dog_app

November 25, 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

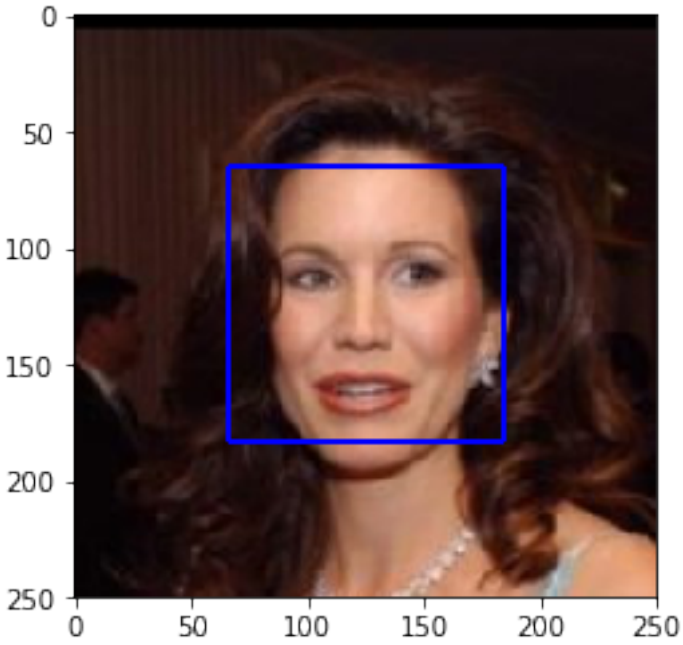
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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[103])
        # 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
```




```
In [3]: def img_show_haar(image_path):
            # load color (BGR) image
            img = cv2.imread(image_path)
            # 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()
```

Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [4]: # returns "True" if face is detected in image stored at img_path
        def face_detector_haar(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:

The results of using haarcascades face detector:

1. Detected human faces in `human_files_short` : 98% (98/100)

There are two pictures with human face were missed by haarcascades face detector. Please have a look at `human_files_short[56]` and `human_files_short[87]`, as we can see in `human_files_short[87]`, the person wore a sun glasses, this might be the reason why the detector could not identify his face. However, for `human_files_short[56]` I have no clear idea why she did not be detected, maybe the camera shot angles was too high.

2. Detected human faces in `dog_files_short` : 17% (17/100)

There are really some pictures in `dog_files_short` show human faces because people took photo with their pet. For example, we can see in `dog_files_short[27]`, there is a woman took photo with her dog.

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

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

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

        ## Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        human_test = []
        dog_test = []
        for i in range(100):
            human_test.append(face_detector_haar(human_files_short[i]))
            dog_test.append(face_detector_haar(dog_files_short[i]))
```

```
In [6]: # Show the test results
        print('Detect faces in human files (haar): {} / {}'.format(sum(human_test), len(human_te
        print('Detect faces in dog files (haar): {} / {}'.format(sum(dog_test), len(dog_test)))
```

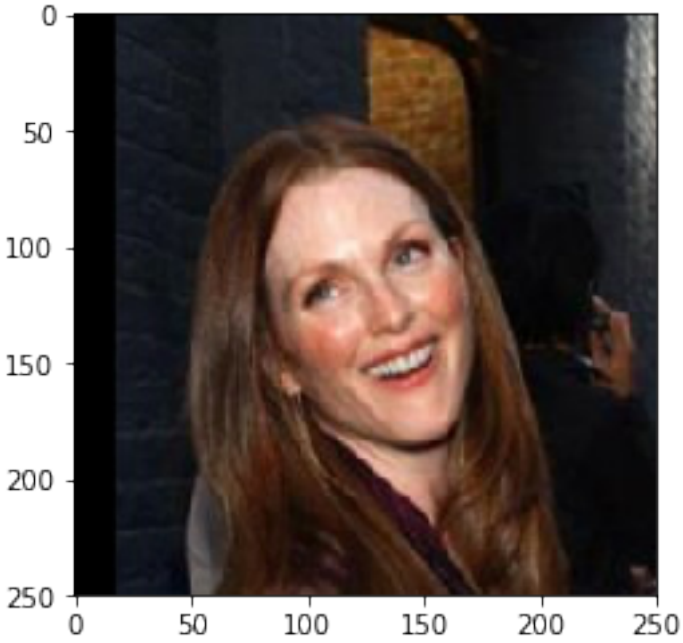
```
Detect faces in human files (haar): 98 / 100
Detect faces in dog files (haar): 17 / 100
```

```
In [7]: print('Human detect false cases:', [i for i, x in enumerate(human_test) if x == False])
        print('dog picture with people face: ',[i for i, x in enumerate(dog_test) if x])
```

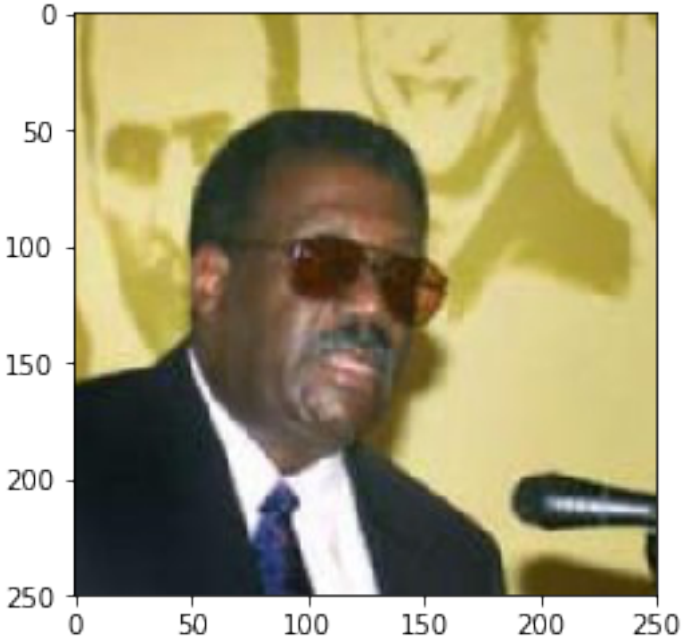
```
Human detect false cases: [56, 87]
dog picture with people face:  [26, 27, 39, 40, 42, 49, 56, 58, 59, 71, 77, 79, 84, 88, 89, 90,
```

```
In [8]: img_show_haar(human_files_short[56])
        img_show_haar(human_files_short[87])
        img_show_haar(dog_files_short[27])
```

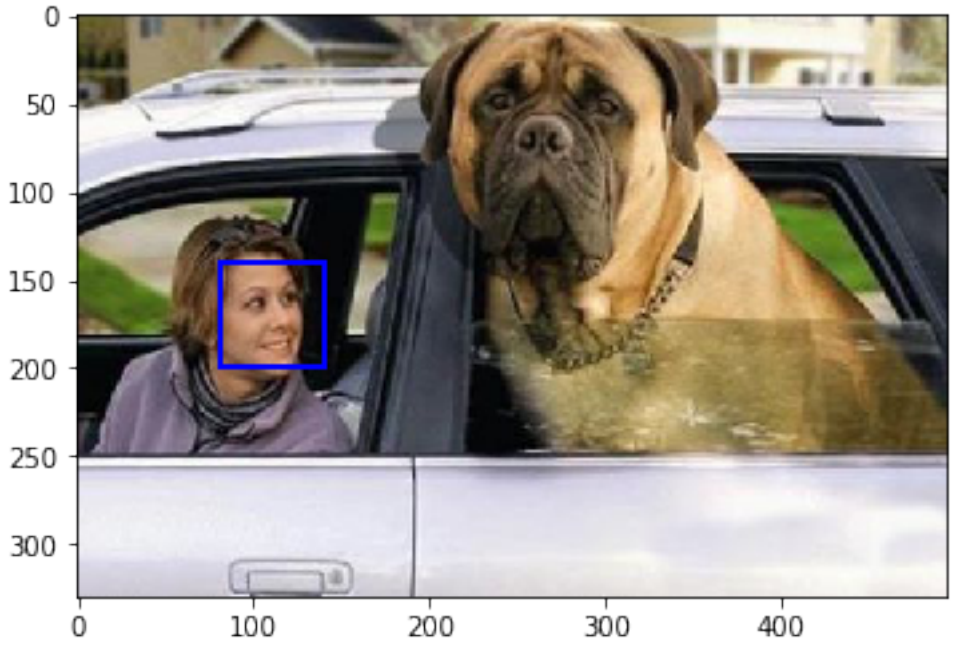
```
Number of faces detected: 0
```




```
Number of faces detected: 0
```



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



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 [9]: ### Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
        # cv2.CascadeClassifier('lbpcascade/lbpcascade_frontalface_improved.xml')

In [10]: ! pip install dlib

Collecting dlib
  Using cached https://files.pythonhosted.org/packages/a4/7b/2f7f29f460629a8143b2deea1911e2fb1d9
Building wheels for collected packages: dlib
  Running setup.py bdist_wheel for dlib ... done
  Stored in directory: /root/.cache/pip/wheels/e3/fd/51/22af51f198c3d1adde947c1189c6dfc70923d70c
Successfully built dlib
Installing collected packages: dlib
Successfully installed dlib-19.21.0


In [11]: import dlib
```

```
        def face_detector_dlib(img_path):
            detector = dlib.get_frontal_face_detector()
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = detector(gray, 1)
            return len(faces) > 0
```

```
In [12]: def img_show_dlib(img_path):
            detector = dlib.get_frontal_face_detector()
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = detector(gray, 1) # result

            #to draw faces on image
            for result in faces:
                x = result.left()
                y = result.top()
                x1 = result.right()
                y1 = result.bottom()
                cv2.rectangle(img, (x, y), (x1, y1), (0, 0, 255), 2)

            cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            # display the image, along with bounding bo
            plt.imshow(cv_rgb)
            plt.show()
```

```
In [13]: human_test_dlib = []
         dog_test_dlib = []
         for i in range(100):
             human_test_dlib.append(face_detector_dlib(human_files_short[i]))
             dog_test_dlib.append(face_detector_dlib(dog_files_short[i]))
```

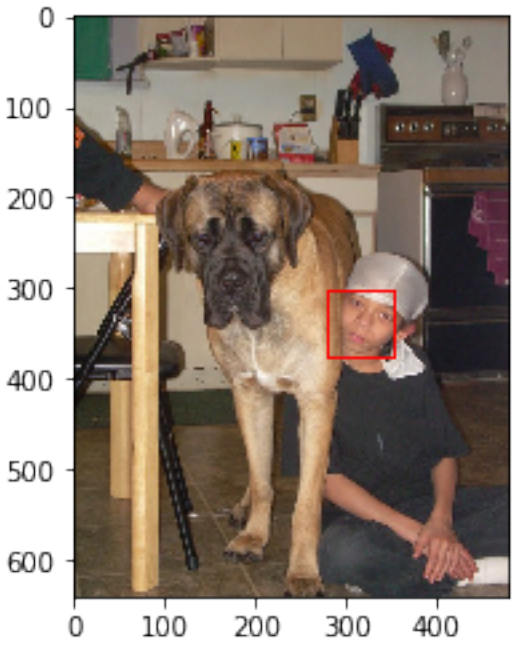
```
In [14]: # Show the test results
         print('Detect faces in human files (dlib): {} / {}'.format(sum(human_test_dlib), len(hu
         print('Detect faces in dog files (dlib): {} / {}'.format(sum(dog_test_dlib), len(dog_te
```

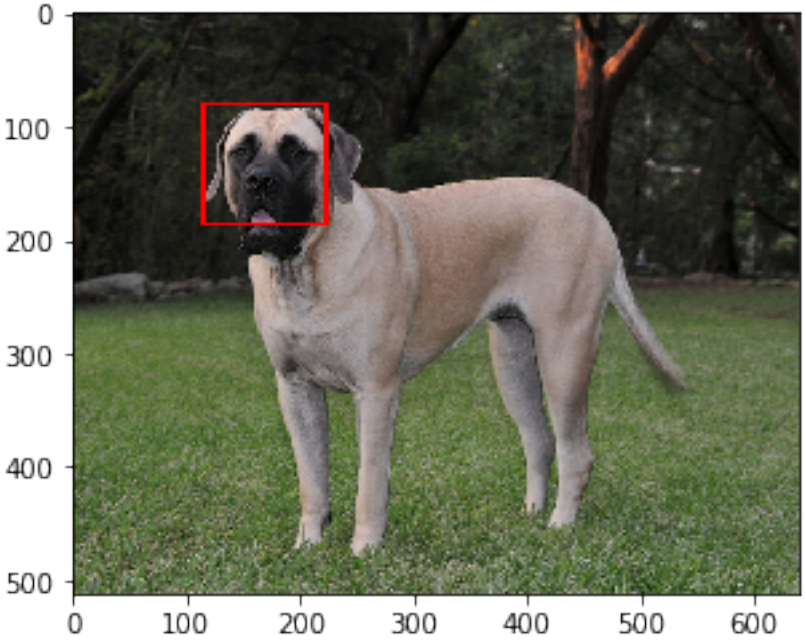
```
Detect faces in human files (dlib): 100 / 100
Detect faces in dog files (dlib): 7 / 100
```

```
In [15]: print('Human detect false cases:', [i for i, x in enumerate(human_test_dlib) if x == Fa
         print('dog picture with people face: ',[i for i, x in enumerate(dog_test_dlib) if x])
```

```
Human detect false cases: []
dog picture with people face:  [7, 16, 27, 39, 42, 56, 65]
```

```
In [16]: img_show_dlib(dog_files_short[7])
         img_show_dlib(dog_files_short[16])
```





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 [17]: 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()

             print(VGG16)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:10<00:00, 54392837.76it/s]


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.

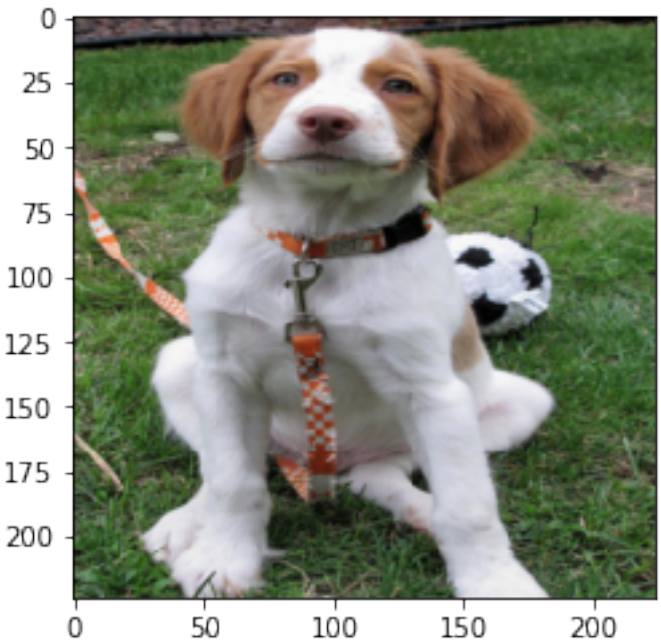
Before writing the function, make sure that you take the time to learn how to appropriately pre-process tensors for pre-trained models in the PyTorch documentation.

```
In [18]: from PIL import Image
         import torchvision.transforms as transforms

         def load_image(img_path):
             ''' Load in and transform an image'''

             image = Image.open(img_path).convert('RGB')
             in_transform = transforms.Compose([
                                 transforms.Resize(size = (224,224)),
                                 #transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize((0.485, 0.456, 0.406),
                                                      (0.229, 0.224, 0.225))])
```

```
        # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
        image = in_transform(image)[:3,:,:].unsqueeze(0)
        return image



    # helper function for un-normalizing an image
    # and converting it from a Tensor image to a NumPy image for display
    def im_convert(tensor):
        """ Display a tensor as an image. """

        image = tensor.to("cpu").clone().detach()
        image = image.numpy().squeeze()
        image = image.transpose(1,2,0)
        image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
        image = image.clip(0, 1)

        return image
```

In [19]:
```
tmp = load_image('images/Brittany_02625.jpg')
plt.imshow(im_convert(tmp))
```

Out[19]: <matplotlib.image.AxesImage at 0x7f7f9db9de80>



In [20]:
```
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
    '''

    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    image_tensor = load_image(img_path)

    # move model inputs to cuda, if GPU available
    if use_cuda:
        image_tensor = image_tensor.cuda()

    # get sample outputs
    VGG16.eval()
    output = VGG16(image_tensor)
    index = torch.max(output, 1)[1].item()
    return index # predicted class index
```

```
In [21]: # test run
         print(VGG16_predict('images/Brittany_02625.jpg'))
         print(VGG16_predict(dog_files_short[10]))
```

```
215
243
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

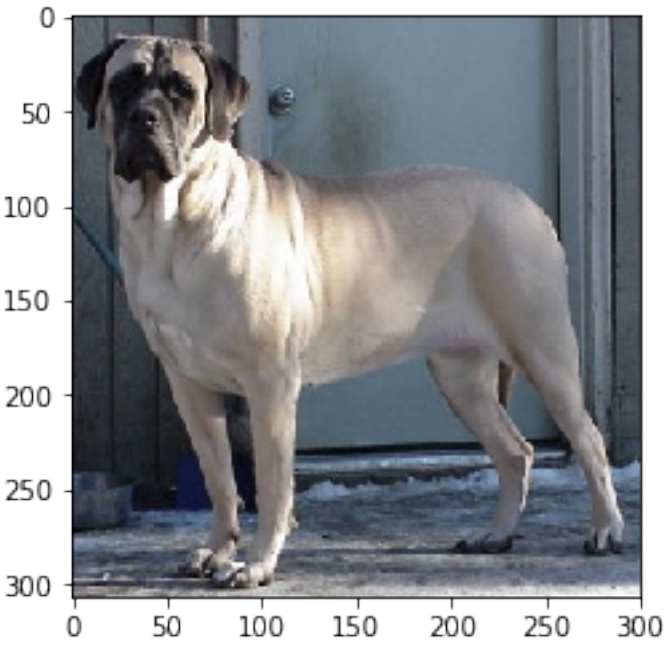
While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted 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 [22]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             index = VGG16_predict(img_path)
             TF = index >= 151 and index <=268
             return TF # True/False
```

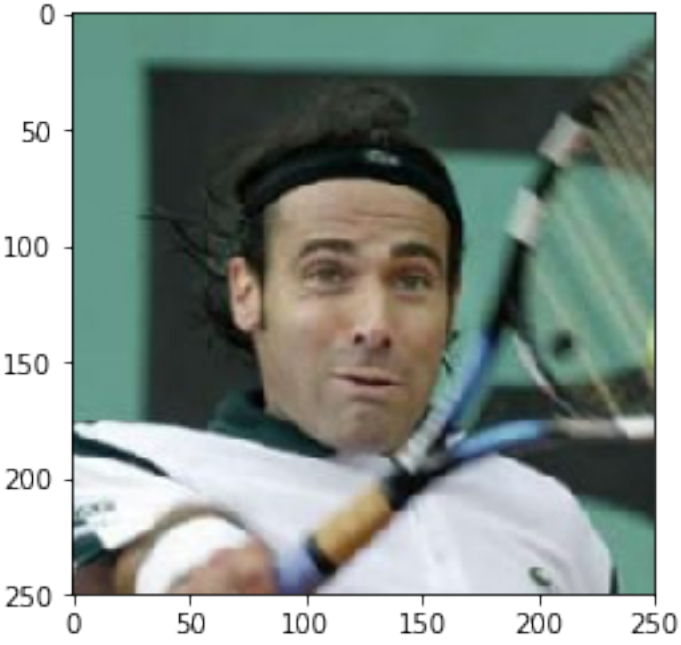
```
In [23]: plt.imshow(Image.open(dog_files_short[1]))
         dog_detector(dog_files_short[1])
```

Out[23]: True



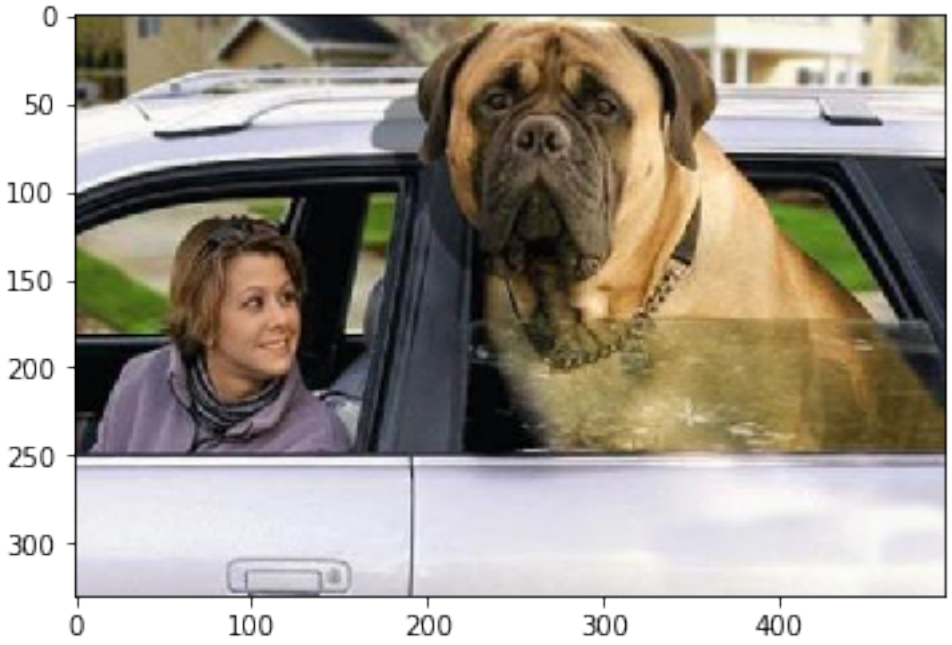
In [24]:
```
plt.imshow(Image.open(human_files_short[1]))
dog_detector(human_files_short[1])
```

Out[24]: False



In [25]:
```
plt.imshow(Image.open(dog_files_short[27]))
dog_detector(dog_files_short[27])
```

Out[25]: True



In [26]:
```
plt.imshow(Image.open(dog_files_short[7]))
dog_detector(dog_files_short[7])
```

Out[26]: True

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:
Below shows the results when using VGG16 pre-trained model:
Detected dogs in `human_files_short`: 0% (0/100)
Detected dogs in `dog_files_short`: 100% (100/100)

```
In [27]: ### Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_det_dog = []
         dog_det_dog = []

         for i in range(100):
             human_det_dog.append(dog_detector(human_files_short[i]))
             dog_det_dog.append(dog_detector(dog_files_short[i]))
```

```
In [28]: # Show the test results
         print('Detect dog in human files (vgg16): {} / {}'.format(sum(human_det_dog), len(human
         print('Detect dog in dog files (vgg16): {} / {}'.format(sum(dog_det_dog), len(dog_det_d
```

```
Detect dog in human files (vgg16): 0 / 100
Detect dog in dog files (vgg16): 100 / 100
```

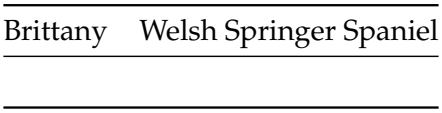
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

```
In [29]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

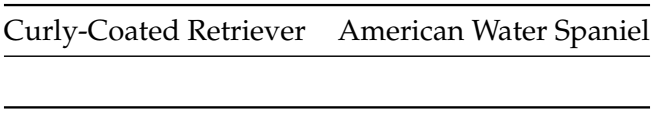
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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



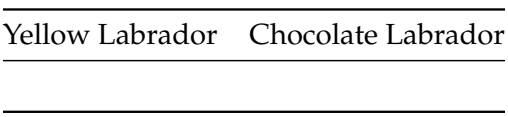
Brittany Welsh Springer Spaniel

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



Curly-Coated Retriever American Water Spaniel

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



Yellow Labrador Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [30]:
```
import os
from torchvision import datasets
import torch
import torchvision.models as models
import torchvision.transforms as transforms

import numpy as np
from glob import glob
import time
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim

from PIL import ImageFile
# To avoid error when images are broken(truncated)
ImageFile.LOAD_TRUNCATED_IMAGES = True

# Use GPU
use_cuda = torch.cuda.is_available()

### Data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images/'
train_dir = data_dir + '/train'
val_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

# VGG-16 Takes 224x224 images as input, so we resize all of them
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.Resize(size = (224,224)),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])


val_transforms = transforms.Compose([transforms.Resize(size = (224,224)),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
                                                          [0.229, 0.224, 0.225])])
```

```
test_transforms = transforms.Compose([transforms.Resize(size = (224,224)),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

# Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform = train_transforms)
val_data = datasets.ImageFolder(val_dir, transform = val_transforms)
test_data = datasets.ImageFolder(test_dir, transform = test_transforms)

# Using the image datasets and the trainforms, define the dataloaders
batch = 20
train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch, shuffle = Tr
val_loader = torch.utils.data.DataLoader(val_data, batch_size = batch)
test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch)
```

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:

- **For training set:**

I apply a 30 degree random rotation, and then resize all photos to size 224x224 since the default input size of VGG16 is 224x224 RGB image, after that I make horizontal flipping randomly on all of them. I do image augmentation because it is a good way to avoid the model memorise the data.

- **For validation and test sets:**

We won't do any image augmentation in validation and test sets because we never train our model in these sets so there are no overfitting problem in it. What we want is to check how good our model is. Therefore, I only make sure the input image size is 224z224, and each colour channel is normalized separately in mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [31]:
```
num_classes = 133
out_lay = 64


# define the CNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
```

```
            # conv layer (input 3 means RGB. depth from 1 -> 64), 3x3 kernels
            self.conv1 = nn.Conv2d(3, 16, 3, padding = 1)
            self.conv2 = nn.Conv2d(16, 32, 3, padding = 1)
            self.conv3 = nn.Conv2d(32, out_lay, 3, padding = 1)

            # linear layer
            self.fc1 = nn.Linear(28*28*out_lay, 500)
            self.fc2 = nn.Linear(500, num_classes)

            # Maxpoll layer
            self.pool = nn.MaxPool2d(2, 2)

            # dropout layer
            self.dropout = nn.Dropout(0.25)


        def forward(self, x):
            x = F.relu(self.conv1(x))
            x = self.pool(x)
            x = F.relu(self.conv2(x))
            x = self.pool(x)
            x = F.relu(self.conv3(x))
            x = self.pool(x)


            x = x.view(x.size(0), -1) # = x.view(-1, 28*28*out_lay)
            x = self.dropout(x)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = self.fc2(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()
    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=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 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))
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
```

```
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (dropout): Dropout(p=0.25)
)
```

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

Answer:

As our goal is to reach a 10% accuracy which is fairly high comparing to random guess 1/133 = 0.75%. Therefore, we need some convolution layers to extract more detail features. However, it is relatively time consuming if we use many convolutional layers. After I saw the first two convolutional layers in VGG16 whose depth were 64, hence, I decide to have two to three convolutional layers which have at most 64 filters. And then I'd flatten the output from convolutional layer, let them go to two fully connected layers and give us a 133 nodes output layer in terms of 133 dog breeds. In between, I put several Max Pooling and Dropout layers to reduce the chance of overfitting.

Below shows the architecture of my CNN:

```
1.  conv1 = a convolutation layer with depth 16 and kernel size 3x3
2.  MaxPooling
3.  conv2 = a convolutation layer with depth 32 and kernel size 3x3
4.  MaxPooling
5.  conv3 = a convolutation layer with depth 64 and kernel size 3x3
6.  MaxPooling
7.  tensor flatten
8.  Dropout
9.  fc1 = a fully connected layer convert nodes from 50176 (28*28*64) to 500
10. Dropout
11. fc2 = a fully connected layer convert nodes from 500 to 133
```

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 [32]: ### select loss function
         #criterion_scratch = nn.NLLLoss()
         criterion_scratch = nn.CrossEntropyLoss()

         ### select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.075)
```

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 [33]: def train(n_epochs, train_loader, 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
    start = time.time()

    steps = 0
    print_every = len(train_loader) / 2

    ###################
    # train the model #
    ###################
    model.train()
    for data, target in train_loader:
        steps += 1
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()

        optimizer.zero_grad()
        output = model.forward(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

        if steps % print_every == 0:
            print(f"Epoch {epoch}/{n_epochs}.. "
              f"Train loss: {train_loss/len(train_loader):.3f} | "
              f"During: {time.time() - start:.3f} sec | "
              f"Steps {steps}.. ")


    ######################
    # validate the model #
    ######################
    model.eval()
    with torch.no_grad(): # to avoid "cuda runtime error (2) : out of memory"
        for data, target in val_loader:
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()

            ## update the average validation loss
            val_output = model.forward(data)
```

```
                    batch_loss = criterion(val_output, target)
                    valid_loss += batch_loss.item()


            # print training/validation statistics
            print('Epoch:{} | Training Loss:{:.3f} | Validation Loss:{:.3f} | During: {:.3f
                epoch,
                train_loss/len(train_loader),
                valid_loss/len(val_loader),
                time.time() - start
                ))

            ## 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("Saving Model... ")

        # return trained model
        return model
```

```
In [34]: # train the model
         print('Using GPU: ', use_cuda)
         model_scratch = train(15, train_loader, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Using GPU:  True
Epoch 1/15.. Train loss: 2.441 | During: 62.131 sec | Steps 167..
Epoch 1/15.. Train loss: 4.830 | During: 126.435 sec | Steps 334..
Epoch:1 | Training Loss:4.830 | Validation Loss:4.662 | During: 136.643 sec
Saving Model...
Epoch 2/15.. Train loss: 2.327 | During: 55.850 sec | Steps 167..
Epoch 2/15.. Train loss: 4.590 | During: 111.486 sec | Steps 334..
Epoch:2 | Training Loss:4.590 | Validation Loss:4.448 | During: 121.210 sec
Saving Model...
Epoch 3/15.. Train loss: 2.208 | During: 56.853 sec | Steps 167..
Epoch 3/15.. Train loss: 4.394 | During: 111.142 sec | Steps 334..
Epoch:3 | Training Loss:4.394 | Validation Loss:4.313 | During: 120.805 sec
Saving Model...
Epoch 4/15.. Train loss: 2.141 | During: 54.470 sec | Steps 167..
Epoch 4/15.. Train loss: 4.269 | During: 110.881 sec | Steps 334..
Epoch:4 | Training Loss:4.269 | Validation Loss:4.189 | During: 120.573 sec
Saving Model...
Epoch 5/15.. Train loss: 2.080 | During: 56.941 sec | Steps 167..
Epoch 5/15.. Train loss: 4.156 | During: 111.526 sec | Steps 334..
Epoch:5 | Training Loss:4.156 | Validation Loss:4.120 | During: 121.287 sec
Saving Model...
Epoch 6/15.. Train loss: 2.032 | During: 56.096 sec | Steps 167..
```

```
Epoch 6/15.. Train loss: 4.051 | During: 111.330 sec | Steps 334..
Epoch:6 | Training Loss:4.051 | Validation Loss:4.051 | During: 121.026 sec
Saving Model...
Epoch 7/15.. Train loss: 1.948 | During: 57.206 sec | Steps 167..
Epoch 7/15.. Train loss: 3.942 | During: 111.383 sec | Steps 334..
Epoch:7 | Training Loss:3.942 | Validation Loss:4.032 | During: 121.116 sec
Saving Model...
Epoch 8/15.. Train loss: 1.934 | During: 56.546 sec | Steps 167..
Epoch 8/15.. Train loss: 3.873 | During: 111.209 sec | Steps 334..
Epoch:8 | Training Loss:3.873 | Validation Loss:3.950 | During: 120.950 sec
Saving Model...
Epoch 9/15.. Train loss: 1.851 | During: 55.630 sec | Steps 167..
Epoch 9/15.. Train loss: 3.742 | During: 111.313 sec | Steps 334..
Epoch:9 | Training Loss:3.742 | Validation Loss:4.262 | During: 121.029 sec
Epoch 10/15.. Train loss: 1.811 | During: 56.068 sec | Steps 167..
Epoch 10/15.. Train loss: 3.656 | During: 111.159 sec | Steps 334..
Epoch:10 | Training Loss:3.656 | Validation Loss:3.828 | During: 120.874 sec
Saving Model...
Epoch 11/15.. Train loss: 1.765 | During: 55.665 sec | Steps 167..
Epoch 11/15.. Train loss: 3.560 | During: 110.800 sec | Steps 334..
Epoch:11 | Training Loss:3.560 | Validation Loss:3.833 | During: 120.451 sec
Epoch 12/15.. Train loss: 1.717 | During: 54.870 sec | Steps 167..
Epoch 12/15.. Train loss: 3.457 | During: 110.882 sec | Steps 334..
Epoch:12 | Training Loss:3.457 | Validation Loss:3.862 | During: 120.529 sec
Epoch 13/15.. Train loss: 1.658 | During: 53.456 sec | Steps 167..
Epoch 13/15.. Train loss: 3.363 | During: 110.972 sec | Steps 334..
Epoch:13 | Training Loss:3.363 | Validation Loss:4.014 | During: 120.781 sec
Epoch 14/15.. Train loss: 1.578 | During: 56.538 sec | Steps 167..
Epoch 14/15.. Train loss: 3.238 | During: 111.151 sec | Steps 334..
Epoch:14 | Training Loss:3.238 | Validation Loss:3.847 | During: 120.906 sec
Epoch 15/15.. Train loss: 1.547 | During: 53.669 sec | Steps 167..
Epoch 15/15.. Train loss: 3.154 | During: 110.927 sec | Steps 334..
Epoch:15 | Training Loss:3.154 | Validation Loss:3.852 | During: 120.648 sec
```

```
In [35]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [36]: def test(loaders, model, criterion, use_cuda):

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

```
    total = 0.
    if use_cuda:
        model.cuda()

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

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

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

```
In [37]: # call test function
         test(test_loader, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.809948


Test Accuracy: 11% (92/836)
```

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.

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

         ## Specify model architecture
         # initial variables
         arch = 'vgg16'
         hidden_layer = 1024
         device = 'cuda'
         n_epochs = 10
         num_classes = 133

         def Classifier(arch = 'vgg16', hidden_layer = 1024):
             model = models.vgg16(pretrained=True)
             input_layer = 25088

             # freeze parameters
             for param in model.parameters():
                 param.requires_grad = False

             My_Classifier = nn.Sequential(
                             nn.Linear(input_layer, hidden_layer),
                             nn.Linear(hidden_layer, num_classes),
                             nn.LogSoftmax(dim = 1))

             model.classifier = My_Classifier
             return model

In [39]: model_transfer = Classifier()

         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:

It is judicious to use a pre-trained model rather than training a new model from scratch as we can see in `Step 3` it took around a hour GPU time to reach a 10% accuracy, if we want more accurate result, it may take days or weeks to train to model. Since in `Step 2`, we have already seen the potential strength of VGG16 shows it is a well-trained model. Therefore, I'd simply transfer its convolution layers and only modify its classifier.

The easiest way to modify the classifier is set the number of output nodes as the number of dog breeds:

```
model_transfer.classifier[6] = nn.Linear(input_size, 133)
```

However, I don't want the nodes decrease so rapidly since the input size of the first fully connected layer is 25088, it might loss much information from 25088 nodes go down sharply to 133 nodes. Hence, I set a hidden layer to bridge the gap. The length of my hidden layer is 1024, below shows my architecture:

```
1. VGG16 pre-trained convolution layers
2. Linear(25088, 1024)
3. Linear(1024, 133)
```

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [41]: # train the model
         model_transfer = train(n_epochs, train_loader, model_transfer, optimizer_transfer,
                                criterion_transfer, use_cuda, 'model_transfer.pt')
```

```
Epoch 1/10.. Train loss: 1.712 | During: 113.766 sec | Steps 167..
Epoch 1/10.. Train loss: 2.678 | During: 230.190 sec | Steps 334..
Epoch:1 | Training Loss:2.678 | Validation Loss:1.348 | During: 256.624 sec
Saving Model...
Epoch 2/10.. Train loss: 0.604 | During: 117.242 sec | Steps 167..
Epoch 2/10.. Train loss: 1.128 | During: 230.457 sec | Steps 334..
Epoch:2 | Training Loss:1.128 | Validation Loss:0.907 | During: 256.753 sec
Saving Model...
Epoch 3/10.. Train loss: 0.407 | During: 115.098 sec | Steps 167..
Epoch 3/10.. Train loss: 0.770 | During: 230.461 sec | Steps 334..
Epoch:3 | Training Loss:0.770 | Validation Loss:0.732 | During: 256.691 sec
Saving Model...
Epoch 4/10.. Train loss: 0.307 | During: 115.409 sec | Steps 167..
Epoch 4/10.. Train loss: 0.599 | During: 230.044 sec | Steps 334..
Epoch:4 | Training Loss:0.599 | Validation Loss:0.665 | During: 256.313 sec
Saving Model...
Epoch 5/10.. Train loss: 0.243 | During: 114.460 sec | Steps 167..
Epoch 5/10.. Train loss: 0.483 | During: 230.427 sec | Steps 334..
Epoch:5 | Training Loss:0.483 | Validation Loss:0.598 | During: 256.768 sec
Saving Model...
Epoch 6/10.. Train loss: 0.204 | During: 115.795 sec | Steps 167..
Epoch 6/10.. Train loss: 0.413 | During: 230.475 sec | Steps 334..
Epoch:6 | Training Loss:0.413 | Validation Loss:0.573 | During: 256.800 sec
```

```
Saving Model...
Epoch 7/10.. Train loss: 0.180 | During: 116.937 sec | Steps 167..
Epoch 7/10.. Train loss: 0.364 | During: 230.418 sec | Steps 334..
Epoch:7 | Training Loss:0.364 | Validation Loss:0.541 | During: 256.879 sec
Saving Model...
Epoch 8/10.. Train loss: 0.150 | During: 115.656 sec | Steps 167..
Epoch 8/10.. Train loss: 0.306 | During: 230.511 sec | Steps 334..
Epoch:8 | Training Loss:0.306 | Validation Loss:0.523 | During: 256.830 sec
Saving Model...
Epoch 9/10.. Train loss: 0.144 | During: 114.719 sec | Steps 167..
Epoch 9/10.. Train loss: 0.285 | During: 230.004 sec | Steps 334..
Epoch:9 | Training Loss:0.285 | Validation Loss:0.507 | During: 256.213 sec
Saving Model...
Epoch 10/10.. Train loss: 0.127 | During: 114.645 sec | Steps 167..
Epoch 10/10.. Train loss: 0.252 | During: 230.094 sec | Steps 334..
Epoch:10 | Training Loss:0.252 | Validation Loss:0.510 | During: 256.329 sec
```

```
In [42]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

In [43]: test(test_loader, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.550096


Test Accuracy: 82% (689/836)
```

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%.

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 [44]: ### 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]
         print('First 3 dog types: ', class_names[:3])


         def predict_breed_transfer(img_path, model, topk = 5):
             # load the image and return the predicted breed
```

```
            img = load_image(img_path)

            device = torch.device('cuda' if use_cuda else 'cpu')
            model.to(device)
            model.eval()
            img = img.to(device)

            # calculate the class probabilities for img
            with torch.no_grad():
                output = model.forward(img)

            ps = torch.exp(output)
            top_p, top_class = ps.topk(topk)
            probs = top_p.cpu().numpy()[0]
            idx = top_class.cpu().numpy()[0]
            classes = [class_names[x] for x in idx]

            return probs, classes
```

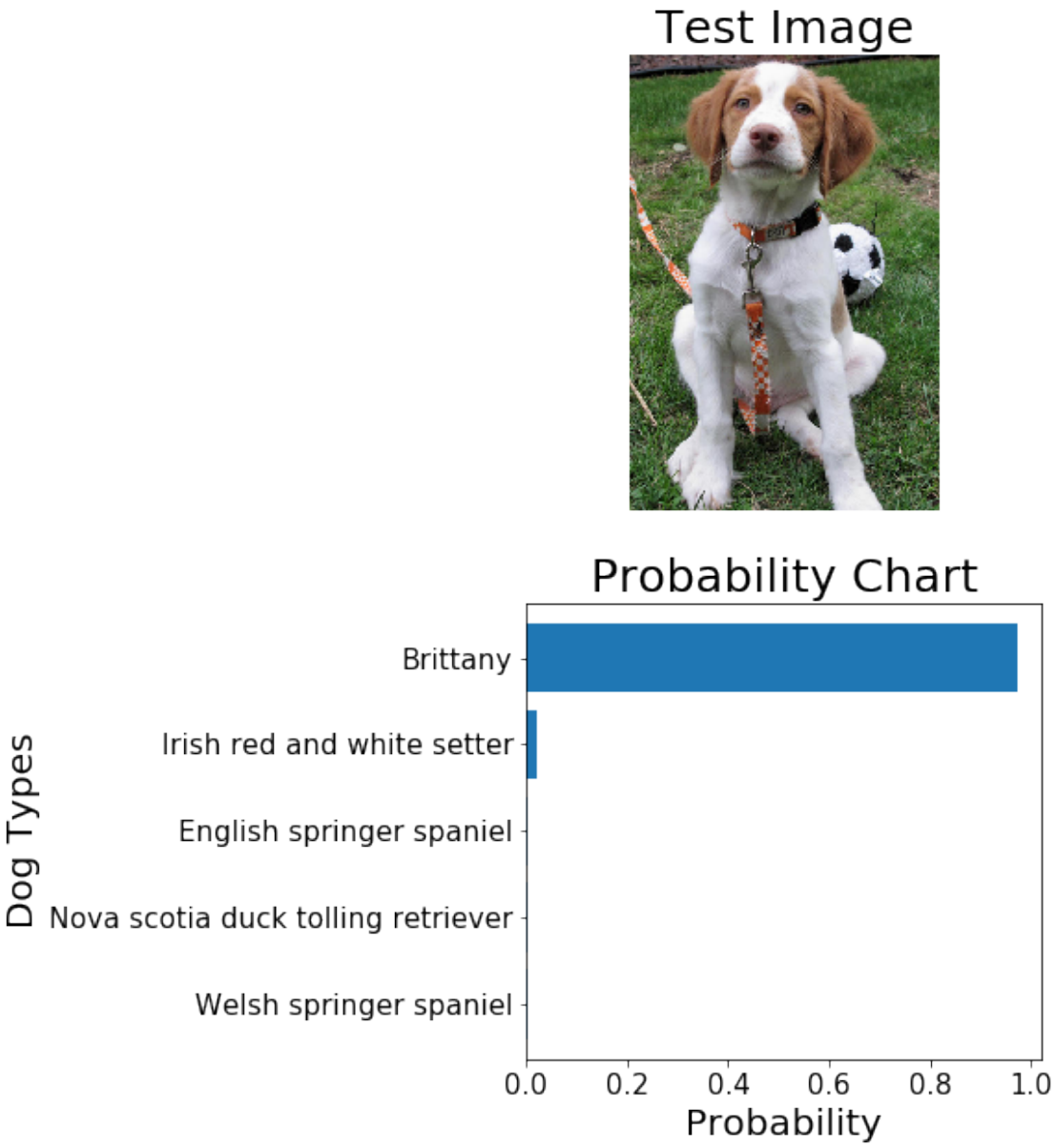
```
First 3 dog types:  ['Affenpinscher', 'Afghan hound', 'Airedale terrier']
```

```
In [45]: # Display an image along with the top 5 classes
         import matplotlib.pyplot as plt
         def view_classify(img_path, model):
             probs, classes = predict_breed_transfer(img_path, model)

             fig = plt.figure(figsize=(5, 10))
             ax1 = fig.add_subplot(2, 1, 1)
             ax1.axis('off')
             ax1.set_title('Test Image', fontsize = 25)
             image = Image.open(img_path)
             ax1.imshow(image)

             ax2 = fig.add_subplot(2, 1, 2)
             ax2.set_title('Probability Chart', fontsize = 25)
             ax2.set_xlabel('Probability', fontsize = 20)
             ax2.set_ylabel('Dog Types', fontsize = 20)
             ax2 = plt.yticks(range(5), classes[::-1], fontsize = 15)
             #probs[::-1], ex. a = [1,2,3] a[::-1] = [3,2,1]
             ax2 = plt.barh(range(5), probs[::-1])
             ax2 = plt.xticks(fontsize = 15)

             plt.show()
```

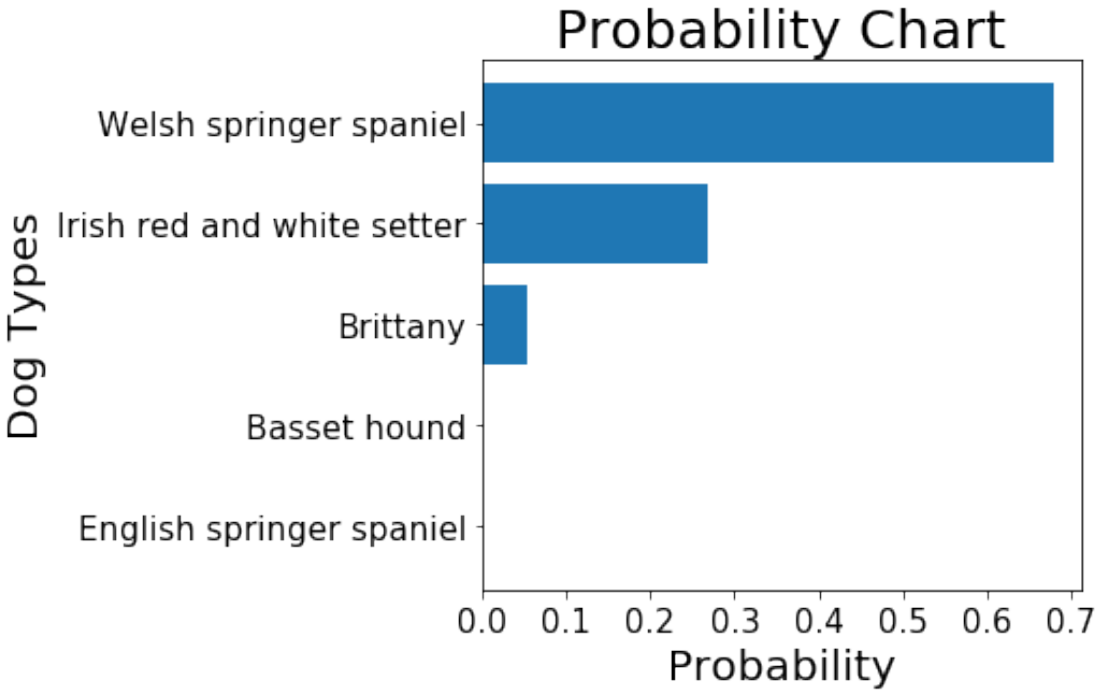
```
In [46]: img_path1 =  'images/Brittany_02625.jpg'
         probs, classes = predict_breed_transfer(img_path1, model_transfer)
         print('probs =', probs)
```

```
        print('class =', classes)

        view_classify(img_path1, model_transfer)
```

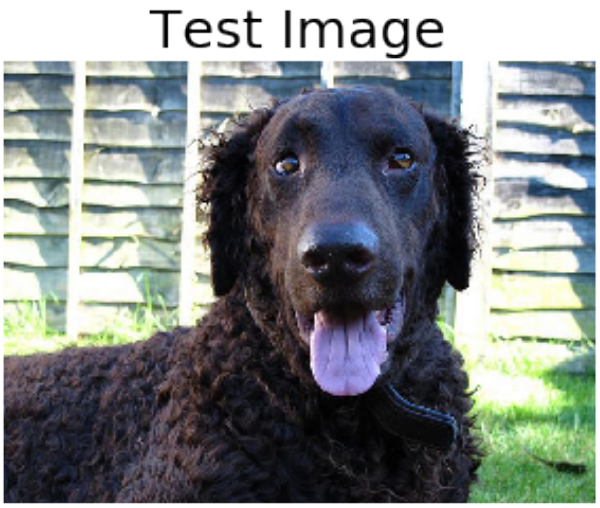
```
probs = [ 0.97355384  0.02070661  0.00204748  0.00156527  0.00116926]
class = ['Brittany', 'Irish red and white setter', 'English springer spaniel', 'Nova scotia duck
```

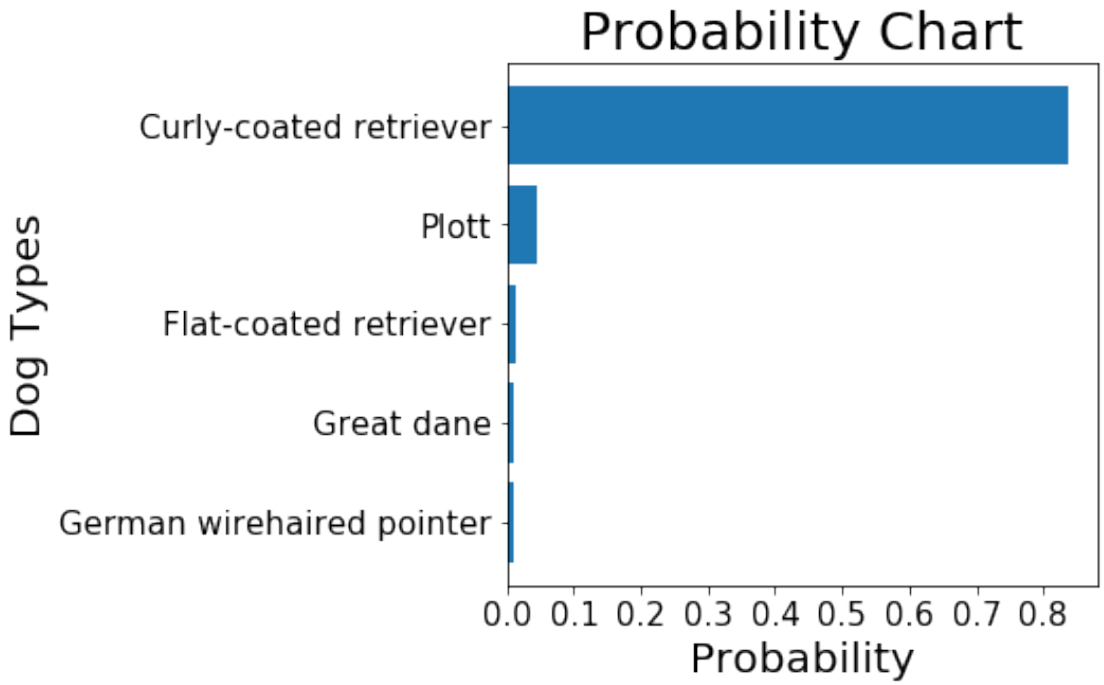


In [47]:
```
img_path2 = 'images/Welsh_springer_spaniel_08203.jpg'
view_classify(img_path2, model_transfer)
```

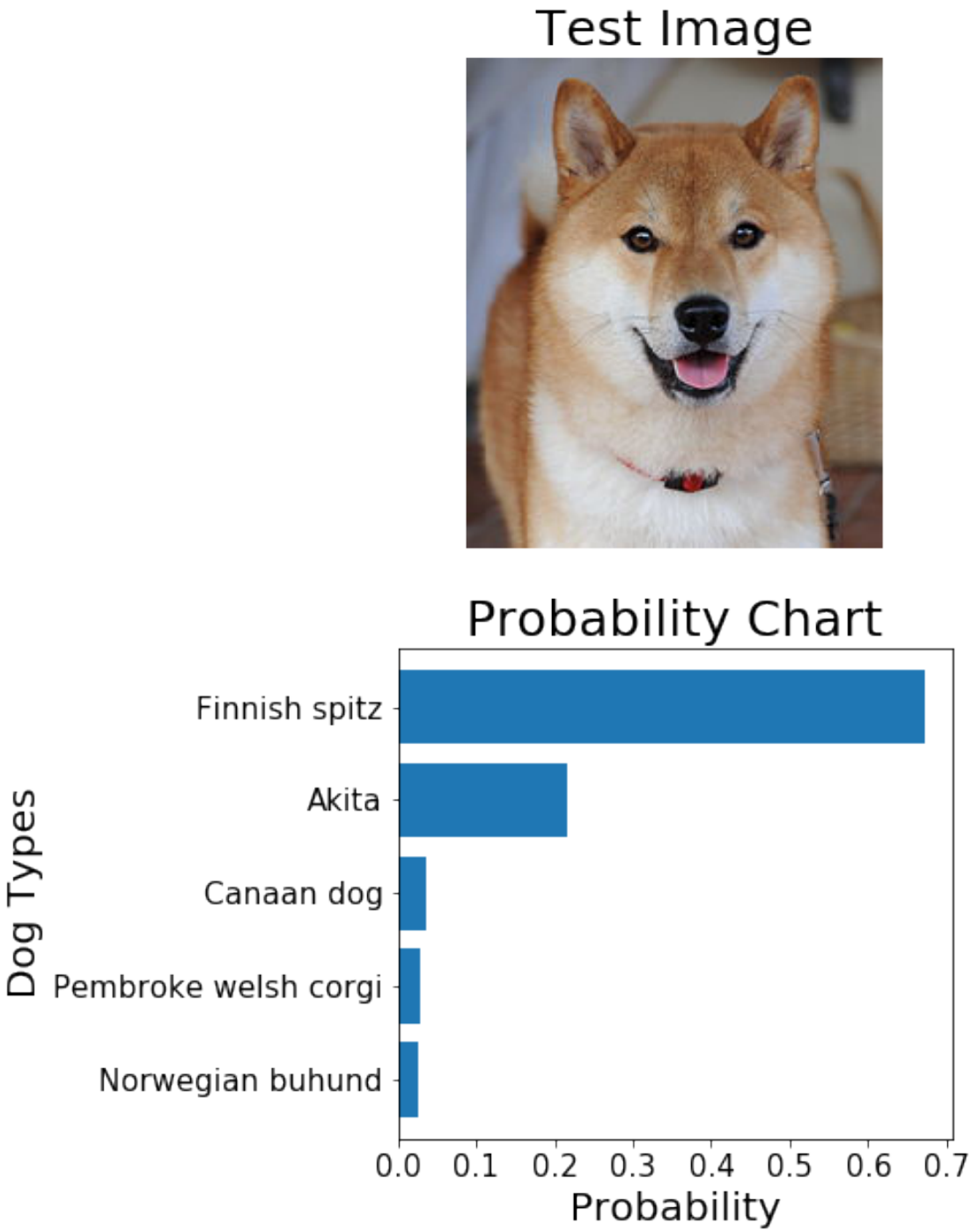



```
In [48]: img_path3 = 'images/Curly-coated_retriever_03896.jpg'
         view_classify(img_path3, model_transfer)
```



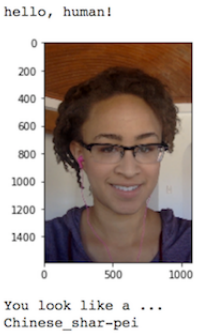



```
In [49]: img_path4 = 'images/220px-Shiba_inu_taiki.jpg'
         view_classify(img_path4, model_transfer)
```



Step 5: Write your Algorithm

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



Sample Human Output

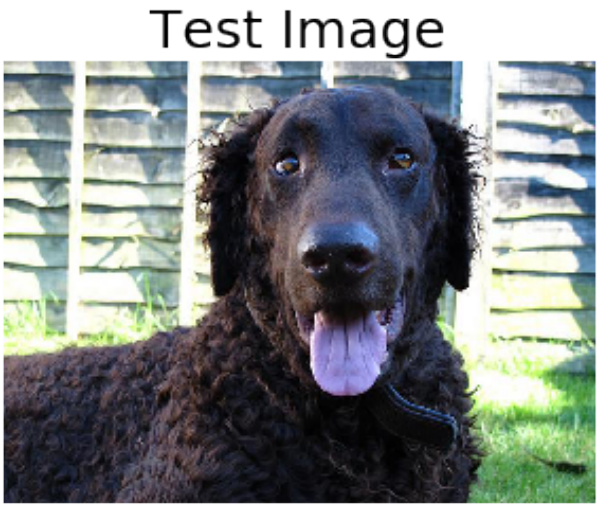
breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

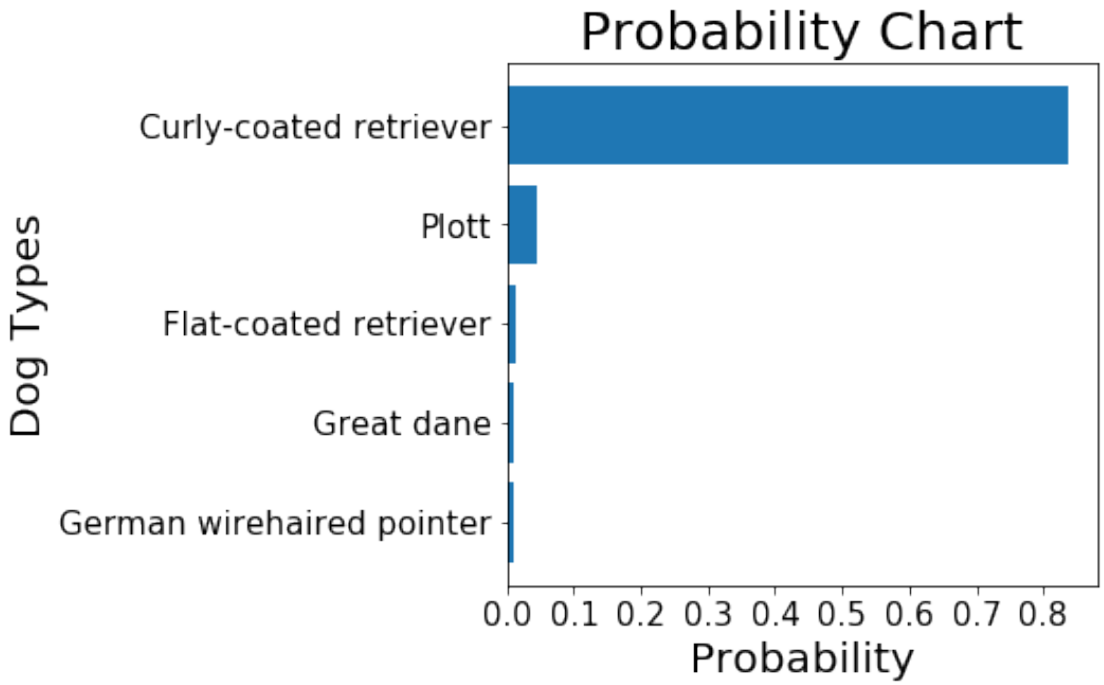
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

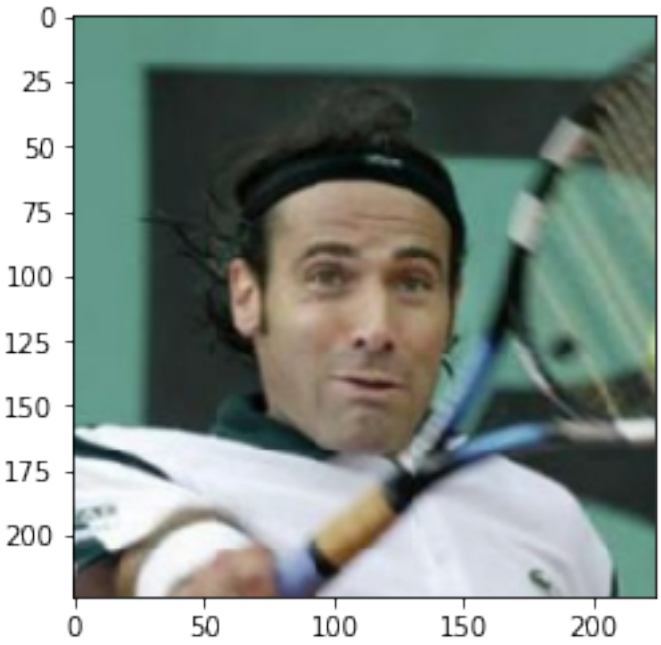
1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [50]: def run_app(img_path, model):
             ## handle cases for a human face, dog, and neither
             if dog_detector(img_path):
                 print("Hello doggy ~ :D \nbelow shows your potential breed: ")
                 view_classify(img_path, model)
             elif face_detector_dlib(img_path):
                 _, classes = predict_breed_transfer(img_path, model)
                 print("Hello Human ~ you look like a... : \n", classes[0])
                 plt.imshow(im_convert(load_image(img_path)))
                 plt.show()
             else:
                 print("Neither dog nor human:")
                 plt.imshow(im_convert(load_image(img_path)))
                 plt.show()

In [51]: img_path5 = 'images/Curly-coated_retriever_03896.jpg'
         run_app(img_path5, model_transfer)

Hello doggy ~ :D
below shows your potential breed:
```



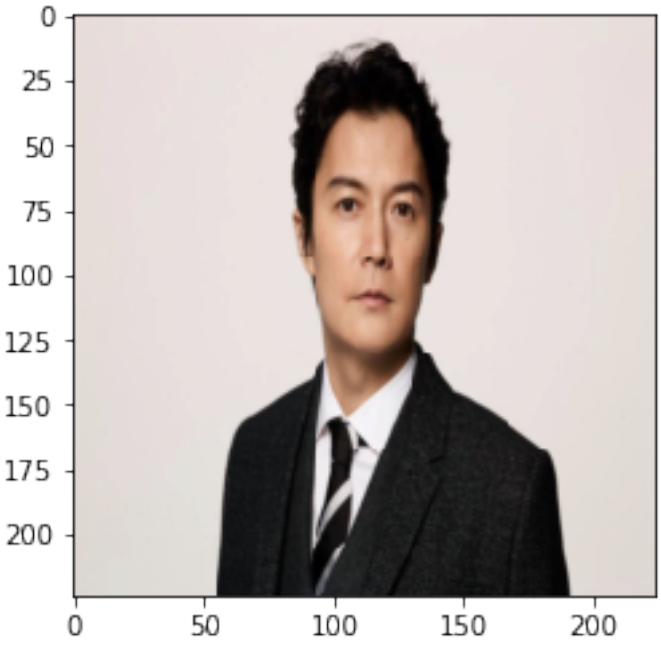



```
In [52]: run_app(human_files[1], model_transfer)

Hello Human ~ you look like a... :
 Italian greyhound
```



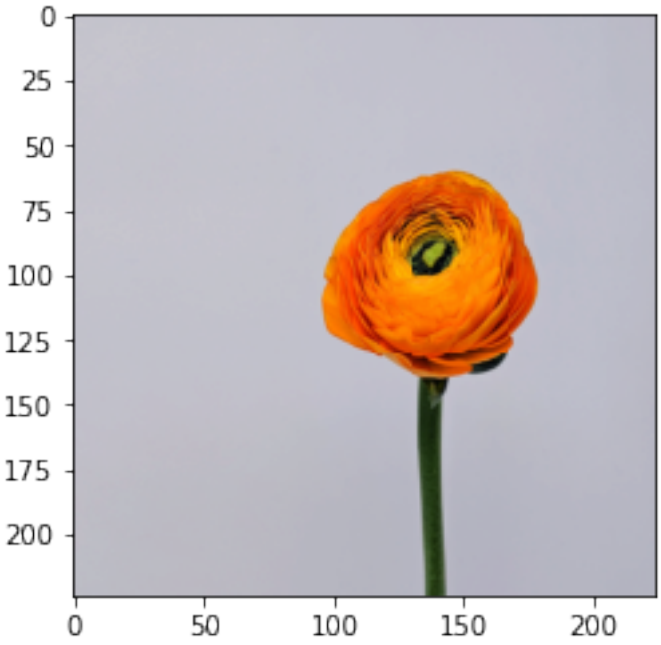

```
In [53]: img_path6 = 'images/human.jpg'
         run_app(img_path6, model_transfer)
```

```
Hello Human ~ you look like a... :
 German shorthaired pointer
```



```
In [54]: img_path7 = 'images/flower.jpg'
         run_app(img_path7, model_transfer)
```

```
Neither dog nor human:
```



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:

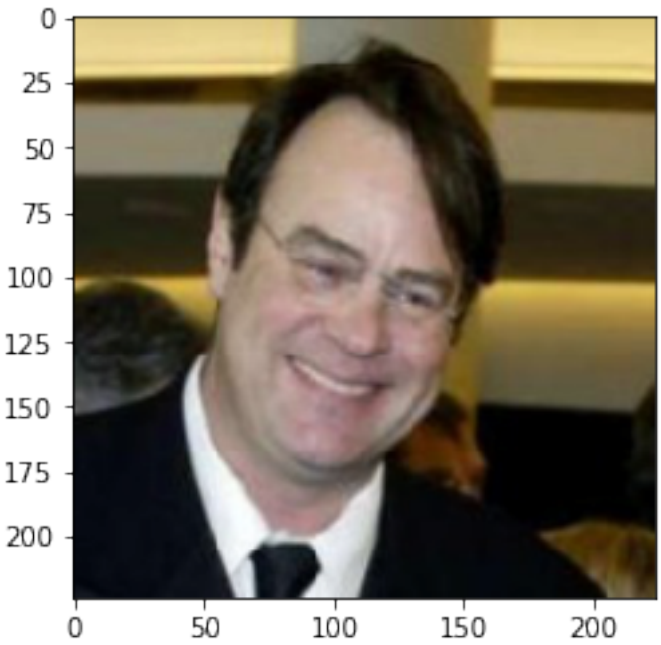
I think the results is better than I expect since I only take 10 epochs to train the classifier and it reach a over 80% accuracy. Plus, it doesn't mismatch Brittany and Welsh Springer Spaniel ;)

Three possible points of improvement: 1. Increase the training epochs to reach a higher accuracy 2. Implement the grid search to fine-tune hyper-parameters. I tried `skorch` to find a good

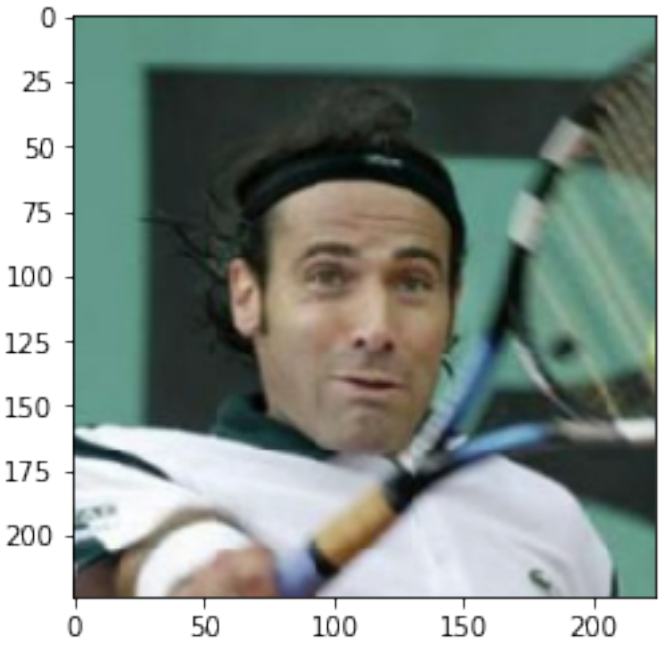
learning rate, but it seems that `skorch` got some problem lead it can not apply cross-entropy loss function during the grid search. 3. Increasing the dog database if we want to distinguish dogs more accuracy. Also, more image augmentation, such as flip, rotation, crop, channel shift, zoom in/out ...etc, would be helpful.

In [55]:
```
## 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, model_transfer)
```

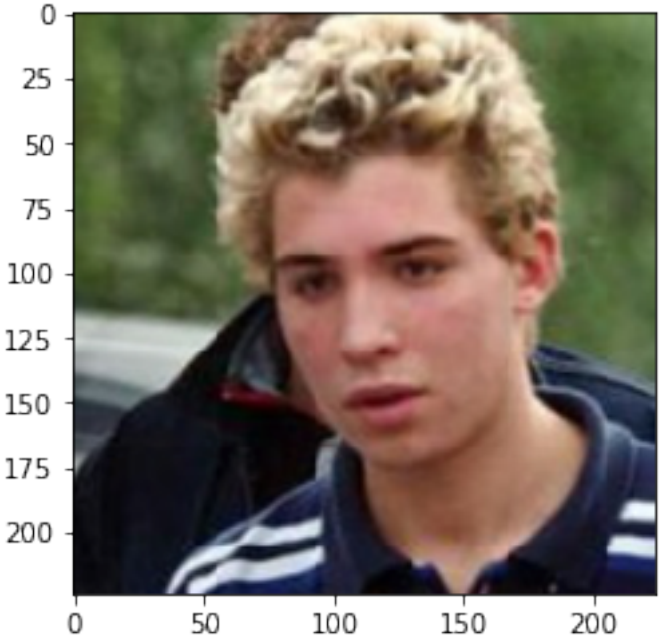
```
Hello Human ~ you look like a... :
 Chihuahua
```



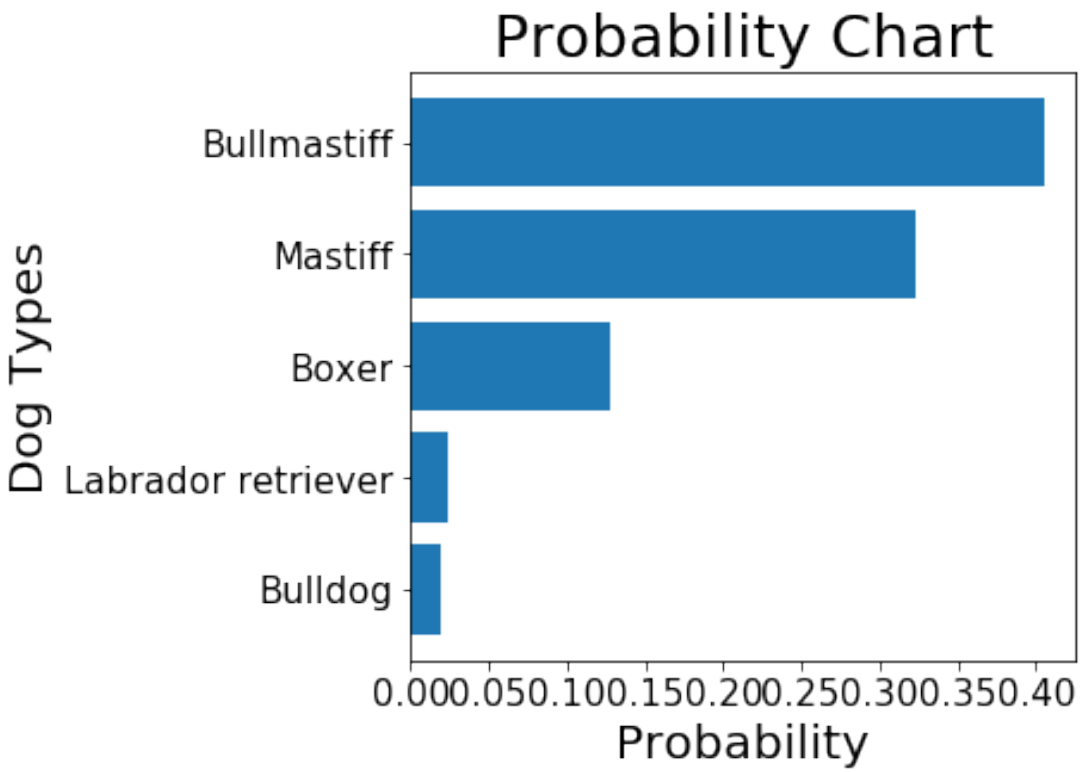

```
Hello Human ~ you look like a... :
 Italian greyhound
```



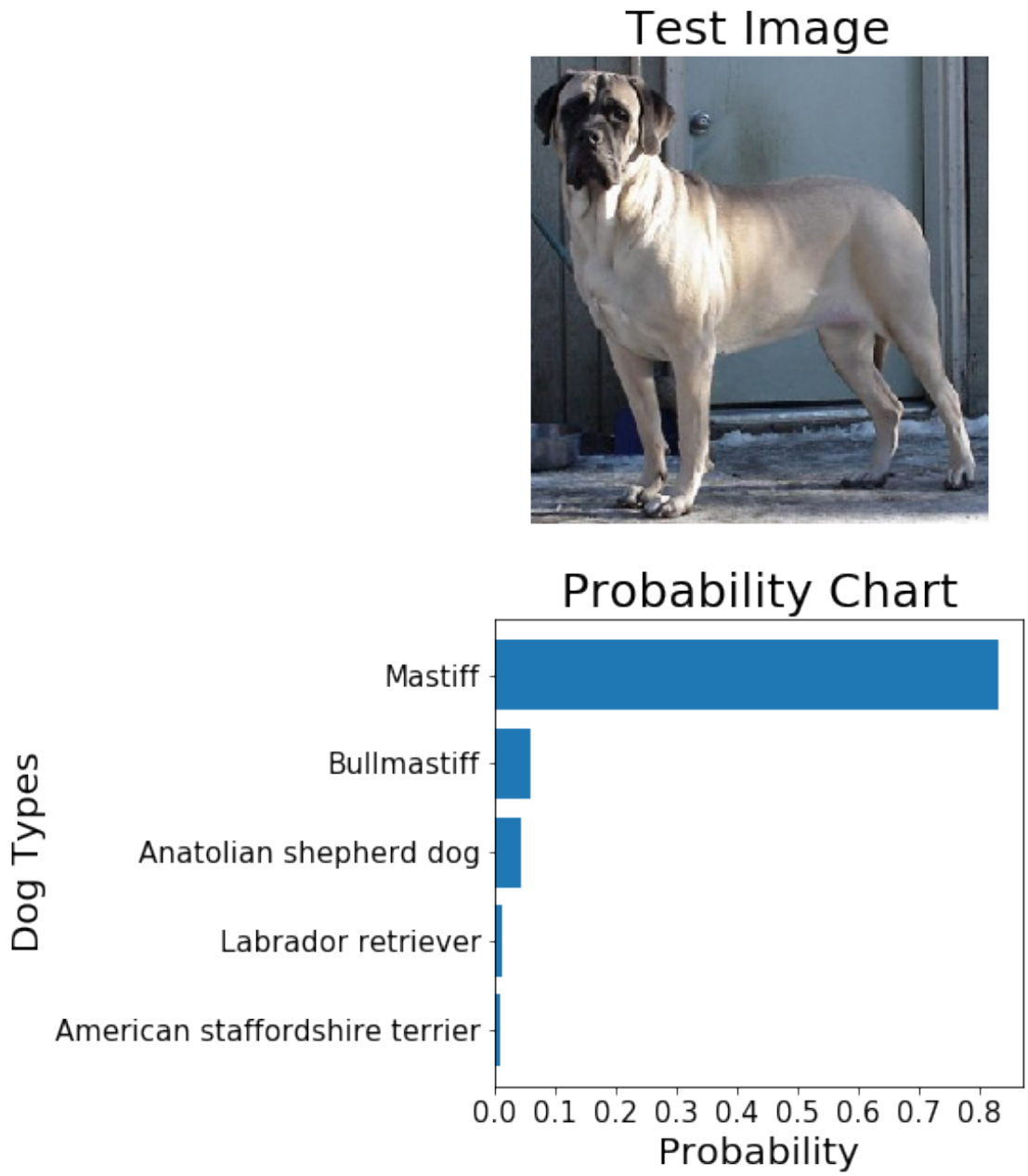

```
Hello Human ~ you look like a... :
 Poodle
```



```
Hello doggy ~ :D
below shows your potential breed:
```

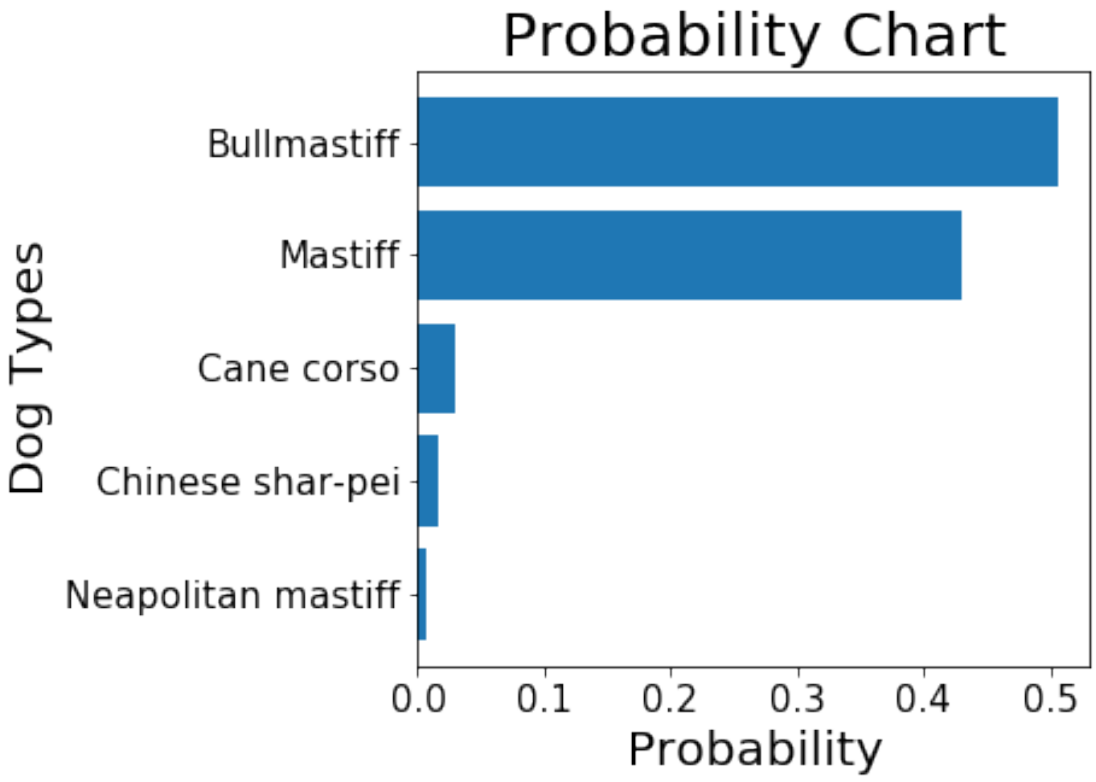


```
Hello doggy ~ :D
below shows your potential breed:
```




```
Hello doggy ~ :D
below shows your potential breed:
```


Test Image


Probability Chart
Bullmastiff
Mastiff
Cane corso
Chinese shar-pei
Neapolitan mastiff
Dog Types
0.0
0.1
0.2
0.3
0.4
0.5
Probability

In [ ]: