dog_app

October 28, 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 [2]: 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.
```

1.2 Explore and Visualize the Datasets

Human images are distributed in 5749 folders named after human names such as "Jonathan Tiomkin", "Charlie Williams", etc. There are 13233 human face images. Images are not evenly distributed among the folders.

The dog dataset is structured into a training set, a validation set, and a test set. Each set comprises 133 breeds of dogs organized into subdirectories by breed name. Later, we will see that the number of images per dog breed is imbalanced. There are 6680 dog images in the training set, 835 images in the validation set, and 836 images in the test set.

```
In [3]: import os

        # Data Exploration
        print('Number of folders in /lfw:', len(glob('/data/lfw/*')))
        print('Number of human images:',len(glob("/data/lfw/*/*")))
        print('Numner of folders in /dog_images:', len(glob('/data/dog_images/*')))
        print('Folder names in /dog_images:', end=' ')
        print(*[x.split('/')[-1] for x in glob('/data/dog_images/*')], sep=', ')
        print('Number of folders (breed classes) in /dog_images/train:', len(glob("/data/dog_ima
        print('Number of images in /dog_images/train:', len(glob("/data/dog_images/train/*/*")))
        print('Number of folders (breed classes) in /dog_images/valid:', len(glob("/data/dog_ima
        print('Number of images in /dog_images/valid:', len(glob("/data/dog_images/valid/*/*")))
        print('Number of folders (breed classes) in /dog_images/test:', len(glob("/data/dog_imag
        print('Number of images in /dog_images/test:', len(glob("/data/dog_images/test/*/*")))
```

```
Number of folders in /lfw: 5749
Number of human images: 13233
Numner of folders in /dog_images: 3
Folder names in /dog_images: train, test, valid
Number of folders (breed classes) in /dog_images/train: 133
Number of images in /dog_images/train: 6680
Number of folders (breed classes) in /dog_images/valid: 133
Number of images in /dog_images/valid: 835
Number of folders (breed classes) in /dog_images/test: 133
Number of images in /dog_images/test: 836
```

The dog dataset contains 133 classes. Let print out the first 10 classes.

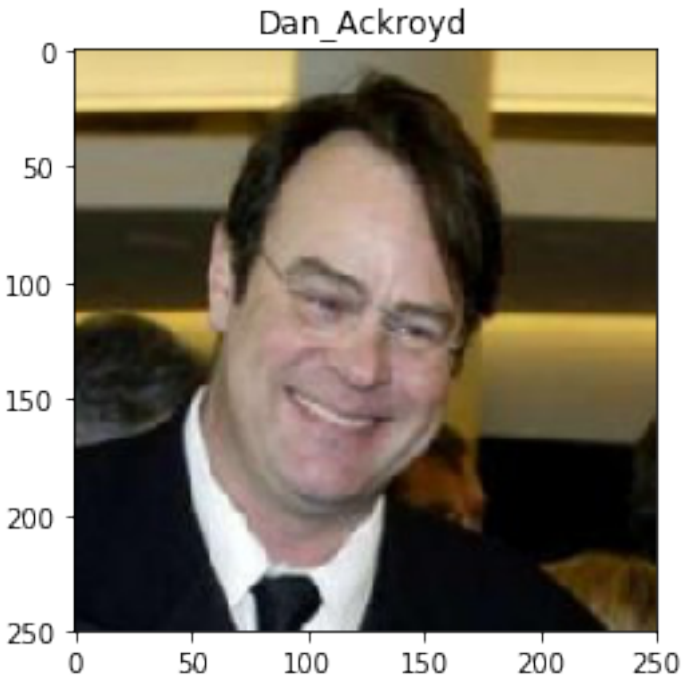
```
In [4]: classes = sorted([x.split('/')[-1] for x in glob("/data/dog_images/train/*")])
        print(classes[:10])
```

```
['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mala
```

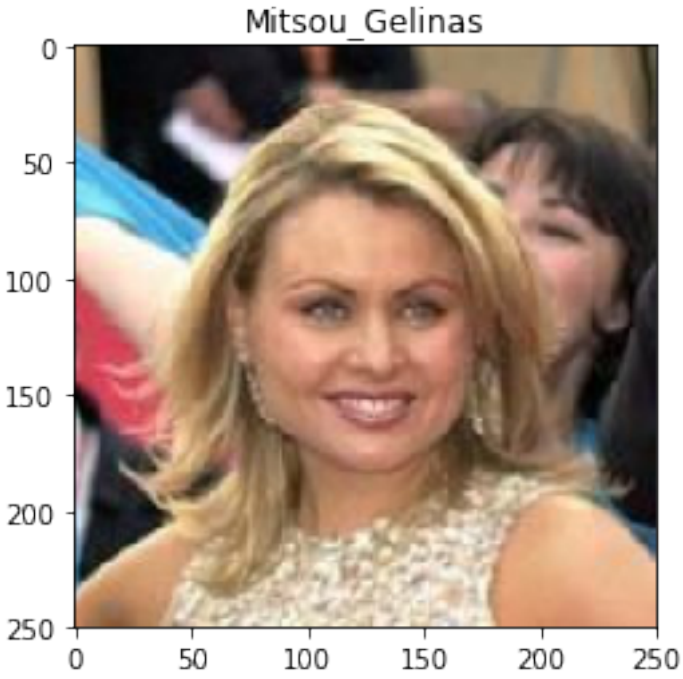
Let's look at a couple of images from the human and the dog datasets.

```
In [5]: from PIL import Image
        import matplotlib.pyplot as plt
        %matplotlib inline

        img = Image.open(human_files[0])
        title = human_files[0].split('/')[-2]
        plt.title(title)
        plt.imshow(img)
```

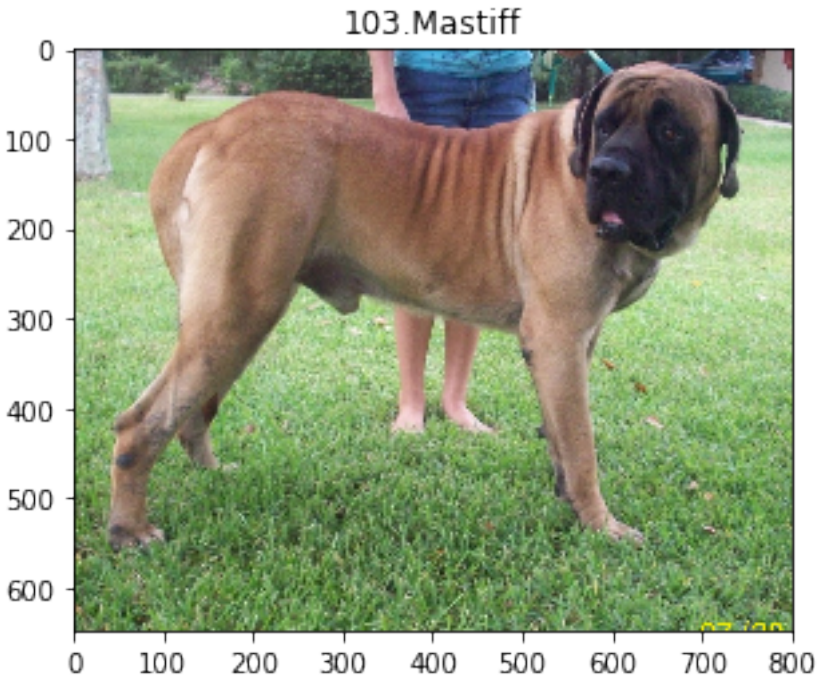
```
Out[5]: <matplotlib.image.AxesImage at 0x7f1122168400>
```



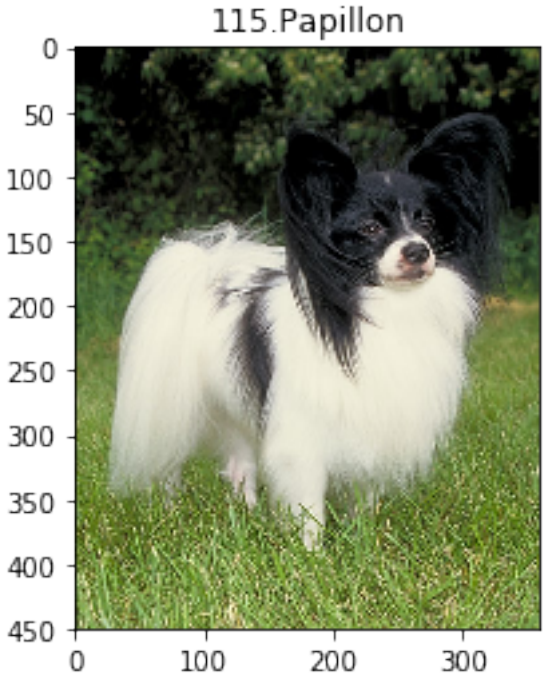

```
In [6]: img = Image.open(human_files[100])
        title = human_files[100].split('/')[-2]
        plt.title(title)
        plt.imshow(img)

Out[6]: <matplotlib.image.AxesImage at 0x7f112210aa58>
```




```
In [7]: img = Image.open(dog_files[0])
        title = dog_files[0].split('/')[-2]
        plt.title(title)
        plt.imshow(img)

Out[7]: <matplotlib.image.AxesImage at 0x7f112086b7f0>
```



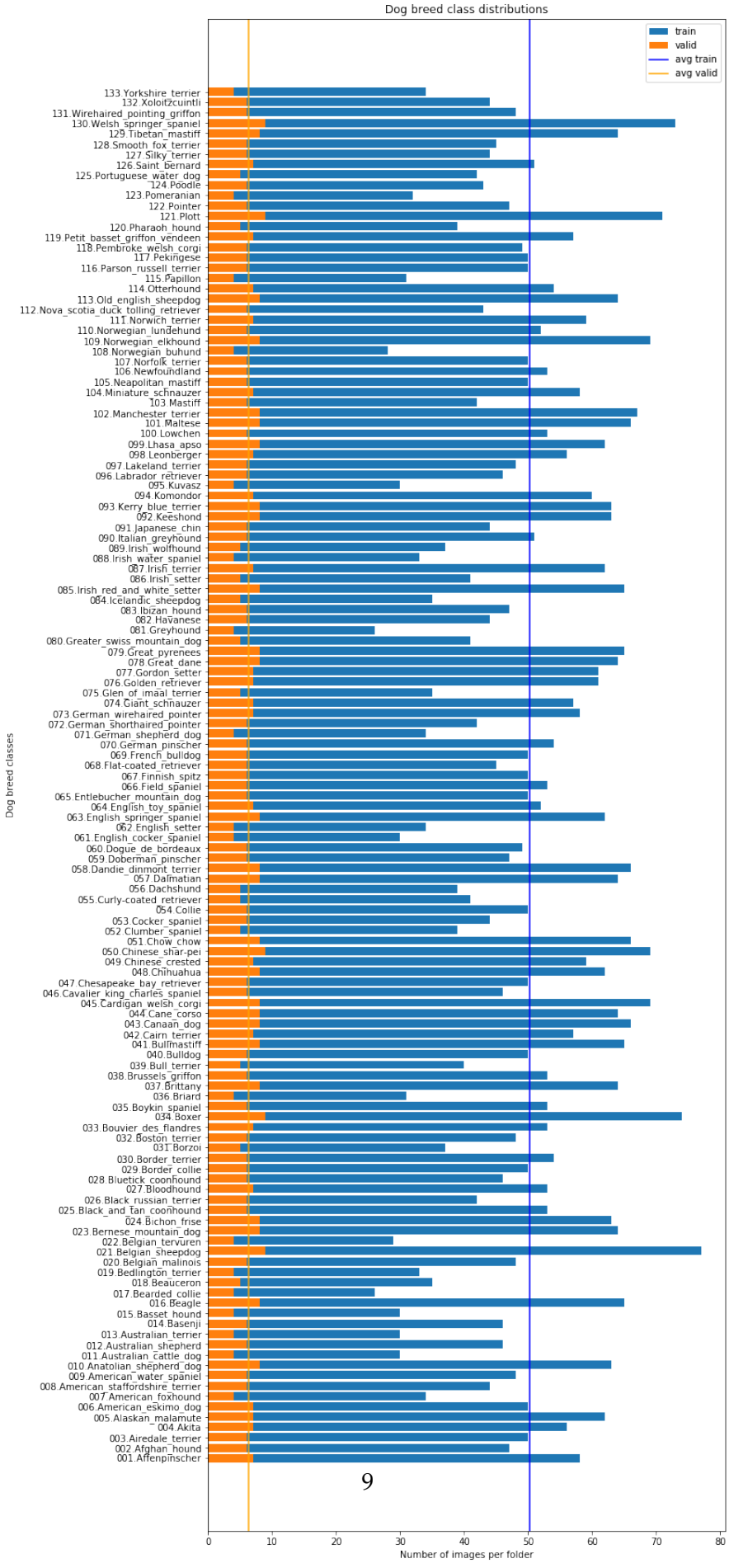

```
In [8]: img = Image.open(dog_files[500])
        title = dog_files[500].split('/')[-2]
        plt.title(title)
        plt.imshow(img)

Out[8]: <matplotlib.image.AxesImage at 0x7f11207d6208>
```



The dog dataset is imbalanced. This observation is critical to use other metrics instead of accuracy that distinguishes correctly between the numbers of correctly classified examples of different classes i.e., precision and recall.

```
In [9]: num_images_per_folder = {}
        num_images_per_folder['train'] = [len(glob(x+'/*')) for x in glob("/data/dog_images/trai
        num_images_per_folder['valid'] = [len(glob(x+'/*')) for x in glob("/data/dog_images/vali
        num_images_per_folder['test'] = [len(glob(x+'/*')) for x in glob("/data/dog_images/test/
        num_images_avg = {}
        num_images_avg['train'] = sum(num_images_per_folder['train'])/len(num_images_per_folder[
        num_images_avg['valid'] = sum(num_images_per_folder['valid'])/len(num_images_per_folder[
        num_images_avg['test'] = sum(num_images_per_folder['test'])/len(num_images_per_folder['t

In [10]: print('Average number of images per dog breed in traning set is', num_images_avg['train
         print('Average number of images per dog breed in validation set is', num_images_avg['va
         print('Average number of images per dog breed in test set is', num_images_avg['test'])

Average number of images per dog breed in traning set is 50.225563909774436
Average number of images per dog breed in validation set is 6.2781954887218046
Average number of images per dog breed in test set is 6.285714285714286


In [11]: import matplotlib.pyplot as plt
         %matplotlib inline

         plt.figure(figsize=(10, 30))
```

```
bar1 = plt.barh(classes, num_images_per_folder['train'])
#plt.barh(classes, num_images_per_folder['test'])
bar2 = plt.barh(classes, num_images_per_folder['valid'])
line1 = plt.axvline(num_images_avg['train'], color='blue')
line2 = plt.axvline(num_images_avg['valid'], color='orange')
plt.ylabel('Dog breed classes')
plt.xlabel('Number of images per folder')
plt.title('Dog breed class distributions')
plt.legend((bar1, bar2, line1, line2), ('train', 'valid', 'avg train', 'avg valid'))
plt.show()
```


Dog breed class distributions
train
valid
avg train
avg valid
133.Yorkshire_terrier
132.Xoloitzcuintli
131.Wirehaired_pointing_griffon
130.Welsh_springer_spaniel
129.Tibetan_mastiff
128.Smooth_fox_terrier
127.Silky_terrier
126.Saint_bernard
125.Portuguese_water_dog
124.Poodle
123.Pomeranian
122.Pointer
121.Plott
120.Pharaoh_hound
119.Petit_basset_griffon_vendeen
118.Pembroke_welsh_corgi
117.Pekingese
116.Parson_russell_terrier
115.Papillon
114.Otterhound
113.Old_english_sheepdog
112.Nova_scotia_duck_tolling_retriever
111.Norwich_terrier
110.Norwegian_lundehund
109.Norwegian_elkhound
108.Norwegian_buhund
107.Norfolk_terrier
106.Newfoundland
105.Neapolitan_mastiff
104.Miniature_schnauzer
103.Mastiff
102.Manchester_terrier
101.Maltese
100.Lowchen
099.Lhasa_apso
098.Leonberger
097.Lakeland_terrier
096.Labrador_retriever
095.Kuvasz
094.Komondor
093.Kerry_blue_terrier
092.Keeshond
091.Japanese_chin
090.Italian_greyhound
089.Irish_wolfhound
088.Irish_water_spaniel
087.Irish_terrier
086.Irish_setter
085.Irish_red_and_white_setter
084.Icelandic_sheepdog
083.Ibizan_hound
082.Havanese
081.Greyhound
080.Greater_swiss_mountain_dog
079.Great_pyrenees
078.Great_dane
077.Gordon_setter
076.Golden_retriever
075.Glen_of_imaal_terrier
074.Giant_schnauzer
073.German_wirehaired_pointer
072.German_shorthaired_pointer
071.German_shepherd_dog
070.German_pinscher
069.French_bulldog
068.Flat-coated_retriever
067.Finnish_spitz
066.Field_spaniel
065.Entlebucher_mountain_dog
064.English_toy_spaniel
063.English_springer_spaniel
062.English_setter
061.English_cocker_spaniel
060.Dogue_de_bordeaux
059.Doberman_pinscher
058.Dandie_dinmont_terrier
057.Dalmatian
056.Dachshund
055.Curly-coated_retriever
054.Collie
053.Cocker_spaniel
052.Clumber_spaniel
051.Chow_chow
050.Chinese_shar-pei
049.Chinese_crested
048.Chihuahua
047.Chesapeake_bay_retriever
046.Cavalier_king_charles_spaniel
045.Cardigan_welsh_corgi
044.Cane_corso
043.Canaan_dog
042.Cairn_terrier
041.Bullmastiff
040.Bulldog
039.Bull_terrier
038.Brussels_griffon
037.Brittany
036.Briard
035.Boykin_spaniel
034.Boxer
033.Bouvier_des_flandres
032.Boston_terrier
031.Borzoi
030.Border_terrier
029.Border_collie
028.Bluetick_coonhound
027.Bloodhound
026.Black_russian_terrier
025.Black_and_tan_coonhound
024.Bichon_frise
023.Bernese_mountain_dog
022.Belgian_tervuren
021.Belgian_sheepdog
020.Belgian_malinois
019.Bedlington_terrier
018.Beauceron
017.Bearded_collie
016.Beagle
015.Basset_hound
014.Basenji
013.Australian_terrier
012.Australian_shepherd
011.Australian_cattle_dog
010.Anatolian_shepherd_dog
009.American_water_spaniel
008.American_staffordshire_terrier
007.American_foxhound
006.American_eskimo_dog
005.Alaskan_malamute
004.Akita
003.Airedale_terrier
002.Afghan_hound
001.Affenpinscher
Dog breed classes
0
10
20
30
40
50
60
70
80
Number of images per folder

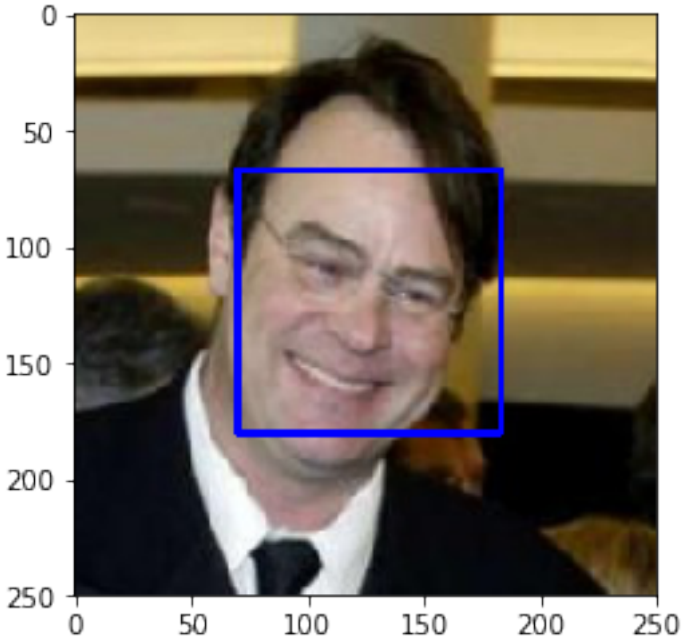
Step 1: Detect Humans

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 [12]: 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.

1.2.1 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 [13]: # 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
```

1.2.2 (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: (You can print out your results and/or write your percentages in this cell)
98 of the first 100 images in `human_files` or 98% have a detected human face.
17 of the first 100 images in `dog_files` or 17% have a detected human face.

In [14]:
```
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_count = sum([face_detector(human) for human in human_files_short])
dog_count = sum([face_detector(dog) for dog in dog_files_short])

print('{} of the first 100 images in human_files have a detected human face.'.format(hu
print('{} of the first 100 images in dog_files have a detected human face.'.format(dog_
```

```
98 of the first 100 images in human_files have a detected human face.
17 of the first 100 images in dog_files have a detected human face.
```

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 [15]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.2.3 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 [16]: 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, 107552511.85it/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.

1.2.4 (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 [17]: from PIL import Image
         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
             transform = transforms.Compose([
                 transforms.Resize(256),
```

```
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ])

    img = Image.open(img_path).convert('RGB')
    img = transform(img)
    img = img.unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    VGG16.eval()
    output = VGG16(img)

    return torch.max(output,1)[1].item() # predicted class index
```

1.2.5 (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 [18]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             index = VGG16_predict(img_path)
             return index >= 151 and index <= 268 # true/false
```

1.2.6 (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 detected dog on `human_files_short` is 0.0%
Percentage of detected dog on `dog_files_short` is 100.0%

```
In [19]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_count = 0
         dog_count = 0

         for i in human_files_short:
             if dog_detector(i):
```

```
            human_count+=1
    for i in dog_files_short:
        if dog_detector(i):
            dog_count+=1

    print("Percentage of detected dog on human_files_short is {}%".format(100*human_count/1
    print("Percentage of detected dog on dog_files_short is {}%".format(100*dog_count/100.0
```

```
Percentage of detected dog on human_files_short is 0.0%
Percentage of detected dog on dog_files_short is 100.0%
```

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 [20]: ### (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!

1.2.7 (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 [21]: import os
         from torchvision import datasets

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes

         # Define batch size
         batch_size = 8
         num_workers = 0

         # directories
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         valid_dir = os.path.join(data_dir, 'valid/')
         test_dir = os.path.join(data_dir, 'test/')

         # pre-process data
         data_transforms = {'train': transforms.Compose([transforms.Resize(256),
                                                         transforms.RandomHorizontalFlip(),
                                                         transforms.RandomRotation(10),
                                                         transforms.CenterCrop(224),
                                                         transforms.ToTensor(),
                                                         transforms.Normalize((0.5, 0.5, 0.5), (

                                'valid': transforms.Compose([transforms.Resize(256),
                                                         transforms.CenterCrop(224),
                                                         transforms.ToTensor(),
```

```
                                                transforms.Normalize((0.5, 0.5, 0.5), (

                        'test': transforms.Compose([transforms.Resize(256),
                                                    transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize((0.5, 0.5, 0.5), (0
                    }

        train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
        valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['valid'])
        test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
        valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worke
        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers

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

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

Answer:

In the original VGG-16 paper, the authors chosed fixed-size 224x224 images as the input. Thus I rescaled the original images to 256x256 before cropping to get 224x224 images. The image rescaling is critical since cropping a 224x224 image out of a much larger original is unlikely to contain the features we are interested in to train the model.

Data augmentation is a way to extend a dataset and improve generalization to avoid overfitting, therefore, I augmented the dataset using random horizontal flipping and random rotation up to 10 degrees. The data augmentation is performed on training set only.

1.2.8 (IMPLEMENTATION) Model Architecture

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

```
In [22]: 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 layer (224x224x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1)
                 # convolutional layer (56x56x16 image tensor)
                 self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=2, padding=1)
```

```
            # convolutional layer (14x14x32 image tensor)
            self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
            # max pooling layer
            self.pool = nn.MaxPool2d(2, 2)
            # linear layer (64 *7 * 7 -> 512)
            self.fc1 = nn.Linear(64 * 7 * 7, 512)
            # linear layer (512 -> 512)
            self.fc2 = nn.Linear(512, 512)
            # linear layer (512 -> 133)
            self.fc3 = nn.Linear(512, 133)
            # dropout layer (p=0.25)
            self.dropout = nn.Dropout(0.25)

        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            # flatten image input
            x = x.view(-1, 64 * 7 * 7)
            # add dropout layer
            x = self.dropout(x)
            # add 1st hidden layer, with relu activation function
            x = F.relu(self.fc1(x))
            # add dropout layer
            x = self.dropout(x)
            # add 2nd hidden layer, with relu activation function
            x = self.fc2(x)
             # add dropout layer
            x = self.dropout(x)
            # add 3rd hidden layer, with relu activation function
            x = self.fc3(x)

            return x

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

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

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
```

```
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=3136, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

```
In [23]: !pip install torchsummary
         from torchsummary import summary
         summary(model_scratch, (3, 224, 224))
```

```
Collecting torchsummary
  Downloading https://files.pythonhosted.org/packages/7d/18/1474d06f721b86e6a9b9d7392ad68bed711a
Installing collected packages: torchsummary
Successfully installed torchsummary-1.5.1
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 112, 112]             448
         MaxPool2d-2           [-1, 16, 56, 56]               0
            Conv2d-3           [-1, 32, 28, 28]           4,640
         MaxPool2d-4           [-1, 32, 14, 14]               0
            Conv2d-5           [-1, 64, 14, 14]          18,496
         MaxPool2d-6             [-1, 64, 7, 7]               0
           Dropout-7                 [-1, 3136]               0
            Linear-8                  [-1, 512]       1,606,144
           Dropout-9                  [-1, 512]               0
           Linear-10                  [-1, 512]         262,656
          Dropout-11                  [-1, 512]               0
           Linear-12                  [-1, 133]          68,229
================================================================
Total params: 1,960,613
Trainable params: 1,960,613
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 2.31
Params size (MB): 7.48
Estimated Total Size (MB): 10.37
----------------------------------------------------------------
```

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

Answer:

The scratch model is structured as follows:

3 convolutional (conv) layers transform a 3-channel image to a 64-channel feature map. All conv layers have a kernel size of 3 and a padding of 1 (to preserve spatial resolution after convolution). The first 2 conv layers have stride 2 while the third one has stride 1.

Max-pooling is performed over a 2 x 2-pixel window, with stride 2 after conv layers which halves the height and width. The feature map gets smaller as we add more layers until we are finally left with a small feature map, which can be flattened into a vector.

3 fully connected layers at the end produce an output vector of size 133 for each image.

3 dropout layers of 0.25 are added to avoid overfitting.

All hidden layers are equipped with the rectification (ReLU) non-linearity.

1.2.9 (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 [24]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.02)
# optimizer_scratch = optim.Adam(model_scratch.parameters(), lr = 0.01)
```

1.2.10 (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 [14]:
```
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: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update average 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: compute predicted outputs by passing inputs to the model
            with torch.no_grad():
                output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update 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:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
        # train the model
        model_scratch = train(15, 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: 4.885879         Validation Loss: 4.874210
Validation loss decreased (inf --> 4.874210).  Saving model ...
Epoch: 2         Training Loss: 4.866884         Validation Loss: 4.825825
Validation loss decreased (4.874210 --> 4.825825).  Saving model ...
Epoch: 3         Training Loss: 4.694588         Validation Loss: 4.615354
Validation loss decreased (4.825825 --> 4.615354).  Saving model ...
Epoch: 4         Training Loss: 4.531321         Validation Loss: 4.465499
Validation loss decreased (4.615354 --> 4.465499).  Saving model ...
Epoch: 5         Training Loss: 4.371618         Validation Loss: 4.374201
Validation loss decreased (4.465499 --> 4.374201).  Saving model ...
Epoch: 6         Training Loss: 4.265738         Validation Loss: 4.324693
Validation loss decreased (4.374201 --> 4.324693).  Saving model ...
Epoch: 7         Training Loss: 4.158158         Validation Loss: 4.183197
Validation loss decreased (4.324693 --> 4.183197).  Saving model ...
Epoch: 8         Training Loss: 4.071888         Validation Loss: 4.128459
Validation loss decreased (4.183197 --> 4.128459).  Saving model ...
Epoch: 9         Training Loss: 3.981011         Validation Loss: 4.072647
Validation loss decreased (4.128459 --> 4.072647).  Saving model ...
Epoch: 10         Training Loss: 3.893157         Validation Loss: 4.061979
Validation loss decreased (4.072647 --> 4.061979).  Saving model ...
Epoch: 11         Training Loss: 3.785850         Validation Loss: 3.952812
Validation loss decreased (4.061979 --> 3.952812).  Saving model ...
Epoch: 12         Training Loss: 3.697476         Validation Loss: 3.902146
Validation loss decreased (3.952812 --> 3.902146).  Saving model ...
Epoch: 13         Training Loss: 3.559261         Validation Loss: 3.842603
Validation loss decreased (3.902146 --> 3.842603).  Saving model ...
Epoch: 14         Training Loss: 3.480124         Validation Loss: 3.890282
Epoch: 15         Training Loss: 3.361239         Validation Loss: 3.780858
Validation loss decreased (3.842603 --> 3.780858).  Saving model ...
```

1.2.11 (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 [25]:
```
def test(loaders, model, criterion, use_cuda):

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

```
            correct = 0.
            total = 0.

            predicted_labels = []
            true_labels = []

            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]
                predicted_labels.append(pred)
                true_labels.append(target)
                # 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))

            return predicted_labels, true_labels
```

In [27]:
```
# call test function
predicted_labels, true_labels = test(loaders_scratch, model_scratch, criterion_scratch,
```

```
Test Loss: 3.773037


Test Accuracy: 13% (115/836)
```

In [28]:
```
predictions = []
for labels in predicted_labels:
    for label in np.array(labels):
        predictions.append(label[0])

ground_truths = []
for labels in true_labels:
```

```
            for label in np.array(labels):
                ground_truths.append(label)
```

```
In [29]: from sklearn.metrics import confusion_matrix, precision_recall_fscore_support
         precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
         recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
         print('Average test precision: {:.1f}%'.format(precision*100))
         print('Average test recall: {:.1f}%'.format(recall*100))
```

```
Average test precision: 12.2%
Average test recall: 12.3%


/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
```

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.

1.2.12 (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 [30]: ## TODO: Specify data loaders
         batch_size = 32

         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
         valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worke
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers

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

1.2.13 (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 [31]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
```

```
# Load the pretrained model from PyTorch
model_transfer = models.vgg16(pretrained=True)

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

n_inputs = model_transfer.classifier[6].in_features

# add last linear layer (n_inputs -> 133 dog breed classes)
# new layers automatically have requires_grad = True
model_transfer.classifier[6] = nn.Linear(n_inputs, 133)

# print out the model structure

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

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:

The CNN model built from scratch can be improved significantly by transfer learning. The idea is to reuse the lower layers of a pre-trained model for feature extraction and only train a few higher layers for classification. Modern CNNs training on huge datasets like ImageNet have great performance. Here I am interested in a couple of state-of-the-art pre-trained architectures VGGNet (Zisserman, 2015) and (K. He, 2016) as a fixed feature extractor for my models.

VGGNet is worth trying because it has a very simple and classical architecture and has great performance, coming in second in the ImageNet 2014 competition. VGGNet is already adopted in the dog detection task and thus can be reused. The idea here is that I keep all the convolutional layers, but replace the final fully-connected layer with my classifier. This way I can use VGGNet as a fixed feature extractor for the images then easily train a simple classifier on top of that.

1.2.14 (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 [32]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.2.15 (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 [19]: n_epochs = 20

         # train the model
         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: 4.433197         Validation Loss: 3.660359
Validation loss decreased (inf --> 3.660359).  Saving model ...
Epoch: 2         Training Loss: 3.068509         Validation Loss: 2.253841
Validation loss decreased (3.660359 --> 2.253841).  Saving model ...
Epoch: 3         Training Loss: 1.973345         Validation Loss: 1.419213
Validation loss decreased (2.253841 --> 1.419213).  Saving model ...
Epoch: 4         Training Loss: 1.433813         Validation Loss: 1.035404
Validation loss decreased (1.419213 --> 1.035404).  Saving model ...
Epoch: 5         Training Loss: 1.149600         Validation Loss: 0.854170
Validation loss decreased (1.035404 --> 0.854170).  Saving model ...
Epoch: 6         Training Loss: 1.004561         Validation Loss: 0.736856
Validation loss decreased (0.854170 --> 0.736856).  Saving model ...
Epoch: 7         Training Loss: 0.915746         Validation Loss: 0.695458
Validation loss decreased (0.736856 --> 0.695458).  Saving model ...
Epoch: 8         Training Loss: 0.825299         Validation Loss: 0.636218
Validation loss decreased (0.695458 --> 0.636218).  Saving model ...
Epoch: 9         Training Loss: 0.768361         Validation Loss: 0.605259
Validation loss decreased (0.636218 --> 0.605259).  Saving model ...
Epoch: 10         Training Loss: 0.729283         Validation Loss: 0.576355
Validation loss decreased (0.605259 --> 0.576355).  Saving model ...
Epoch: 11         Training Loss: 0.680769         Validation Loss: 0.567901
Validation loss decreased (0.576355 --> 0.567901).  Saving model ...
Epoch: 12         Training Loss: 0.657427         Validation Loss: 0.544684
Validation loss decreased (0.567901 --> 0.544684).  Saving model ...
Epoch: 13         Training Loss: 0.618902         Validation Loss: 0.539705
Validation loss decreased (0.544684 --> 0.539705).  Saving model ...
Epoch: 14         Training Loss: 0.591450         Validation Loss: 0.519682
Validation loss decreased (0.539705 --> 0.519682).  Saving model ...
Epoch: 15         Training Loss: 0.577838         Validation Loss: 0.509746
Validation loss decreased (0.519682 --> 0.509746).  Saving model ...
Epoch: 16         Training Loss: 0.557196         Validation Loss: 0.505338
Validation loss decreased (0.509746 --> 0.505338).  Saving model ...
Epoch: 17         Training Loss: 0.528556         Validation Loss: 0.495418
Validation loss decreased (0.505338 --> 0.495418).  Saving model ...
Epoch: 18         Training Loss: 0.504658         Validation Loss: 0.490786
Validation loss decreased (0.495418 --> 0.490786).  Saving model ...
Epoch: 19         Training Loss: 0.493439         Validation Loss: 0.484817
Validation loss decreased (0.490786 --> 0.484817).  Saving model ...
Epoch: 20         Training Loss: 0.494086         Validation Loss: 0.481286
Validation loss decreased (0.484817 --> 0.481286).  Saving model ...
```

1.2.16 (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 [34]: predicted_labels, true_labels = test(loaders_transfer, model_transfer, criterion_transf
```

```
Test Loss: 0.528622


Test Accuracy: 83% (701/836)
```

```
In [35]: predictions = []
         for labels in predicted_labels:
             for label in np.array(labels):
                 predictions.append(label[0])

         ground_truths = []
         for labels in true_labels:
             for label in np.array(labels):
                 ground_truths.append(label)

         precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
         recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
         print('Average test precision: {:.1f}%'.format(precision*100))
         print('Average test recall: {:.1f}%'.format(recall*100))
```

```
Average test precision: 85.0%
Average test recall: 82.9%
```

On the other hand, ResNet is the winner of the ImageNet 2015 competition and has amazing performance on image classification. It helps to train very deep networks by using skip connections (or shortcut connections). Specifically, it adds the original input back to the output feature map obtained bypassing the input through one or more conv layers. This seemingly small change produces a drastic improvement in the performance of the model.

We will use the 101-layer-deep ResNet architecture (ResNet101) pre-trained on the ImageNet dataset. Similar to VGGNet, we only need to replace the final fully-connected layer with our classifier to produce 133-dimensional output. The following code block implements the solution.

```
In [21]: # import torch.optim as optim

         model_transfer2 = models.resnet101(pretrained=True)

         for param in model_transfer2.parameters():
             param.requires_grad = False

         model_transfer2.fc = nn.Linear(2048, 133)
```

```
        if use_cuda:
            model_transfer2 = model_transfer2.cuda()

        criterion_transfer2 = nn.CrossEntropyLoss()
        optimizer_transfer2 = optim.Adam(model_transfer2.fc.parameters(), lr=0.001)

        # train the model
        model_transfer2 = train(20, loaders_transfer, model_transfer2, optimizer_transfer2, cri

        # load the model that got the best validation accuracy (uncomment the line below)
        model_transfer2.load_state_dict(torch.load('model_transfer2.pt'))
```

```
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /root/.torch/models
100%|| 178728960/178728960 [00:14<00:00, 12669644.24it/s]


Epoch: 1         Training Loss: 1.922533         Validation Loss: 0.728569
Validation loss decreased (inf --> 0.728569).  Saving model ...
Epoch: 2         Training Loss: 0.610473         Validation Loss: 0.585945
Validation loss decreased (0.728569 --> 0.585945).  Saving model ...
Epoch: 3         Training Loss: 0.439258         Validation Loss: 0.513522
Validation loss decreased (0.585945 --> 0.513522).  Saving model ...
Epoch: 4         Training Loss: 0.350083         Validation Loss: 0.462946
Validation loss decreased (0.513522 --> 0.462946).  Saving model ...
Epoch: 5         Training Loss: 0.291649         Validation Loss: 0.450441
Validation loss decreased (0.462946 --> 0.450441).  Saving model ...
Epoch: 6         Training Loss: 0.270571         Validation Loss: 0.429810
Validation loss decreased (0.450441 --> 0.429810).  Saving model ...
Epoch: 7         Training Loss: 0.232485         Validation Loss: 0.421586
Validation loss decreased (0.429810 --> 0.421586).  Saving model ...
Epoch: 8         Training Loss: 0.217263         Validation Loss: 0.469459
Epoch: 9         Training Loss: 0.193880         Validation Loss: 0.428393
Epoch: 10         Training Loss: 0.190744         Validation Loss: 0.437733
Epoch: 11         Training Loss: 0.187236         Validation Loss: 0.397946
Validation loss decreased (0.421586 --> 0.397946).  Saving model ...
Epoch: 12         Training Loss: 0.158173         Validation Loss: 0.403430
Epoch: 13         Training Loss: 0.155980         Validation Loss: 0.436753
Epoch: 14         Training Loss: 0.149405         Validation Loss: 0.429206
Epoch: 15         Training Loss: 0.146173         Validation Loss: 0.479776
Epoch: 16         Training Loss: 0.139664         Validation Loss: 0.483285
Epoch: 17         Training Loss: 0.125960         Validation Loss: 0.443054
Epoch: 18         Training Loss: 0.129630         Validation Loss: 0.477187
Epoch: 19         Training Loss: 0.137159         Validation Loss: 0.521238
Epoch: 20         Training Loss: 0.123384         Validation Loss: 0.568297
```

```
In [38]: model_transfer2.load_state_dict(torch.load('model_transfer2.pt'))
         predicted_labels, true_labels = test(loaders_transfer, model_transfer2, criterion_trans
```

```
Test Loss: 0.521041


Test Accuracy: 85% (718/836)
```

```
In [39]: predictions = []
         for labels in predicted_labels:
             for label in np.array(labels):
                 predictions.append(label[0])

         ground_truths = []
         for labels in true_labels:
             for label in np.array(labels):
                 ground_truths.append(label)

         precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
         recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
         print('Average test precision: {:.1f}%'.format(precision*100))
         print('Average test recall: {:.1f}%'.format(recall*100))
```

```
Average test precision: 87.1%
Average test recall: 84.3%


/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
```

1.2.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

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

         # 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_loader.dataset.classes]

         def load_transform_image(img_path):

             # Pre-process
             transform = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
                 ])
```

```
    # Open the image and apply the transformation process
    img = Image.open(img_path).convert('RGB')
    img = transform(img)[:3,:,:].unsqueeze(0)

    return img

def predict_breed_transfer(img_path, class_names, model):
    # load the image and return the predicted breed
    img = load_transform_image(img_path)

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

    model.eval()
    prediction = model(img)

    index = torch.max(prediction, 1)[1].item()

    return class_names[index]
```

```
In [41]: predict_breed_transfer('./images/sample_cnn.png', class_names, model_transfer2)
```

```
Out[41]: 'Cavalier king charles spaniel'
```

```
In [42]: # Test
         predict_breed_transfer('./images/Welsh_springer_spaniel_08203.jpg', class_names, model_
```

```
Out[42]: 'Welsh springer spaniel'
```

Step 5: Write your Algorithm

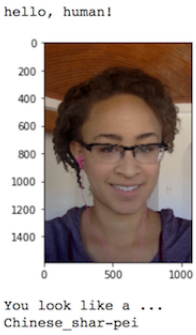
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is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

1.2.18 (IMPLEMENTATION) Write your Algorithm

```
In [43]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
```



Sample Human Output

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)

    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(img_path, class_names, model_transfer2)
        print("\n\nHello, Dog!")
        plt.show()
        print("You look like a ...\n{0}".format(prediction))

    elif face_detector(img_path) is True:
        prediction = predict_breed_transfer(img_path, class_names, model_transfer2)
        print("\n\nHello, Human!")
        plt.show()
        print("You look like a ...\n{0}".format(prediction))
    else:
        print("\n\nError! No Dog or Human detected!")
        plt.show()
```

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?

1.2.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

Overall, the dog app works rather well. It can detect human, dog, or neither in the given images. Then it provides estimates on the breed for the dog image, the resembling dog breed for

the human image, or returns an error message.

There is a lot of room to improve our model performance and reduce training time:

Try to increase the training data by other data augmentation techniques like random shift, random size, generated images with various contrasts, etc.

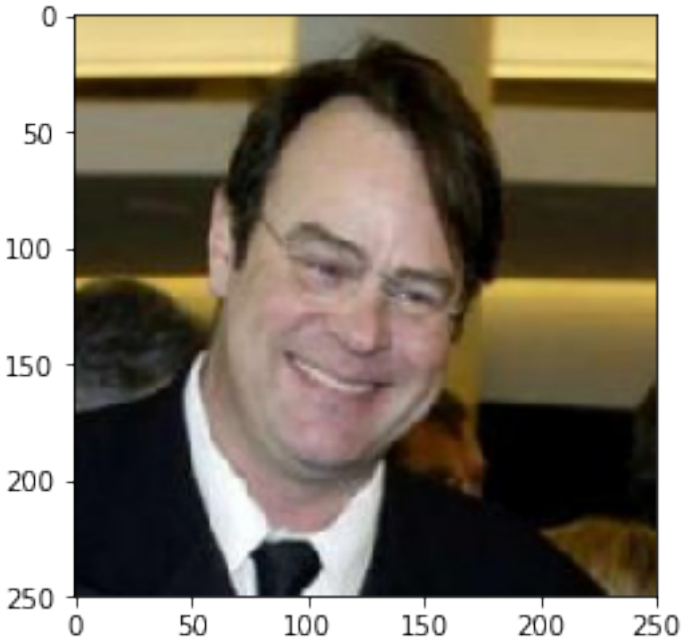
Try other variants of ResNet or different CNN architectures (GoogLeNet, Xception, SENet – winner of ImageNet 2017, etc.) or a combination of CNN models.

Playing with hyperparameters like batch size, learning rate, optimizer, number of epochs, activation function, etc.

Instead of using a fixed learning rate, try to use a learning rate scheduler, which will change the learning rate after every batch of training.

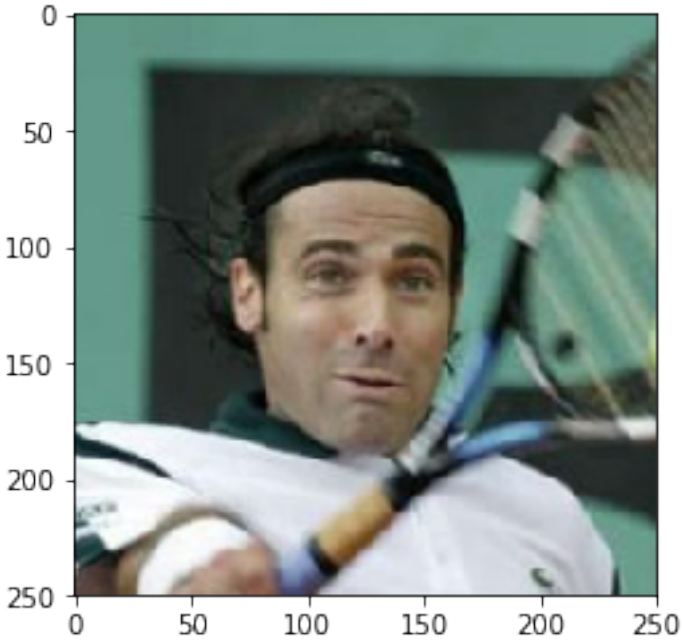
In [44]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

```
Hello, Human!
```

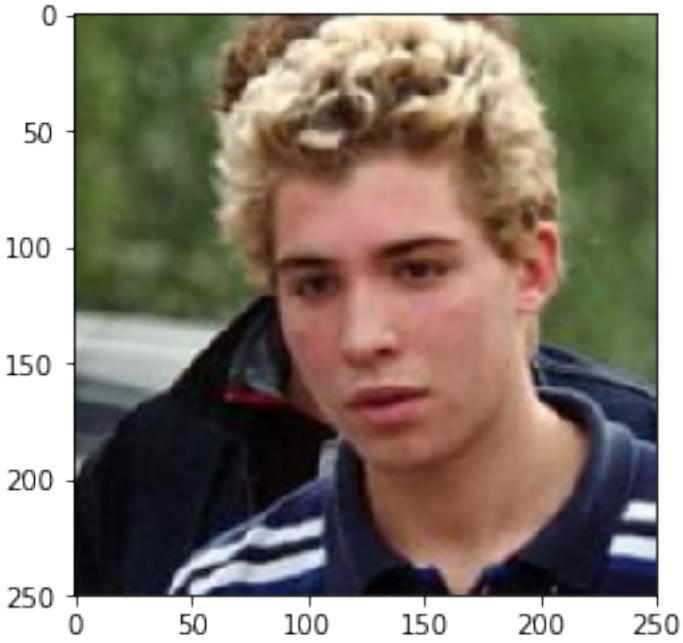



```
You look like a ...
Welsh springer spaniel
```

```
Hello, Human!
```

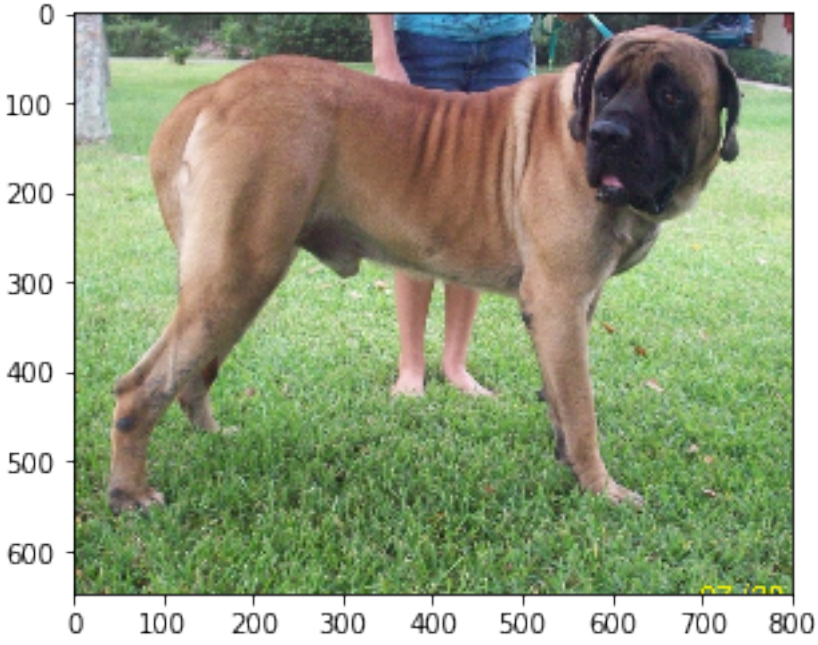



```
You look like a ...
Lowchen


Hello, Human!
```

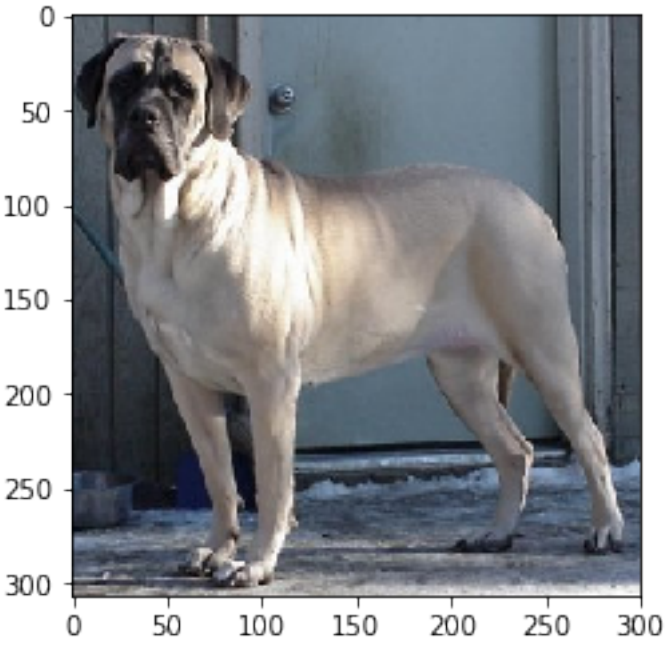



```
You look like a ...
Welsh springer spaniel
```

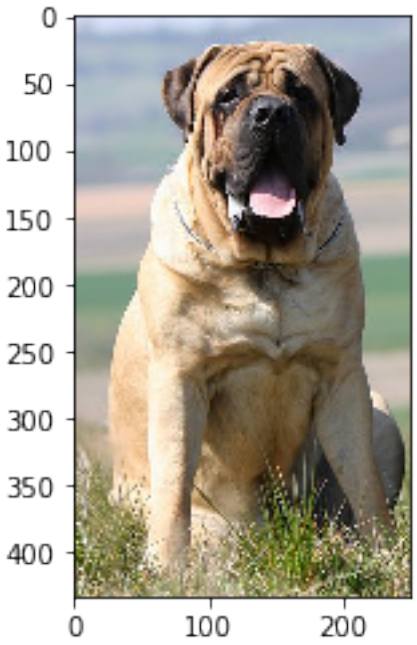
```
Hello, Dog!
```



```
You look like a ...
Mastiff


Hello, Dog!
```




```
You look like a ...
Mastiff


Hello, Dog!
```




```
You look like a ...
Bullmastiff
```

In [ ]: