50 pretrained weights are used because it was trained on similar deta as the train set - In final classification layers only the last layer is changed to a dense linear with the same number of nodes as the number of classes of dogs which we are trying to predict. That is the part of the network which I am going to re-train so that it is able to classify the dog breeds. - This architecture is suitable beacuse the traing data is relative large and has similar data to what the original pretrained model was trained on. - The weights are Randomly initialized in the new fully connected layer and in the rest of the network,

the weights are initialized using the pre-trained weights and re-train the entire neural network

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 [25]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.parameters(), lr=0.01)
```

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 [26]: # train the model
         n_epochs = 15
         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: 3.808256         Validation Loss: 2.297208
...initiating save mode
Epoch: 2         Training Loss: 1.895583         Validation Loss: 1.200905
...initiating save mode
Epoch: 3         Training Loss: 1.139708         Validation Loss: 0.815288
...initiating save mode
Epoch: 4         Training Loss: 0.816437         Validation Loss: 0.647269
...initiating save mode
Epoch: 5         Training Loss: 0.632947         Validation Loss: 0.538353
...initiating save mode
Epoch: 6         Training Loss: 0.511475         Validation Loss: 0.471149
...initiating save mode
Epoch: 7         Training Loss: 0.436376         Validation Loss: 0.435332
...initiating save mode
Epoch: 8         Training Loss: 0.365064         Validation Loss: 0.418808
...initiating save mode
Epoch: 9         Training Loss: 0.319853         Validation Loss: 0.391794
...initiating save mode
Epoch: 10         Training Loss: 0.279545         Validation Loss: 0.391162
...initiating save mode
Epoch: 11         Training Loss: 0.247332         Validation Loss: 0.361073
...initiating save mode
Epoch: 12         Training Loss: 0.224221         Validation Loss: 0.354447
...initiating save mode
Epoch: 13         Training Loss: 0.196717         Validation Loss: 0.371353
Epoch: 14         Training Loss: 0.170017         Validation Loss: 0.333706
...initiating save mode
```

```
Epoch: 15         Training Loss: 0.153742         Validation Loss: 0.327975
...initiating save mode
```

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 [27]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

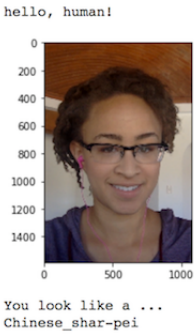
```
Test Loss: 0.337666


Test Accuracy: 89% (748/836)
```

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 [28]:
```
data_transfer = loaders_transfer
```

In [29]:
```
### 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]
from torch.autograd import Variable
class_names = [item[4:].replace("_", " ") for item in train_data_dog.classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breeds for that image
    image = Image.open(img_path).convert('RGB')

    transformer = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize((0.5,),(0.5,))])

    img_processed = transformer(image).unsqueeze(0)

    if use_cuda:
        img_processed = img_processed.to('cuda')
    img_var = Variable(img_processed, requires_grad= False)
    img_var = img_var.cuda()
    model_transfer.eval()
    output = model_transfer(img_var)
    _, pred = torch.max(output, 1)
    return class_names[pred]
```



Sample Human Output

```
prediction = predict_breed_transfer(dog_files[1002])
print(prediction)
```

```
Manchester terrier
```

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 [30]:
```
### 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
    image = Image.open(img_path).convert('RGB')
    plt.imshow(image)
    plt.show()
    if dog_detector(img_path) == True:
        print('Dog found! Predicted breed is: ' + predict_breed_transfer(img_path))

    elif face_detector(img_path) == True:
        print("Human found! Most similar breed is: " + predict_breed_transfer(img_path)
```

```
    else:
        print('No human or dog was found')
```

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?

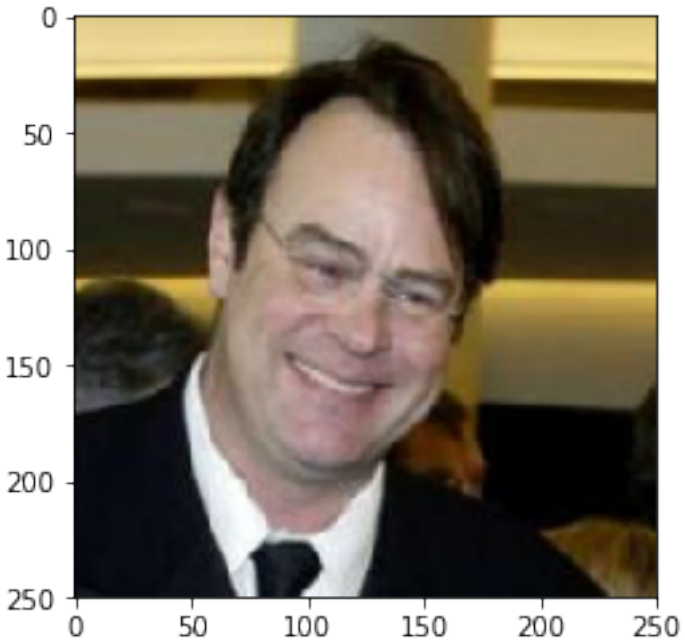
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

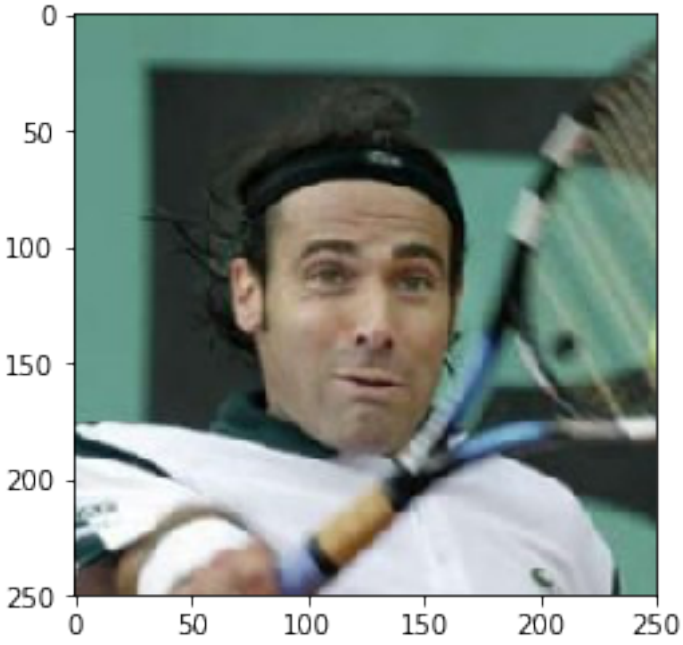
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: - The breed detector could be improved by changing the classifier network to increase accuracy. - Some training dataset for some dog breed have very less images which can be added with more images to get better accuracy - Train the predictor to handle images with both human and dog

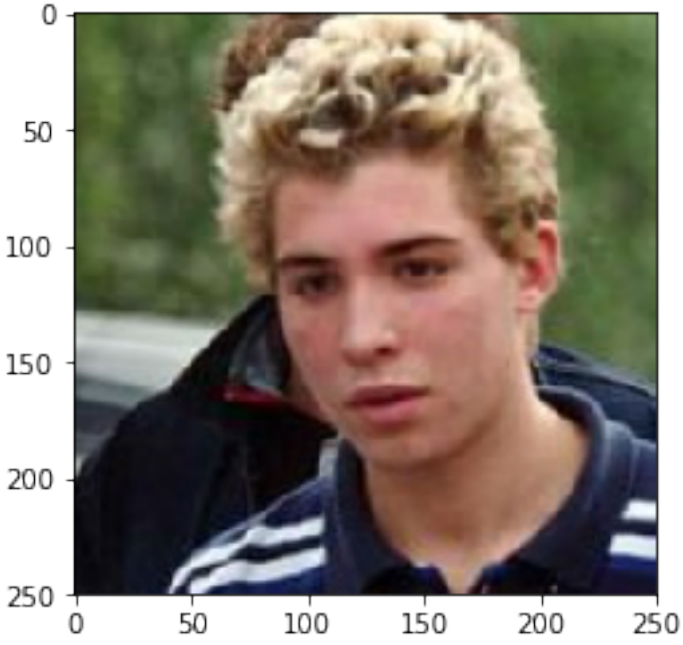
In [31]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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



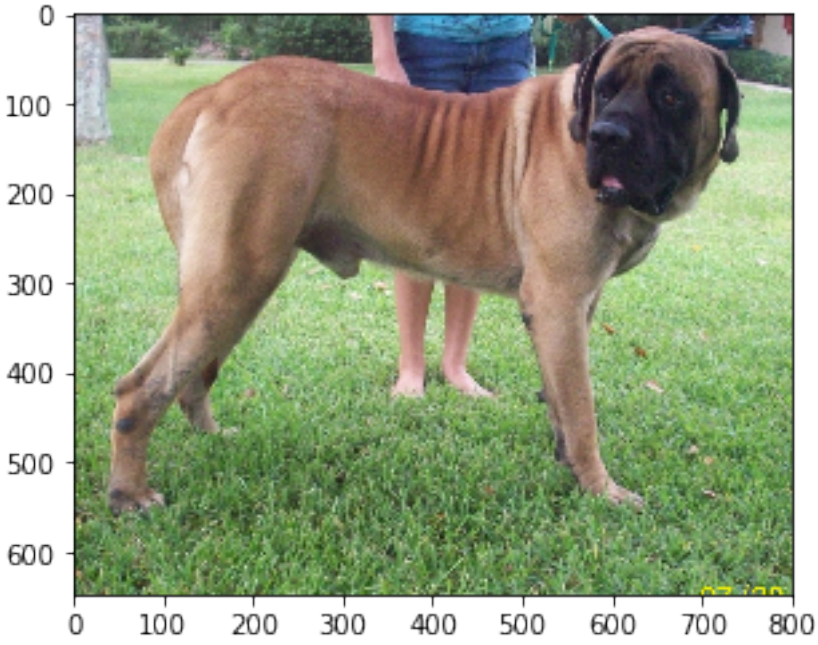
```
Human found! Most similar breed is: Chihuahua
```



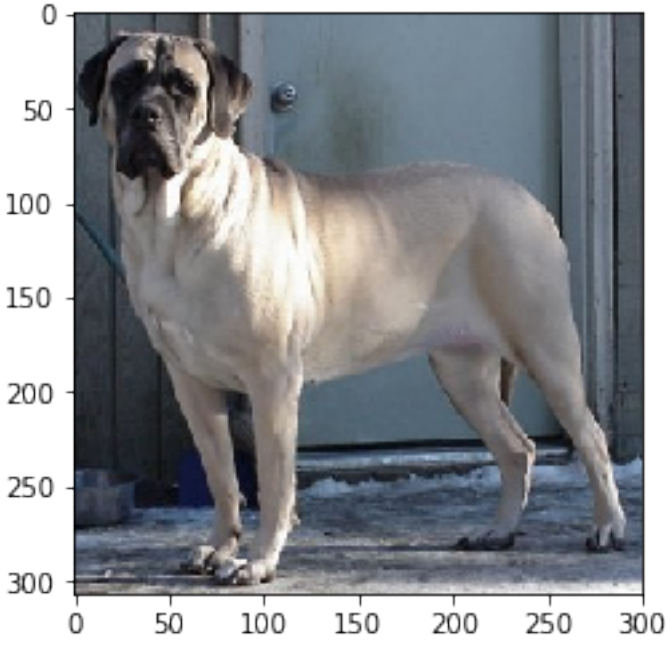

```
Human found! Most similar breed is: Neapolitan mastiff
```



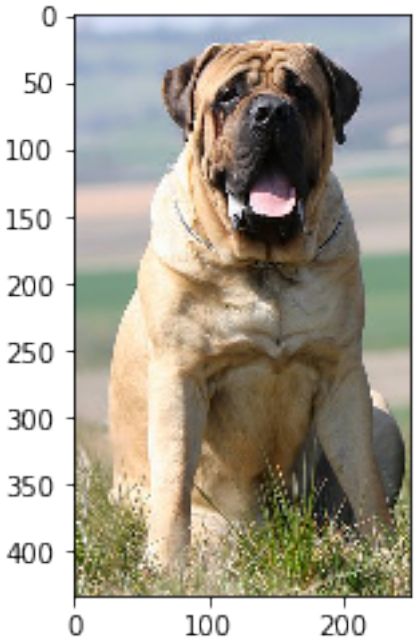
```
Human found! Most similar breed is: American water spaniel
```




```
Dog found! Predicted breed is: Mastiff
```



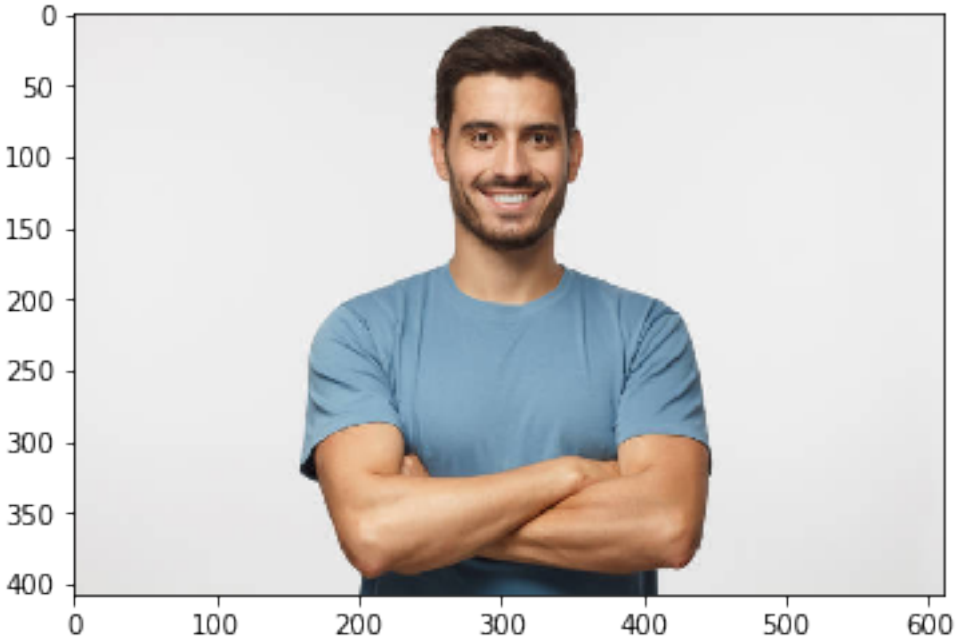
```
Dog found! Predicted breed is: Mastiff
```



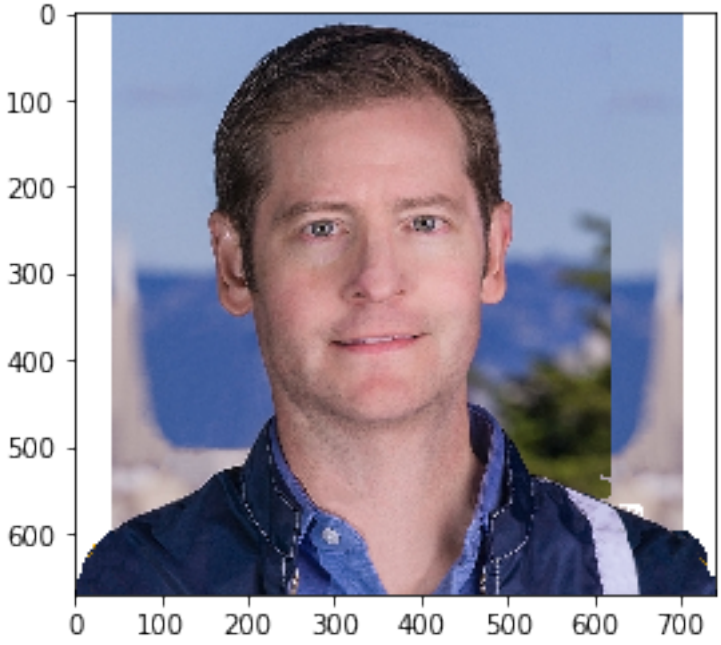

```
Dog found! Predicted breed is: Mastiff
```

```
In [37]: my_human_files = ['./my_images/human3.jpg', './my_images/Accenture-Human-Machine-AI-Jam
         my_dog_files = ['./my_images/American_water_spaniel_00648-Copy1.jpg', './my_images/Brit
```

```
In [38]: for file in np.hstack((my_human_files, my_dog_files)):
             run_app(file)
```




```
Human found! Most similar breed is: Chinese shar-pei
```

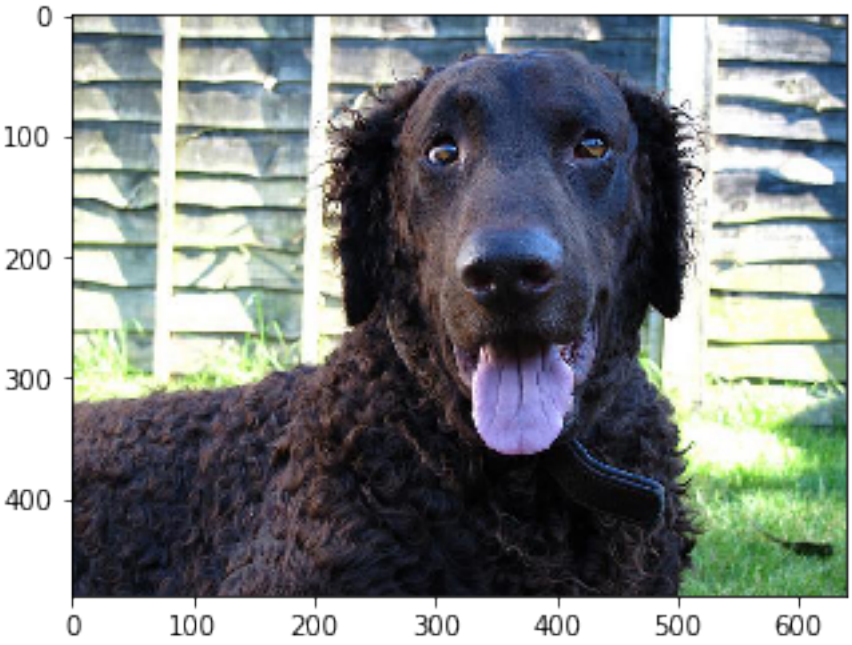


```
Human found! Most similar breed is: Pointer
```


```
Human found! Most similar breed is: Dogue de bordeaux
```

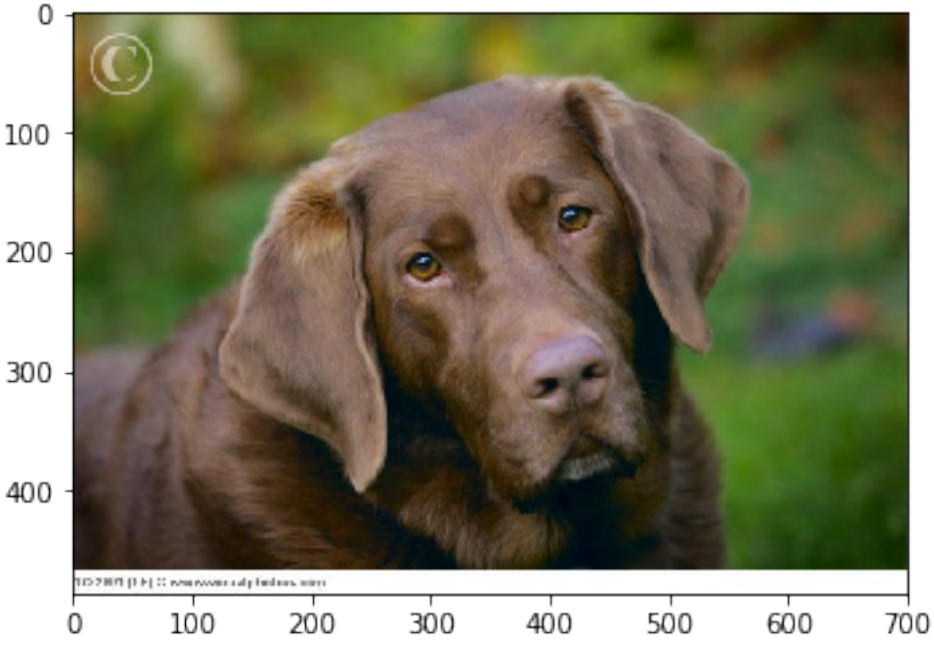
```
Dog found! Predicted breed is: Curly-coated retriever
```


```
Dog found! Predicted breed is: Brittany
```

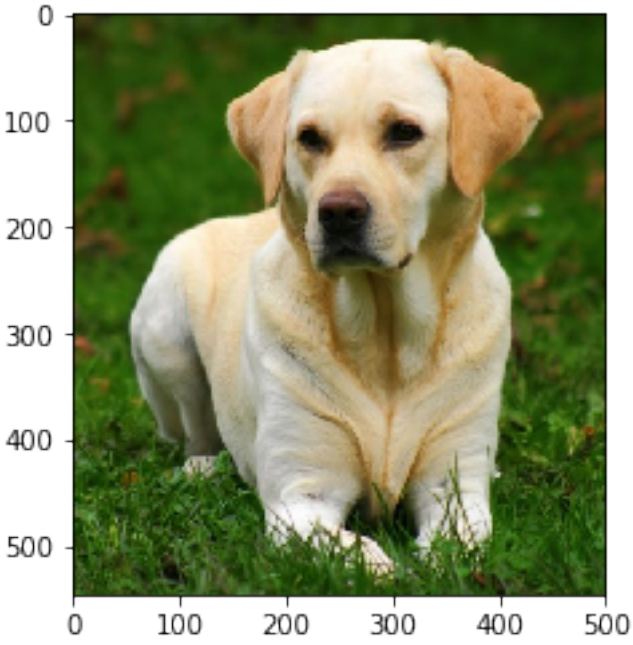


```
Dog found! Predicted breed is: Curly-coated retriever
```

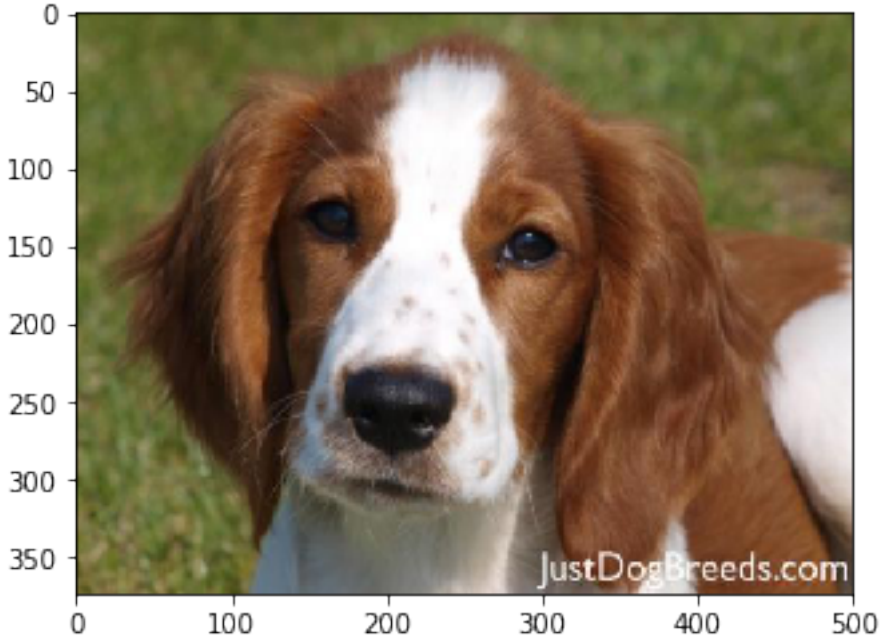

```
Dog found! Predicted breed is: Labrador retriever
```



```
Dog found! Predicted breed is: Labrador retriever
```




```
Dog found! Predicted breed is: Labrador retriever
```



```
Dog found! Predicted breed is: Welsh springer spaniel
```

In [ ]: