dog_app

February 26, 2021

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 [3]: import numpy as np
        from glob import glob

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

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

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

```
In [4]: human_files[0]
```

```
Out[4]: '/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg'
```

Step 1: Detect Humans

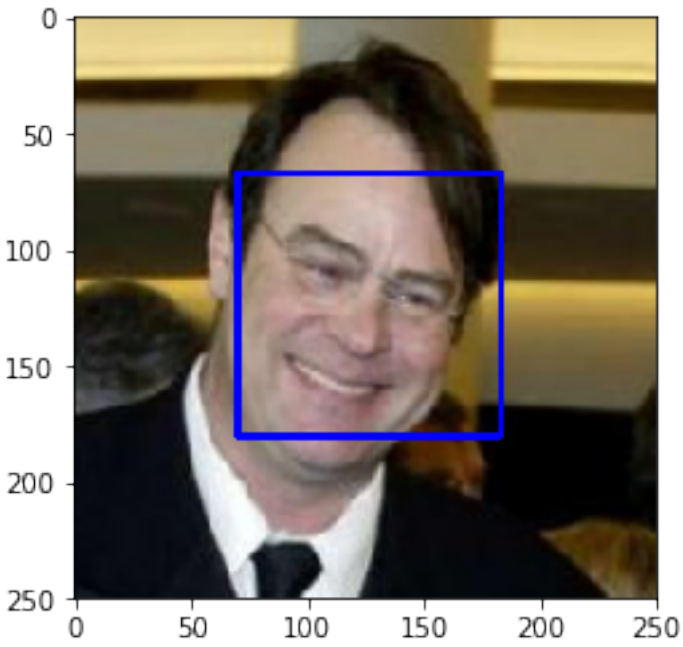
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [5]: 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
```




```
In [6]: faces
```

```
Out[6]: array([[ 70,  67, 113, 113]], dtype=int32)
```

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 [7]: # 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

In [8]: face_detector(human_files[0])

Out[8]: True
```

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: For the first 100 human faces, the face detector has an accuracy of 98% while for the first 100 dog pictures the accuracy of the detectior is 1%.

```
In [9]: 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.
        sum_hum=0
        sum_dog=0
        for hum_file,dog_file in zip(human_files_short,dog_files_short):
            if(face_detector(hum_file)>0):
                sum_hum+=1
```

```
            elif(face_detector(dog_file)>0):
                sum_dog+=1
        print("Accuracy on human image classfication: ",sum_hum/len(human_files_short)*100)
        print("Accuracy on dog image classfication: ",sum_dog/len(dog_files_short)*100)
```

```
Accuracy on human image classfication:  98.0
Accuracy on dog image classfication:  1.0
```

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

Step 2: Detect Dogs

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

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [11]: 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 [00:10<00:00, 53817783.76it/s]
```

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

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

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

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

In [12]:
```
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
    '''

    img = Image.open(img_path)
    mean = [0.485, 0.456, 0.406]
    std = [0.229, 0.224, 0.225]
    transform = transforms.Compose([transforms.Resize((224,224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=mean, std=std)])
    #print(type(img),img.size)
    img_tran = transform(img)
    if use_cuda:
        img_tran = img_tran.cuda()
    #print(type(img_tran),img_tran.shape,img_tran.unsqueeze(0).shape)
    with torch.no_grad():
        output = VGG16(img_tran.unsqueeze(0))
        output = torch.exp(output)
        pred = torch.argmax(output, 1) #the index of the class with maximum probablity
    print(pred)
    ## 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

    return pred # predicted class index
```

In [13]:
```
print(VGG16)
```

```
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)
  )
)
```

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 [14]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    val = VGG16_predict(img_path);
    if val >=151 and val <=268:
        #print("In dog detector",val)
        return True
    else:
        return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: For the fist 100 human faces, the dog detector has an accuracy of 1% while it has an accuracy of 100% for the first 100 dog pictures.

In [12]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
sum_hum = 0
sum_dog =0
for hu_file,dog_file in zip(human_files_short,dog_files_short):
    if(dog_detector(hu_file)==True):
        #print("human true: ",sum_hum)
        sum_hum+=1
    if(dog_detector(dog_file)==True):
        sum_dog+=1;
print("Accuracy on human image: ",sum_hum/len(human_files_short)*100)
print("Accuracy on dog imagess: ",sum_dog/len(dog_files_short)*100)
```

```
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 456], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
```

```
tensor([ 805], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 776], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 246], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 432], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 819], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 0], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 610], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 683], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 568], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 401], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 762], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 722], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 163], device='cuda:0')
tensor([ 683], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 982], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 917], device='cuda:0')
tensor([ 243], device='cuda:0')
```

```
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 255], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 823], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 578], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 209], device='cuda:0')
tensor([ 903], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 225], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 457], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 401], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 578], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 903], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 903], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 982], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 399], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 445], device='cuda:0')
tensor([ 243], device='cuda:0')
```

```
tensor([ 459], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 501], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 542], device='cuda:0')
tensor([ 243], device='cuda:0')
tensor([ 678], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 0], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 389], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 433], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 424], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 432], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 465], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 917], device='cuda:0')
tensor([ 236], device='cuda:0')
```

```
tensor([ 906], device='cuda:0')
tensor([ 234], device='cuda:0')
tensor([ 667], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 480], device='cuda:0')
tensor([ 168], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 513], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 610], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 906], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 617], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 233], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 652], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 834], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 722], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 903], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 683], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 465], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 683], device='cuda:0')
tensor([ 227], device='cuda:0')
tensor([ 430], device='cuda:0')
tensor([ 236], device='cuda:0')
tensor([ 400], device='cuda:0')
tensor([ 236], device='cuda:0')
```

```
Accuracy on human image:  1.0
Accuracy on dog imagess:  100.0


In [13]: sum_hum

Out[13]: 1
```

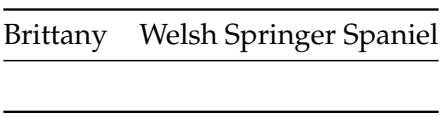
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

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

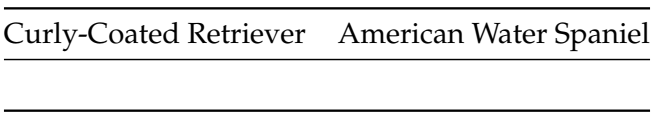
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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



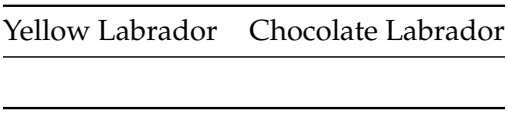
Brittany Welsh Springer Spaniel

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

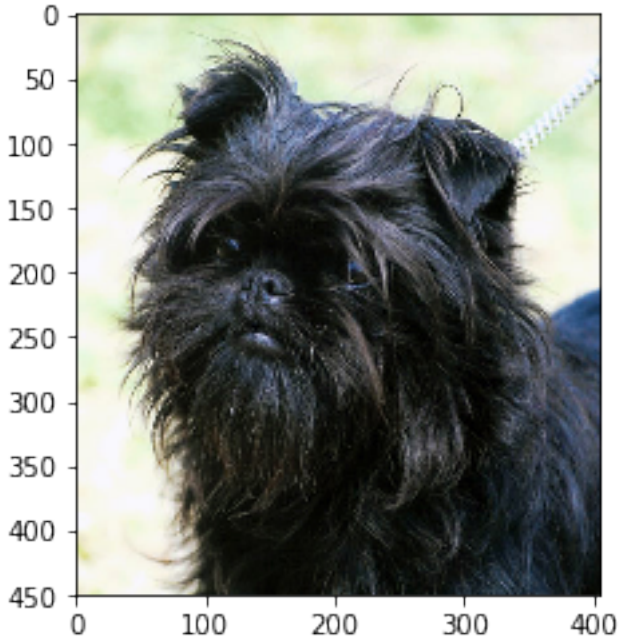
Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested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

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

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         testdir = "/data/dog_images/test"
         valdir = "/data/dog_images/valid"
         traindir =  "/data/dog_images/train"
         mean = [0.485, 0.456, 0.406]
         std = [0.229, 0.224, 0.225]
         #transforms.Normalize(mean,std)
         # data agumentation is needed only for the test set

         transform = {'train': transforms.Compose([
                                          transforms.RandomResizedCrop(224),
                                          transforms.RandomHorizontalFlip(),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406], [0.229
                      'val': transforms.Compose([
                                          transforms.Resize(size=(224,224)),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406], [0.229
                      'test': transforms.Compose([
                                          transforms.Resize(size=(224,224)),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406], [0.229
                        }

         testimage = datasets.ImageFolder(root=testdir, transform=transform['test'])
         trainimage = datasets.ImageFolder(root= traindir, transform = transform['train'])
         valimage = datasets.ImageFolder(root=valdir,transform= transform['val'])
```

```
        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        testloder = torch.utils.data.DataLoader(testimage,batch_size=64,shuffle=True)
        trainloder = torch.utils.data.DataLoader(trainimage, batch_size = 64, shuffle=True)
        valloder = torch.utils.data.DataLoader(valimage,batch_size=64, shuffle=True)
```

In [16]:
```
img = Image.open('/data/dog_images/test/001.Affenpinscher/Affenpinscher_00023.jpg')
tran = transforms.ToTensor()
imgT= tran(img)
imgn = imgT.numpy()
plt.imshow(np.transpose(imgn,(1,2,0)))
```

Out[16]:
```
<matplotlib.image.AxesImage at 0x7f2778bf5f98>
```



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?

With the aim of using the same dataloader to test VGG16 and Resnet50 networks, I decided to resize the image to 224x224. Furhter I perfromed random horizontal flip to avoid over fitting and data normalizaton for the optimizer to converge faster.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [16]:
```
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
        self.conv1 = nn.Conv2d(3,64,stride=2, kernel_size=3,padding=1)
        self.conv2 = nn.Conv2d(64,128, kernel_size= 3, stride=2,padding = 1)
        self.conv3 = nn.Conv2d(128,256, kernel_size = 3, padding =1)
        #self.conv4 = nn.Conv2d(256,512, kernel_size = 3, padding =1)
        self.maxpool = nn.MaxPool2d(2,stride=2)
        self.drop = nn.Dropout(0.3)
        self.fc1   = nn.Linear(256*7*7,512)
        self.fc2 = nn.Linear(512,256)
        self.fc3   = nn.Linear(256,133)
    def forward(self, x):
        ## Define forward behavior
        x = self.maxpool(F.relu(self.conv1(x)))
        x = self.maxpool(F.relu(self.conv2(x)))
        x = self.maxpool(F.relu(self.conv3(x)))
        #x = self.maxpool(F.relu(self.conv4(x)))
        x = x.view(-1,256*7*7)
        x = F.relu(self.fc1(x))
        x = self.drop(x)
        x = F.relu(self.fc2(x))
        x = self.drop(x)
        x = self.fc3(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

In [17]:
```
model_scratch
```

Out[17]:
```
Net(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
```

```
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxpool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop): Dropout(p=0.3)
  (fc1): Linear(in_features=12544, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=133, bias=True)
)
```

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

Answer: I took the inital idea for VGG16 network, but to save time insated of havding a depth of 16, I limited the depth to 6 and used a max-pooling with a stride of 2 at each conv-layer to downsample the input tensor. In addtion, for the first two Conv-net I used a stride of 2 to furhter downsample the input tensor. In the linear network, I used a dropout with a probablity of 0.3 to avoid oevrfitting. Finally, the fully connected layer has 133 number of nodes, which is the same us the dog breed class size.

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 [18]: import torch.optim as optim
         import numpy as np

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [17]: 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
             if use_cuda:
                 model = model.cuda()

             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 #
###################
#print('training started')
model.train()
for batch_idx, (data, target) in enumerate(loaders['train']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    optimizer.zero_grad()
    output = model(data)
    #print(output)
    #print(target)
    loss = criterion(output,target)
    loss.backward()
    optimizer.step()
    train_loss = train_loss + ((1/(batch_idx + 1)) * (loss.data - train_loss))
    ## 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

######################
# validate the model #
######################
#print('validation started')
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
    #with torch.no_grad():
    output = model(data)
    loss = criterion(output,target)
    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...'.form
    torch.save(model.state_dict(), save_path)
```

```
                    valid_loss_min = valid_loss

            # return trained model
            return model
```

```
In [30]: loaders_scratch ={'train':trainloder,'valid':valloder,'test':testloder}

         # train the model
         model_scratch = train(12, 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: 4.850848         Validation Loss: 4.660620
Validation loss decreased (inf --> 4.660620). Saving model...
Epoch: 2         Training Loss: 4.476706         Validation Loss: 4.334430
Validation loss decreased (4.660620 --> 4.334430). Saving model...
Epoch: 3         Training Loss: 4.271538         Validation Loss: 4.225333
Validation loss decreased (4.334430 --> 4.225333). Saving model...
Epoch: 4         Training Loss: 4.124819         Validation Loss: 4.197187
Validation loss decreased (4.225333 --> 4.197187). Saving model...
Epoch: 5         Training Loss: 4.006302         Validation Loss: 3.951389
Validation loss decreased (4.197187 --> 3.951389). Saving model...
Epoch: 6         Training Loss: 3.854568         Validation Loss: 3.996880
Epoch: 7         Training Loss: 3.729747         Validation Loss: 3.888849
Validation loss decreased (3.951389 --> 3.888849). Saving model...
Epoch: 8         Training Loss: 3.588007         Validation Loss: 3.760735
Validation loss decreased (3.888849 --> 3.760735). Saving model...
Epoch: 9         Training Loss: 3.460969         Validation Loss: 3.889879
Epoch: 10         Training Loss: 3.281805         Validation Loss: 3.676277
Validation loss decreased (3.760735 --> 3.676277). Saving model...
Epoch: 11         Training Loss: 3.118114         Validation Loss: 3.622269
Validation loss decreased (3.676277 --> 3.622269). Saving model...
Epoch: 12         Training Loss: 2.922322         Validation Loss: 3.727202
```

```
In [ ]: #from workspace_utils import active_session
        !ls
```

```
In [20]: loaders_scratch ={'train':trainloder,'valid':valloder,'test':testloder}
         loaders_scratch['train']
```

```
Out[20]: <torch.utils.data.dataloader.DataLoader at 0x7f2880391d30>
```

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 [27]: def test(loaders, model, criterion, use_cuda):

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

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

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

             print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
                 100. * (correct*1.0) / total, correct, total))

In [31]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.757013

correct value {} 105.0

Test Accuracy: 12% (105/836)
```

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`, re-

spectively).

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

```
In [18]: ## TODO: Specify data loaders
         testloder = torch.utils.data.DataLoader(testimage,batch_size=64,shuffle=True)
         trainloder = torch.utils.data.DataLoader(trainimage, batch_size = 64, shuffle=True)
         valloder = torch.utils.data.DataLoader(valimage,batch_size=64, shuffle=True)

In [19]: use_cuda

Out[19]: True
```

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

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

         if use_cuda:
             model_transfer = model_transfer.cuda()

Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 63413931.47it/s]


In [24]: model_transfer.fc

Out[24]: Linear(in_features=2048, out_features=1000, bias=True)

In [21]: #freeze the parametes
         for param in model_transfer.parameters():
             param.requires_grad = False
         # modify the last output last  by mofying the output to 133 classes
         model_transfer.fc = nn.Linear(2048, 133, bias=True)

         # unfreeze the last layer parameter
         for param in model_transfer.fc.parameters():
             param.requires_grad = True
```

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

Answer: From the experment in step 4, limiting the number of layers signifcatly affects the accuracy. However, as the number of layers increase the vanishing gradinet problem arise. Since ResNet addresses this issue by employing shottcut connection and has been proved to perfrom very well, I selected to use ResNet50 network for this experment.

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 [22]: import torch.optim as optim
         import numpy as np

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

```
In [23]: print(model_transfer.cuda())
```

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

```
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

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

```
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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 [24]:
```
loaders_transfer ={'train':trainloder,'valid':valloder,'test':testloder}
 n_epochs = 35
```

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

# 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: 4.740268         Validation Loss: 4.678840
Validation loss decreased (inf --> 4.678840). Saving model...
Epoch: 2         Training Loss: 4.654725         Validation Loss: 4.587216
Validation loss decreased (4.678840 --> 4.587216). Saving model...
Epoch: 3         Training Loss: 4.581307         Validation Loss: 4.469493
Validation loss decreased (4.587216 --> 4.469493). Saving model...
Epoch: 4         Training Loss: 4.502668         Validation Loss: 4.421869
Validation loss decreased (4.469493 --> 4.421869). Saving model...
Epoch: 5         Training Loss: 4.433087         Validation Loss: 4.303390
Validation loss decreased (4.421869 --> 4.303390). Saving model...
Epoch: 6         Training Loss: 4.362506         Validation Loss: 4.259737
Validation loss decreased (4.303390 --> 4.259737). Saving model...
Epoch: 7         Training Loss: 4.287757         Validation Loss: 4.162004
Validation loss decreased (4.259737 --> 4.162004). Saving model...
Epoch: 8         Training Loss: 4.218750         Validation Loss: 4.099207
Validation loss decreased (4.162004 --> 4.099207). Saving model...
Epoch: 9         Training Loss: 4.159291         Validation Loss: 4.045194
Validation loss decreased (4.099207 --> 4.045194). Saving model...
Epoch: 10         Training Loss: 4.089224         Validation Loss: 3.935532
Validation loss decreased (4.045194 --> 3.935532). Saving model...
Epoch: 11         Training Loss: 4.025490         Validation Loss: 3.890416
Validation loss decreased (3.935532 --> 3.890416). Saving model...
Epoch: 12         Training Loss: 3.956934         Validation Loss: 3.790433
Validation loss decreased (3.890416 --> 3.790433). Saving model...
Epoch: 13         Training Loss: 3.891143         Validation Loss: 3.751597
Validation loss decreased (3.790433 --> 3.751597). Saving model...
Epoch: 14         Training Loss: 3.835974         Validation Loss: 3.704716
Validation loss decreased (3.751597 --> 3.704716). Saving model...
Epoch: 15         Training Loss: 3.778022         Validation Loss: 3.606452
Validation loss decreased (3.704716 --> 3.606452). Saving model...
Epoch: 16         Training Loss: 3.710493         Validation Loss: 3.534103
Validation loss decreased (3.606452 --> 3.534103). Saving model...
Epoch: 17         Training Loss: 3.659791         Validation Loss: 3.486327
Validation loss decreased (3.534103 --> 3.486327). Saving model...
```

```
Epoch: 18         Training Loss: 3.601212         Validation Loss: 3.367840
Validation loss decreased (3.486327 --> 3.367840). Saving model...
Epoch: 19         Training Loss: 3.535548         Validation Loss: 3.369207
Epoch: 20         Training Loss: 3.483319         Validation Loss: 3.272543
Validation loss decreased (3.367840 --> 3.272543). Saving model...
Epoch: 21         Training Loss: 3.429479         Validation Loss: 3.203476
Validation loss decreased (3.272543 --> 3.203476). Saving model...
Epoch: 22         Training Loss: 3.375171         Validation Loss: 3.133888
Validation loss decreased (3.203476 --> 3.133888). Saving model...
Epoch: 23         Training Loss: 3.331851         Validation Loss: 3.072609
Validation loss decreased (3.133888 --> 3.072609). Saving model...
Epoch: 24         Training Loss: 3.274768         Validation Loss: 3.067329
Validation loss decreased (3.072609 --> 3.067329). Saving model...
Epoch: 25         Training Loss: 3.239724         Validation Loss: 2.998955
Validation loss decreased (3.067329 --> 2.998955). Saving model...
Epoch: 26         Training Loss: 3.179961         Validation Loss: 2.918651
Validation loss decreased (2.998955 --> 2.918651). Saving model...
Epoch: 27         Training Loss: 3.133985         Validation Loss: 2.888261
Validation loss decreased (2.918651 --> 2.888261). Saving model...
Epoch: 28         Training Loss: 3.080050         Validation Loss: 2.826465
Validation loss decreased (2.888261 --> 2.826465). Saving model...
Epoch: 29         Training Loss: 3.046877         Validation Loss: 2.789165
Validation loss decreased (2.826465 --> 2.789165). Saving model...
Epoch: 30         Training Loss: 3.001011         Validation Loss: 2.697955
Validation loss decreased (2.789165 --> 2.697955). Saving model...
Epoch: 31         Training Loss: 2.958932         Validation Loss: 2.703171
Epoch: 32         Training Loss: 2.907220         Validation Loss: 2.650649
Validation loss decreased (2.697955 --> 2.650649). Saving model...
Epoch: 33         Training Loss: 2.873359         Validation Loss: 2.632828
Validation loss decreased (2.650649 --> 2.632828). Saving model...
Epoch: 34         Training Loss: 2.826351         Validation Loss: 2.605716
Validation loss decreased (2.632828 --> 2.605716). Saving model...
```

```
In [30]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
In [25]: model_transfer.fc.type
```

```
Out[25]: <bound method Module.type of Linear(in_features=2048, out_features=133, bias=True)>
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [31]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 2.490567
```

```
correct value {} 569.0

Test Accuracy: 68% (569/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

```
In [40]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         data_transfer ={'train':trainloder,'valid':valloder,'test':testloder}
         # 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 data_transfer['train'].dataset.cl

         def load_input_image(img_path):
             image = Image.open(img_path).convert('RGB')

             transform = transforms.Compose([
                 transforms.Resize((224,224)),
                 transforms.ToTensor(),
                 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
             image_tran = transform(image)
             return image_tran

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img_tran = load_input_image(img_path)
             if use_cuda:
                 img_tran = img_tran.cuda()
             #print(type(img_tran),img_tran.shape,img_tran.unsqueeze(0).shape)
             with torch.no_grad():
                 output = model_transfer(img_tran.unsqueeze(0))
                 output = torch.exp(output)
                 indx = torch.argmax(output, 1) #the index of the class with maximum probablity
             return class_names[indx]
```

```
In [41]: data_transfer['train'].dataset.classes
```

```
Out[41]: ['001.Affenpinscher',
          '002.Afghan_hound',
          '003.Airedale_terrier',
          '004.Akita',
          '005.Alaskan_malamute',
          '006.American_eskimo_dog',
          '007.American_foxhound',
          '008.American_staffordshire_terrier',
          '009.American_water_spaniel',
```

```
'010.Anatolian_shepherd_dog',
'011.Australian_cattle_dog',
'012.Australian_shepherd',
'013.Australian_terrier',
'014.Basenji',
'015.Basset_hound',
'016.Beagle',
'017.Bearded_collie',
'018.Beauceron',
'019.Bedlington_terrier',
'020.Belgian_malinois',
'021.Belgian_sheepdog',
'022.Belgian_tervuren',
'023.Bernese_mountain_dog',
'024.Bichon_frise',
'025.Black_and_tan_coonhound',
'026.Black_russian_terrier',
'027.Bloodhound',
'028.Bluetick_coonhound',
'029.Border_collie',
'030.Border_terrier',
'031.Borzoi',
'032.Boston_terrier',
'033.Bouvier_des_flandres',
'034.Boxer',
'035.Boykin_spaniel',
'036.Briard',
'037.Brittany',
'038.Brussels_griffon',
'039.Bull_terrier',
'040.Bulldog',
'041.Bullmastiff',
'042.Cairn_terrier',
'043.Canaan_dog',
'044.Cane_corso',
'045.Cardigan_welsh_corgi',
'046.Cavalier_king_charles_spaniel',
'047.Chesapeake_bay_retriever',
'048.Chihuahua',
'049.Chinese_crested',
'050.Chinese_shar-pei',
'051.Chow_chow',
'052.Clumber_spaniel',
'053.Cocker_spaniel',
'054.Collie',
'055.Curly-coated_retriever',
'056.Dachshund',
'057.Dalmatian',
```

```
'058.Dandie_dinmont_terrier',
'059.Doberman_pinscher',
'060.Dogue_de_bordeaux',
'061.English_cocker_spaniel',
'062.English_setter',
'063.English_springer_spaniel',
'064.English_toy_spaniel',
'065.Entlebucher_mountain_dog',
'066.Field_spaniel',
'067.Finnish_spitz',
'068.Flat-coated_retriever',
'069.French_bulldog',
'070.German_pinscher',
'071.German_shepherd_dog',
'072.German_shorthaired_pointer',
'073.German_wirehaired_pointer',
'074.Giant_schnauzer',
'075.Glen_of_imaal_terrier',
'076.Golden_retriever',
'077.Gordon_setter',
'078.Great_dane',
'079.Great_pyrenees',
'080.Greater_swiss_mountain_dog',
'081.Greyhound',
'082.Havanese',
'083.Ibizan_hound',
'084.Icelandic_sheepdog',
'085.Irish_red_and_white_setter',
'086.Irish_setter',
'087.Irish_terrier',
'088.Irish_water_spaniel',
'089.Irish_wolfhound',
'090.Italian_greyhound',
'091.Japanese_chin',
'092.Keeshond',
'093.Kerry_blue_terrier',
'094.Komondor',
'095.Kuvasz',
'096.Labrador_retriever',
'097.Lakeland_terrier',
'098.Leonberger',
'099.Lhasa_apso',
'100.Lowchen',
'101.Maltese',
'102.Manchester_terrier',
'103.Mastiff',
'104.Miniature_schnauzer',
'105.Neapolitan_mastiff',
```

```
'106.Newfoundland',
'107.Norfolk_terrier',
'108.Norwegian_buhund',
'109.Norwegian_elkhound',
'110.Norwegian_lundehund',
'111.Norwich_terrier',
'112.Nova_scotia_duck_tolling_retriever',
'113.Old_english_sheepdog',
'114.Otterhound',
'115.Papillon',
'116.Parson_russell_terrier',
'117.Pekingese',
'118.Pembroke_welsh_corgi',
'119.Petit_basset_griffon_vendeen',
'120.Pharaoh_hound',
'121.Plott',
'122.Pointer',
'123.Pomeranian',
'124.Poodle',
'125.Portuguese_water_dog',
'126.Saint_bernard',
'127.Silky_terrier',
'128.Smooth_fox_terrier',
'129.Tibetan_mastiff',
'130.Welsh_springer_spaniel',
'131.Wirehaired_pointing_griffon',
'132.Xoloitzcuintli',
'133.Yorkshire_terrier']
```

Step 5: Write your Algorithm

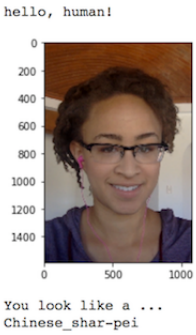
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [42]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
```



Sample Human Output

```
## handle cases for a human face, dog, and neither
#img = load_input_image(img_path)
#plt.imshow(img)
#plt.show()
img = Image.open(img_path)
plt.imshow(img)
plt.show()
if dog_detector(img_path) == True:
    class_name = predict_breed_transfer(img_path)
    print("Dog with breed {} detected".format(class_name))
elif face_detector(img_path) == True:
    print("Hello, human")
    class_name = predict_breed_transfer(img_path)
    print("You resumble a {} breed".format(class_name))
else:
    print("Error: no face or dog detected")
```

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!

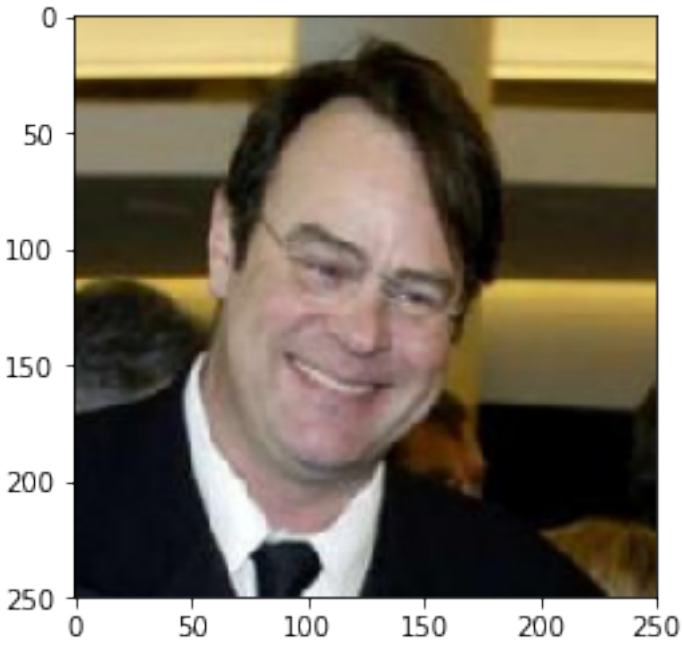
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

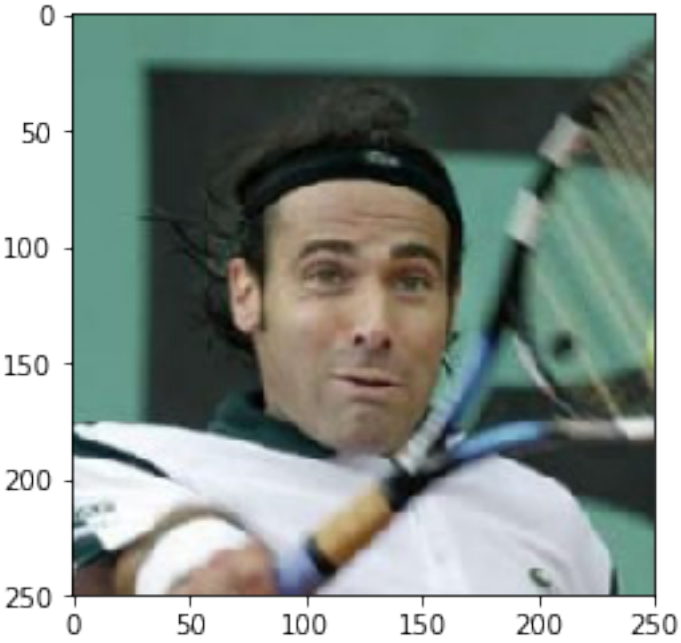
Answer: I belive the algorithm worked relatively well. However the algorithm can be furher improved by: 1. Tranning the model for more epochs, I only used number of ephos = 35 2. Test other variants of ResNet architecture with more depth, like ResNet101 3. Introduce more augmentation techniques

```
In [43]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
```

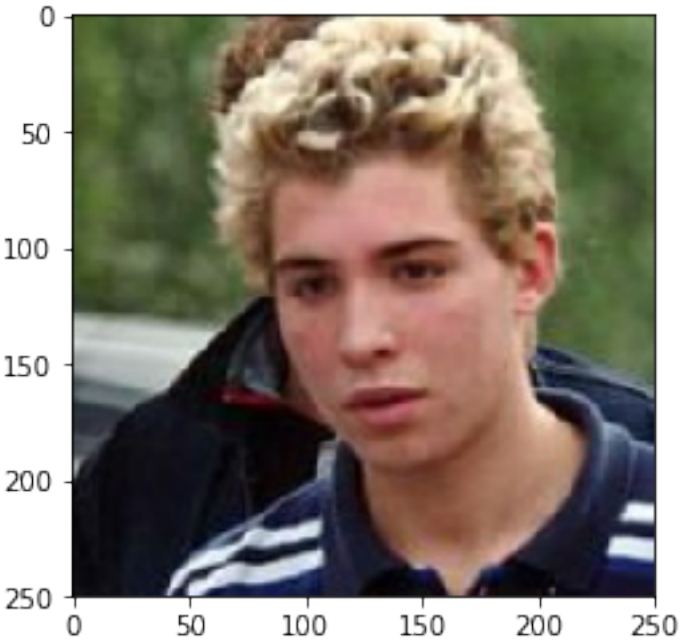
```
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[:20], dog_files[:20])):
    run_app(file)
```



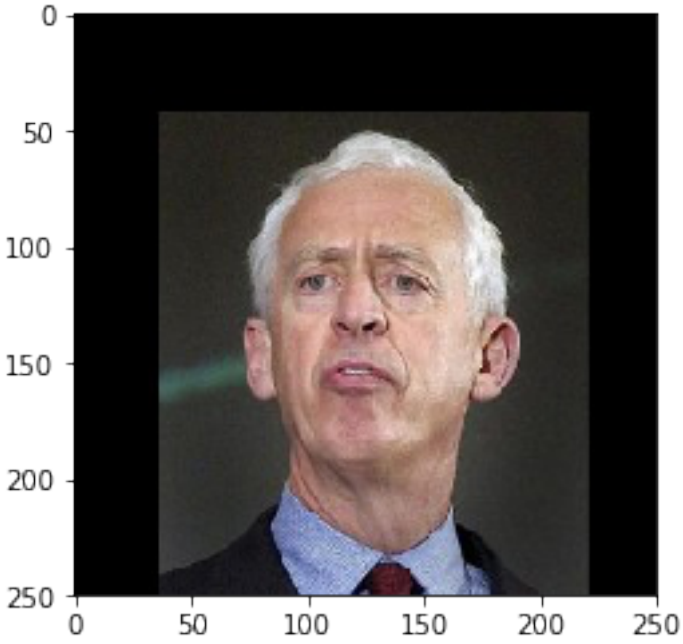

```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



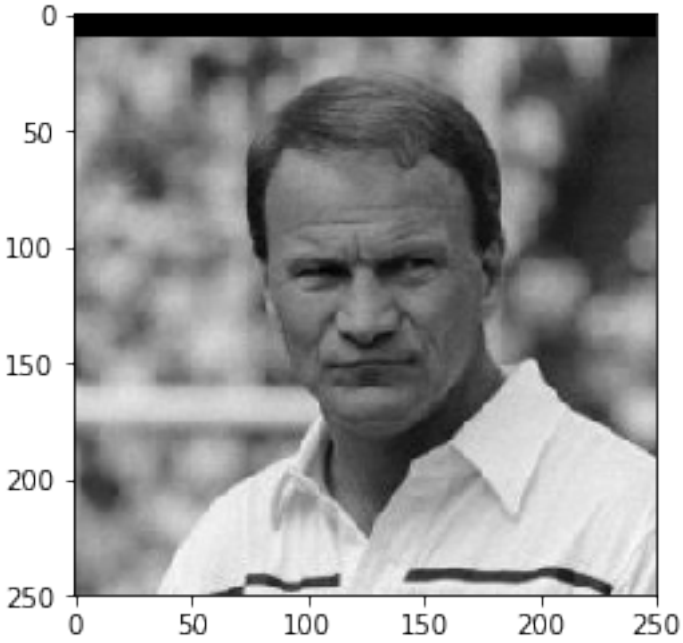

```
tensor([ 456], device='cuda:0')
Hello, human
You resumble a American staffordshire terrier breed
```



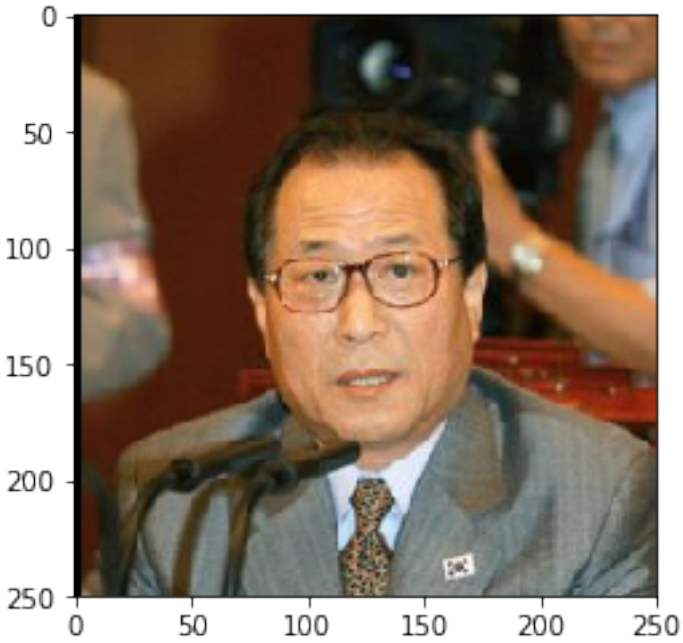
```
tensor([ 400], device='cuda:0')
Hello, human
You resumble a Cane corso breed
```



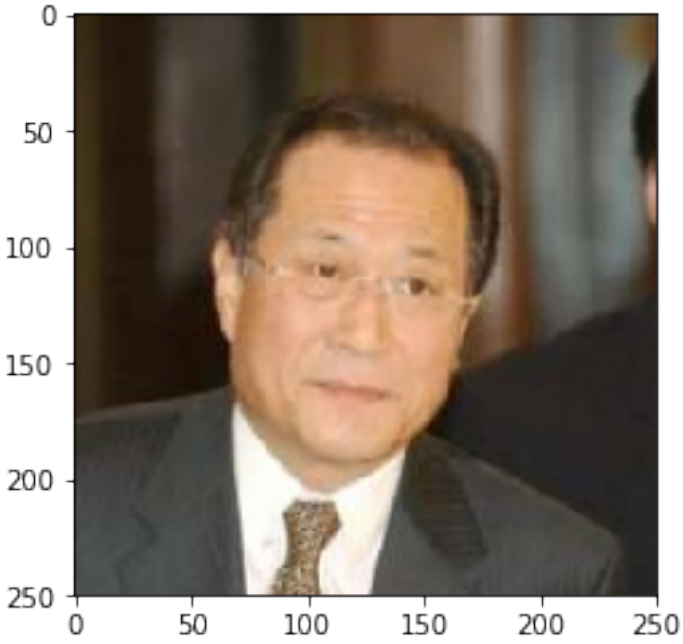

```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



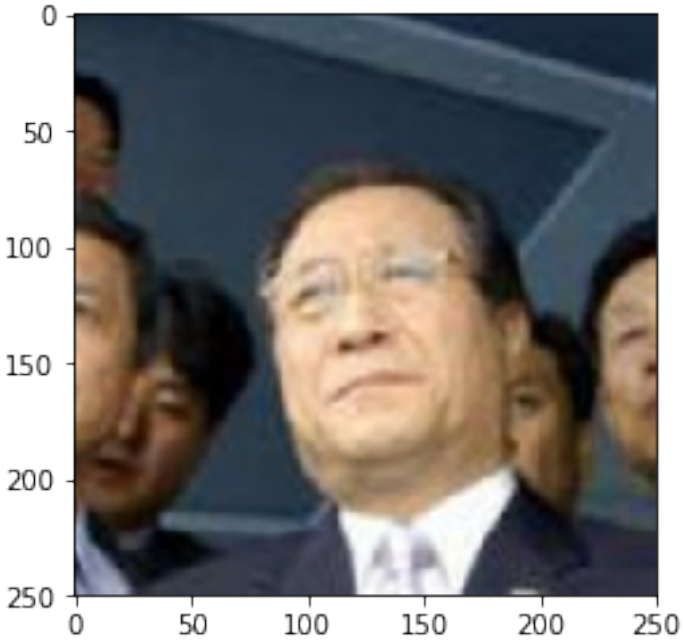

```
tensor([ 429], device='cuda:0')
Hello, human
You resumble a Bullmastiff breed
```



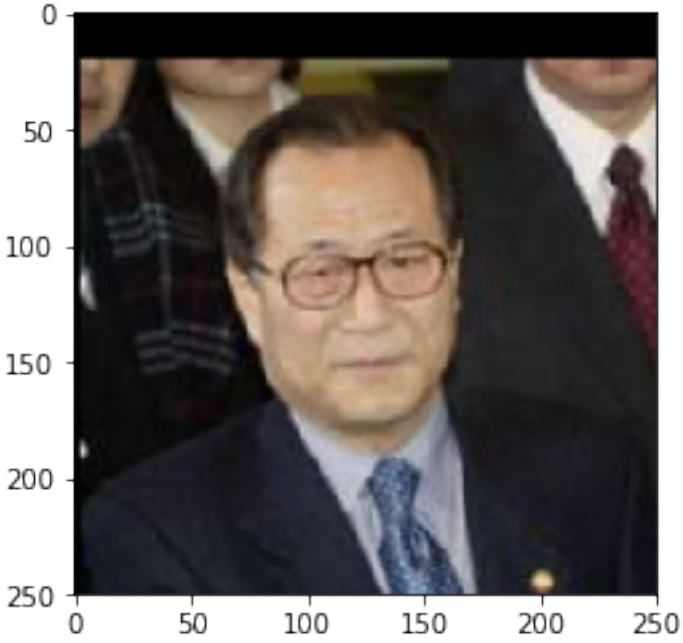
```
tensor([ 642], device='cuda:0')
Hello, human
You resumble a Cavalier king charles spaniel breed
```



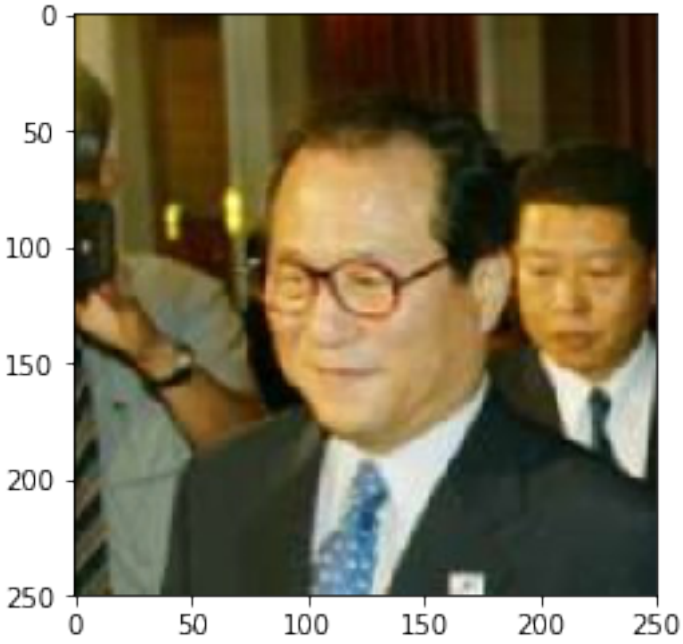

```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



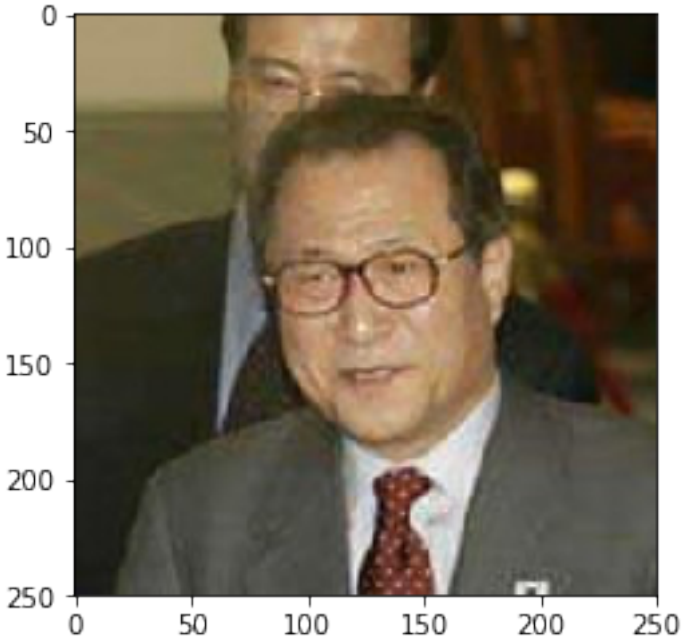

```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Bull terrier breed
```



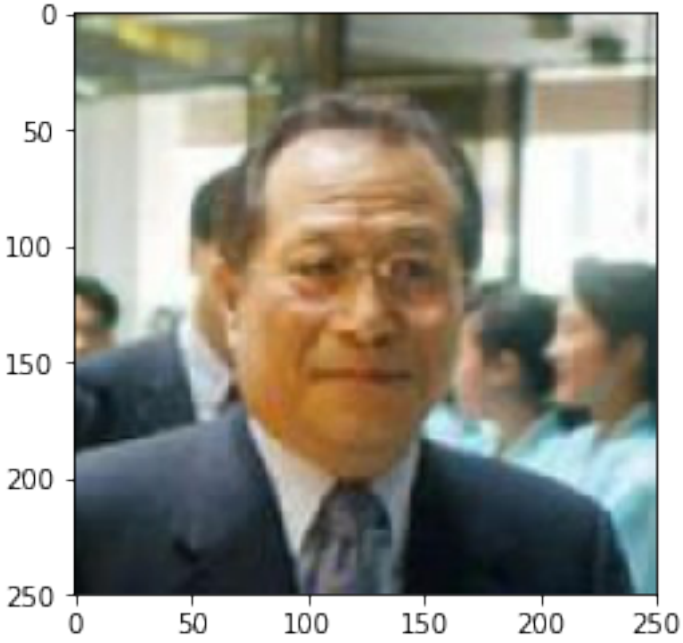
```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



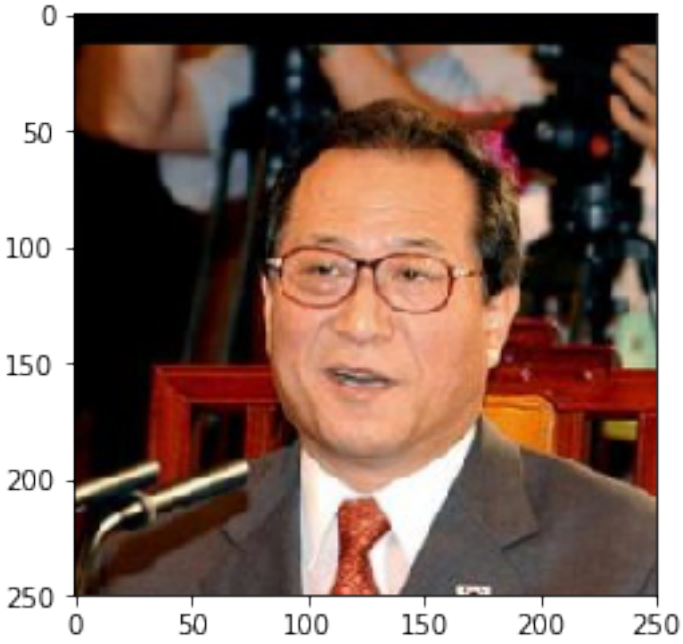

```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Bloodhound breed
```



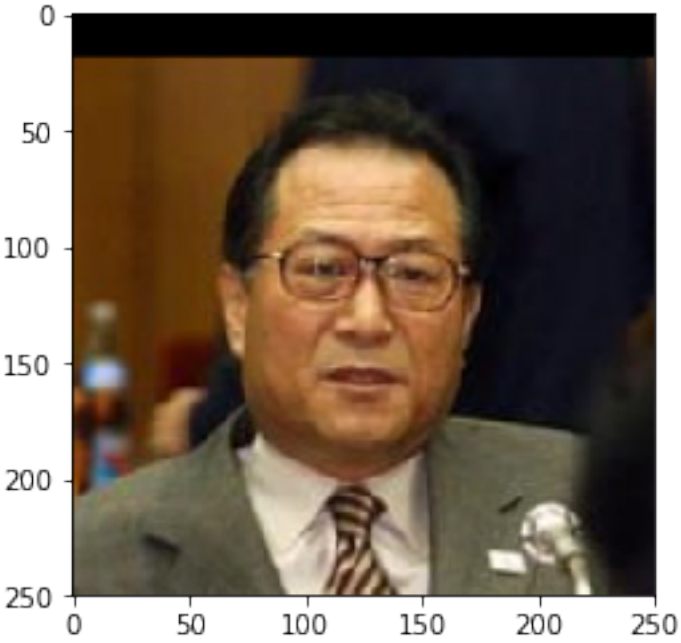

```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



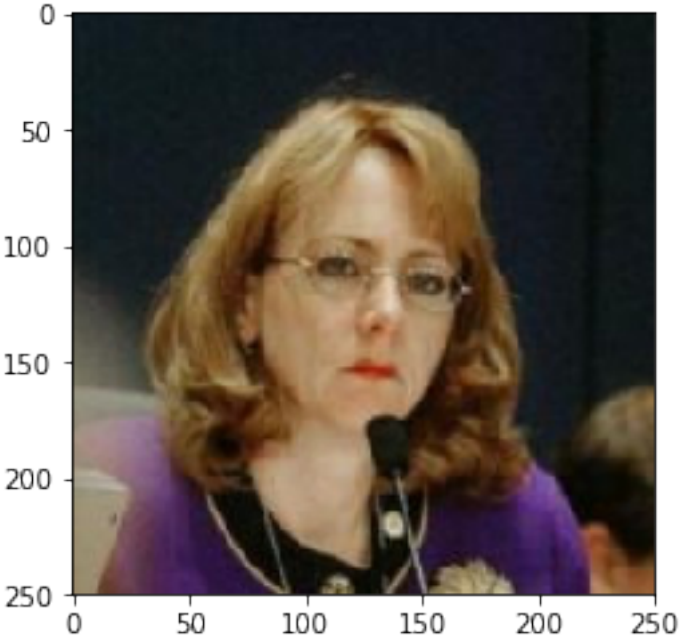
```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Irish wolfhound breed
```



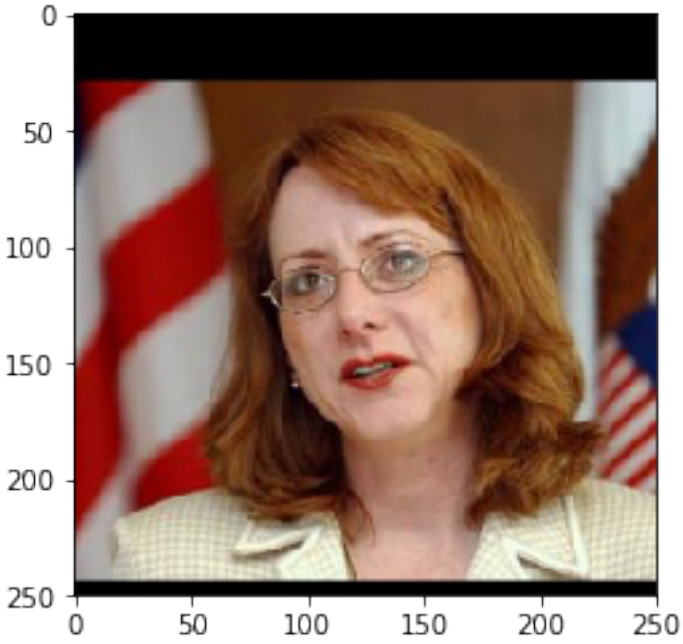

```
tensor([ 513], device='cuda:0')
Hello, human
You resumble a Cairn terrier breed
```



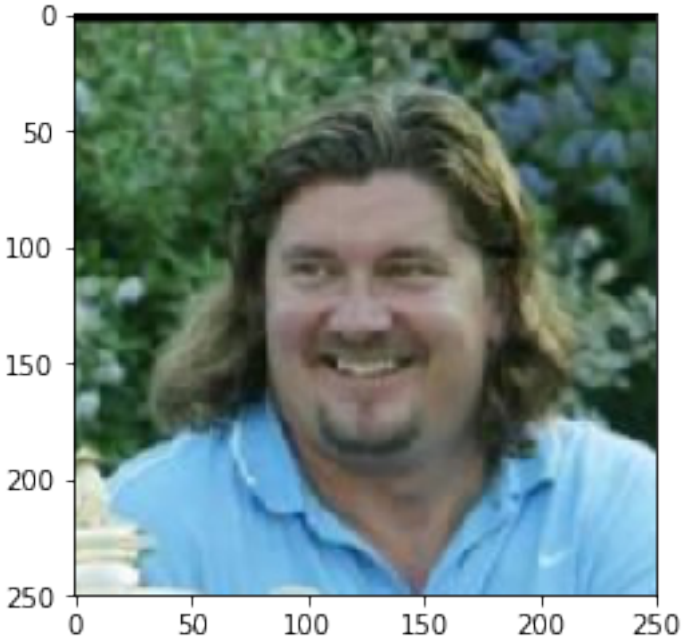

```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Basenji breed
```



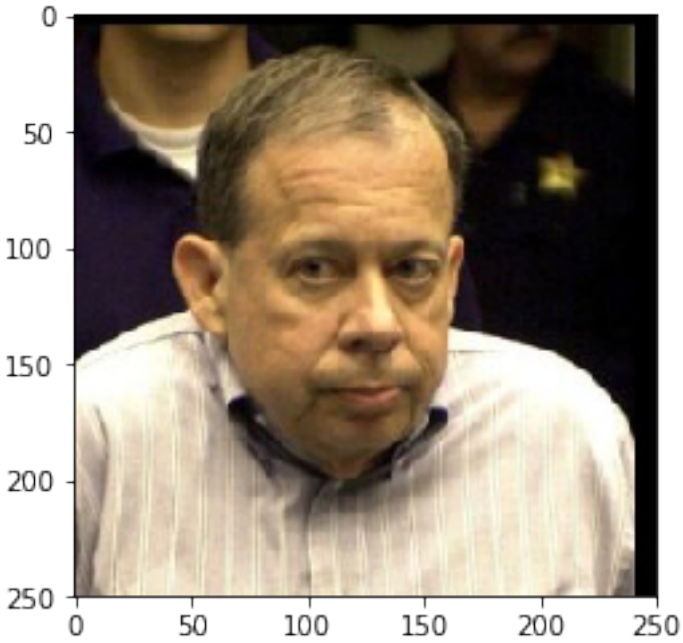
```
tensor([ 819], device='cuda:0')
Hello, human
You resumble a Bichon frise breed
```



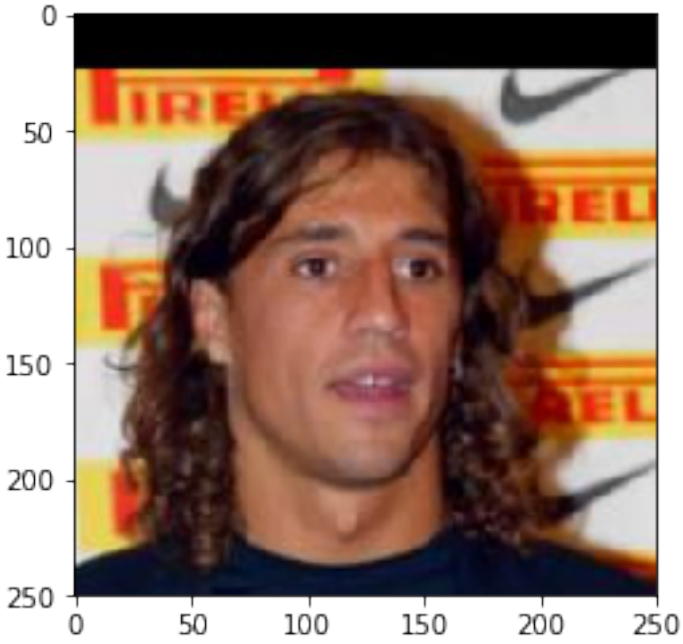

```
tensor([ 400], device='cuda:0')
Hello, human
You resumble a Cavalier king charles spaniel breed
```



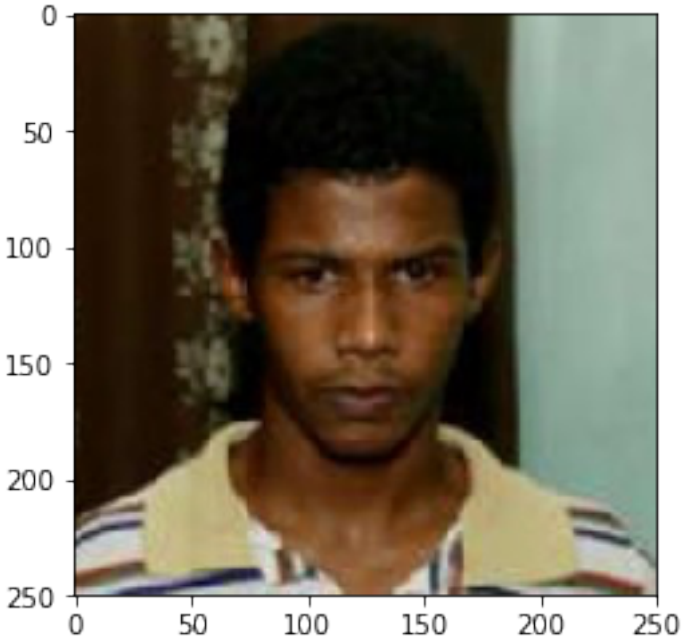

```
tensor([ 410], device='cuda:0')
Hello, human
You resumble a American staffordshire terrier breed
```



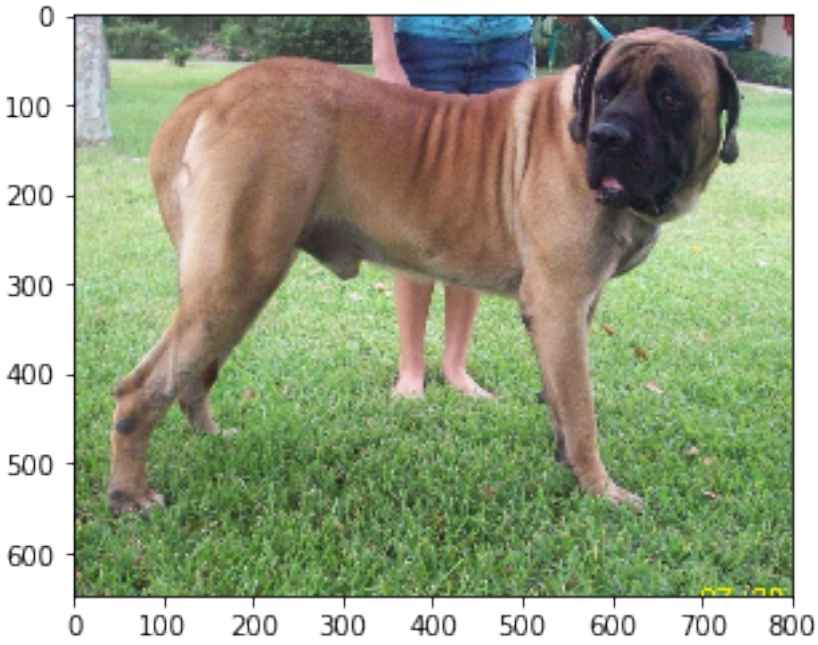
```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Bullmastiff breed
```



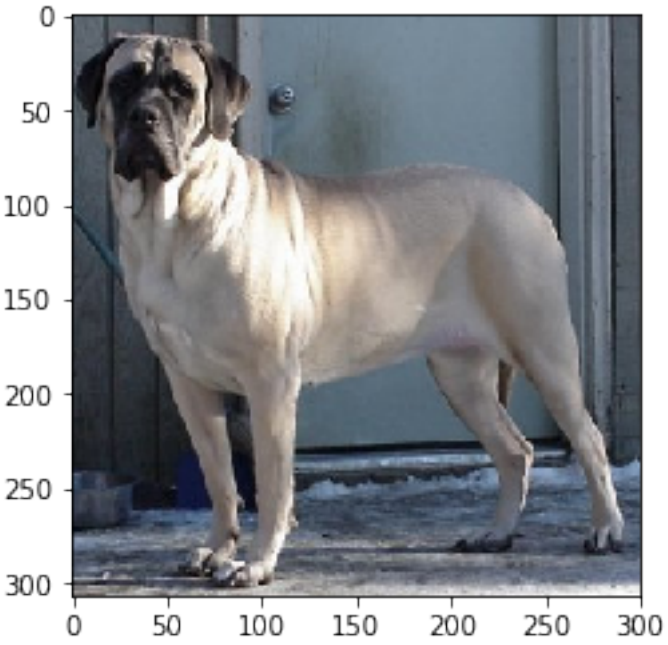

```
tensor([ 610], device='cuda:0')
Hello, human
You resumble a Dachshund breed
```



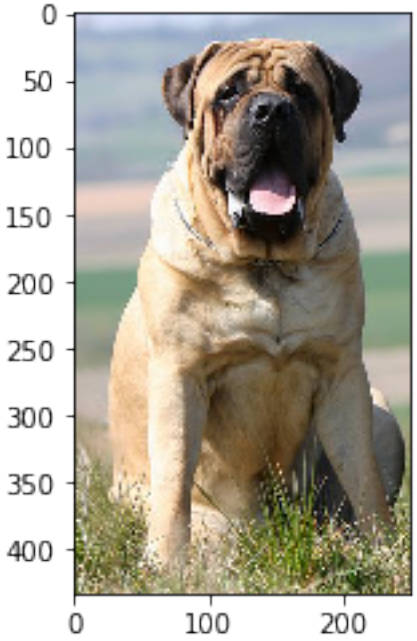

```
tensor([ 862], device='cuda:0')
Hello, human
You resumble a American staffordshire terrier breed
```



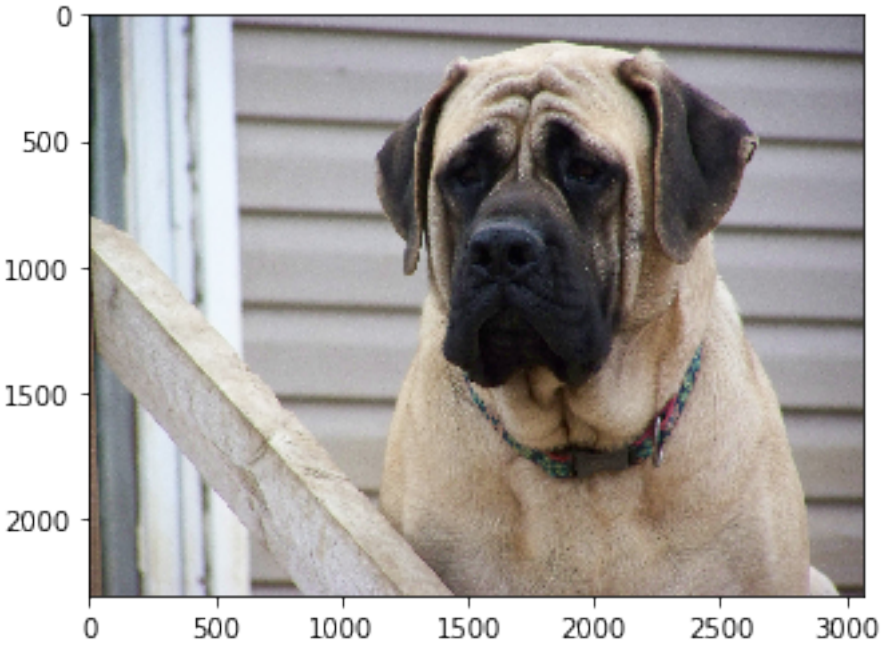
```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



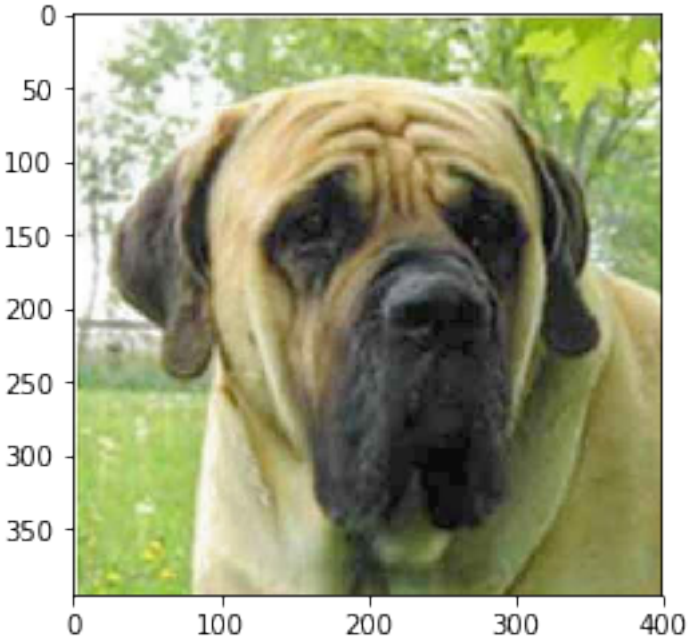

```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



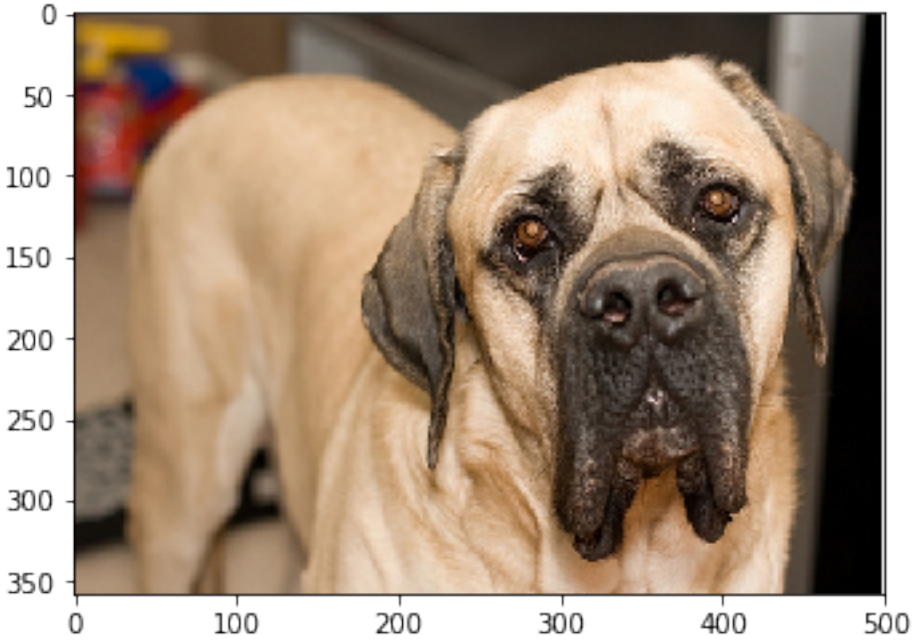
```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



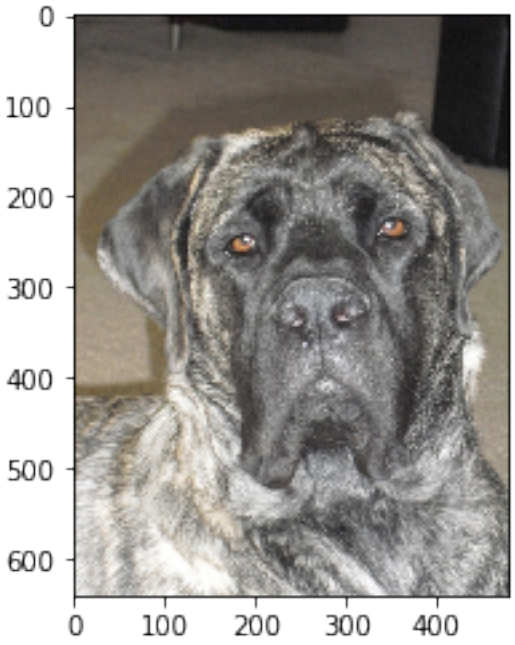

```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```




```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```

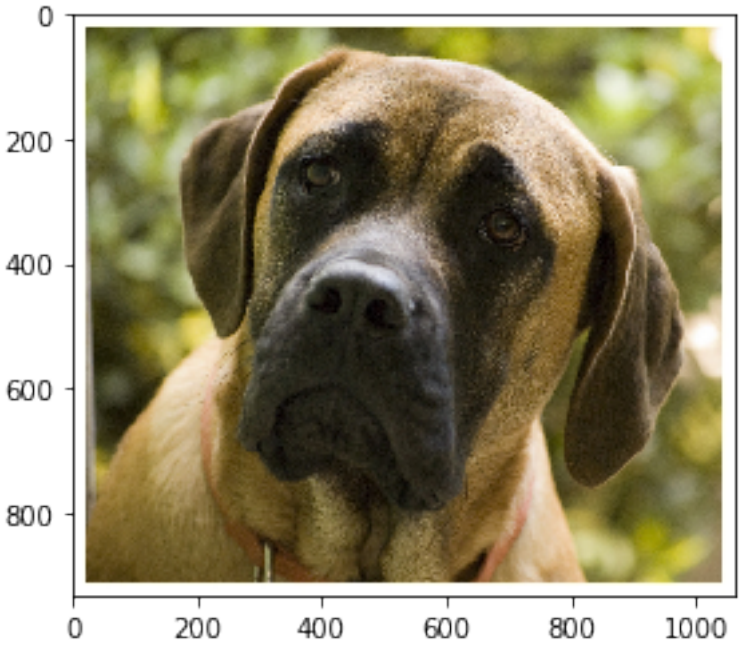


```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```

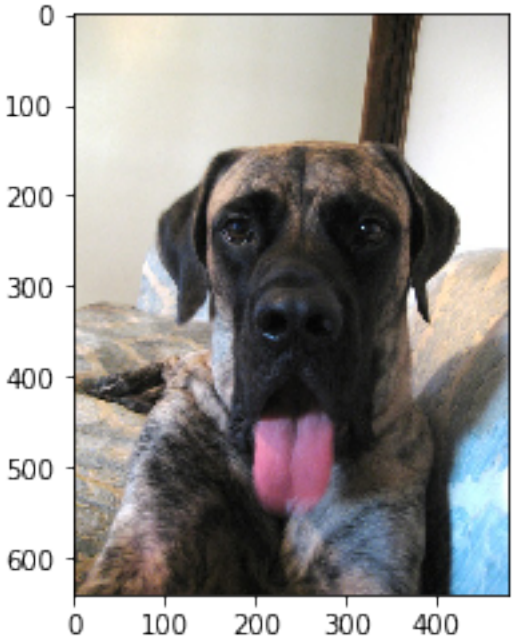



```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```

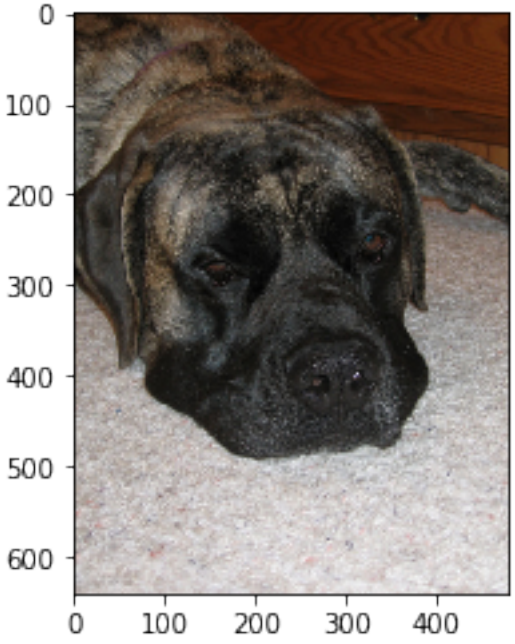
```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



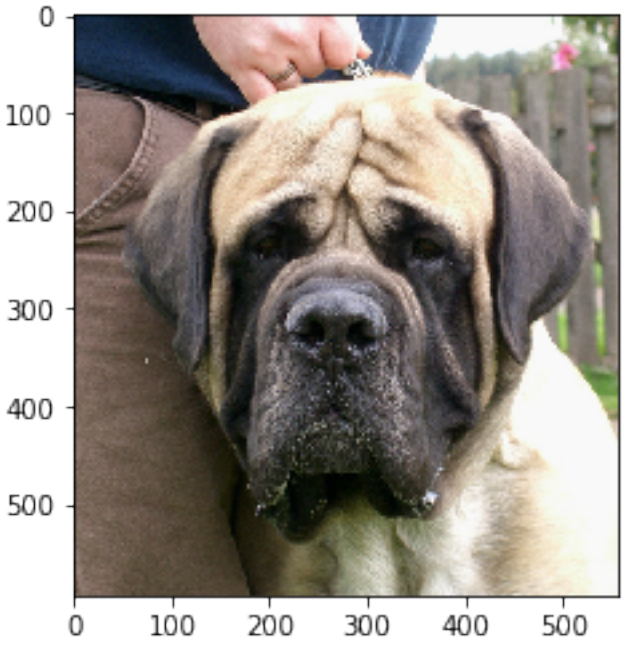

```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```



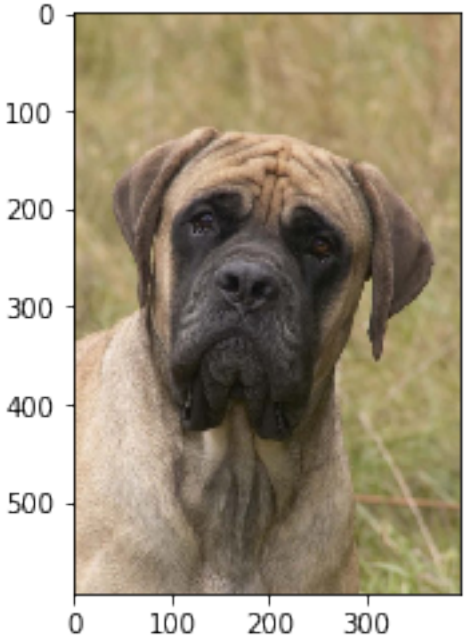

```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```



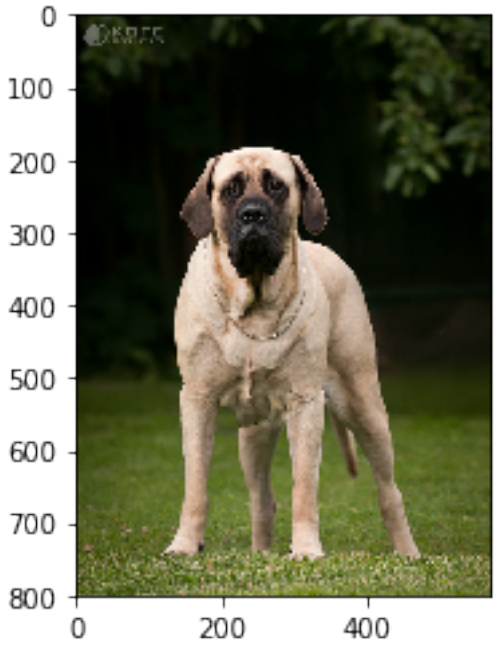
```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```



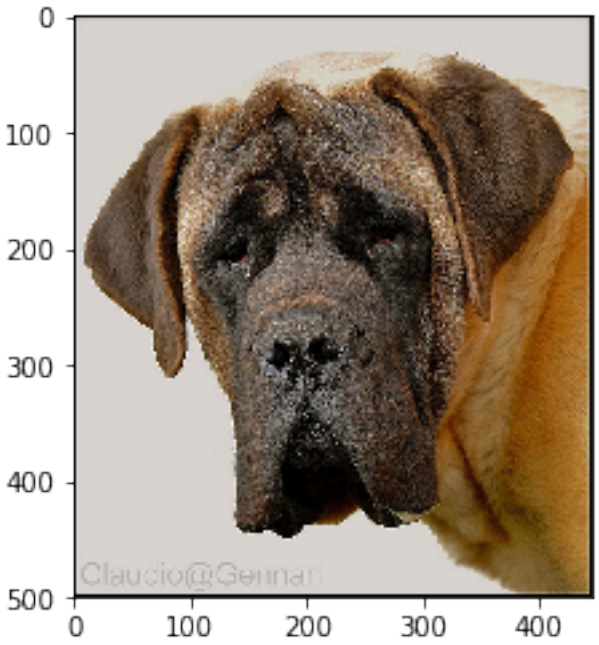

```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```



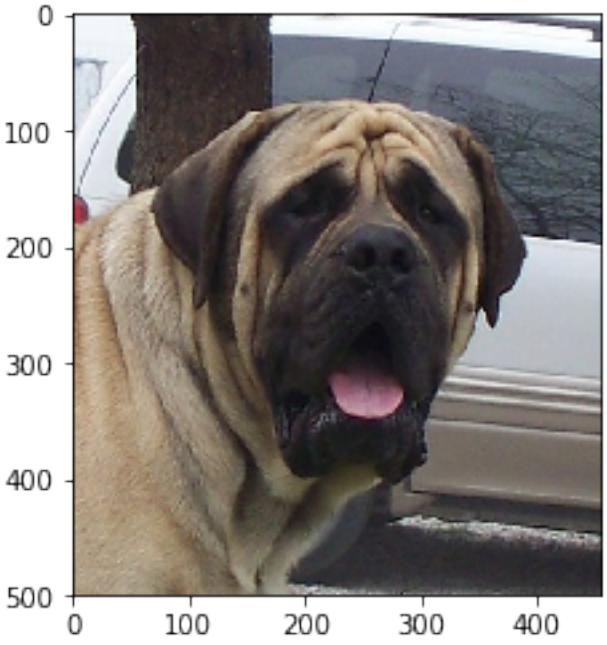
```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



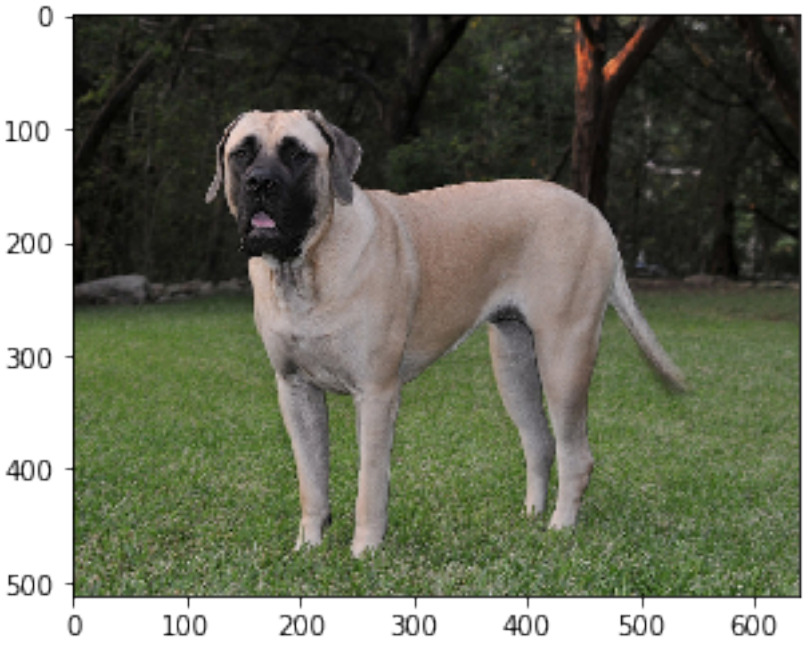

```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



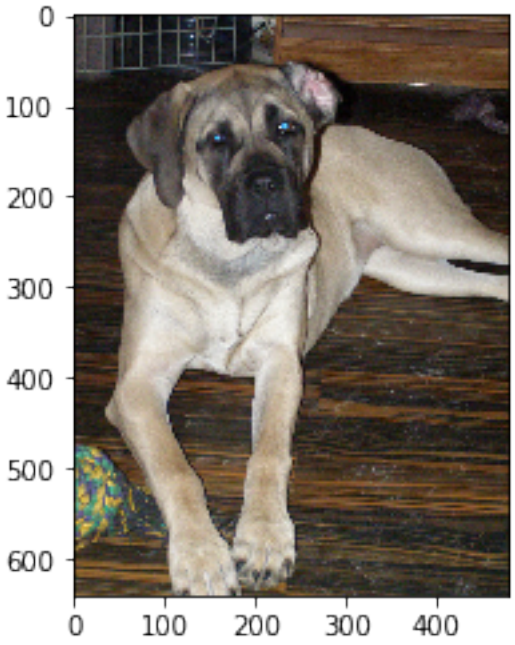

```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```

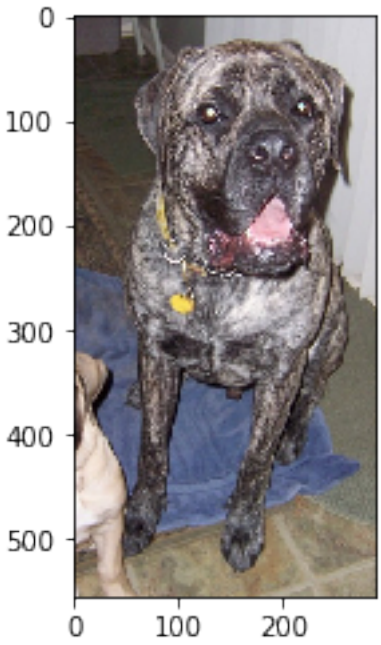



```
tensor([ 246], device='cuda:0')
Dog with breed Mastiff detected
```



```
tensor([ 243], device='cuda:0')
Dog with breed Mastiff detected
```


```
tensor([ 243], device='cuda:0')
Dog with breed Bullmastiff detected
```



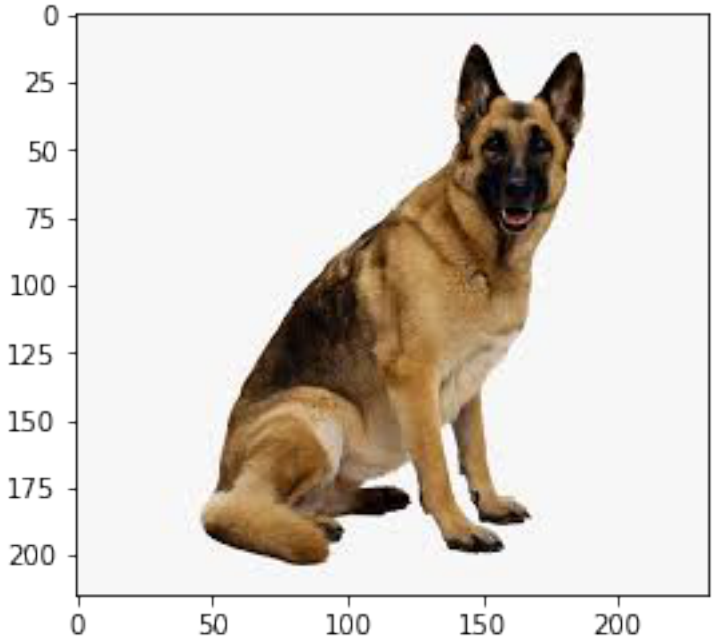

```
tensor([ 243], device='cuda:0')
Dog with breed German shorthaired pointer detected
```

In [38]:

In [44]:
```
# images taken from the internet
celfiles = np.array(glob("./images_cel/*.*"))
for file in np.hstack(celfiles):
    run_app(file)
```


```
tensor([ 234], device='cuda:0')
Dog with breed Beauceron detected
```

```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Cairn terrier breed
```

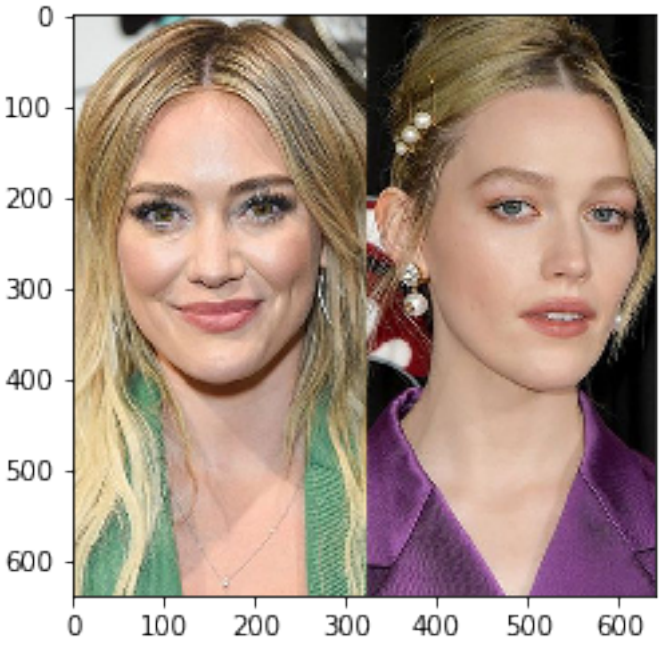



```
tensor([ 225], device='cuda:0')
Dog with breed Belgian malinois detected
```


```
tensor([ 608], device='cuda:0')
Hello, human
You resumble a Basenji breed
```

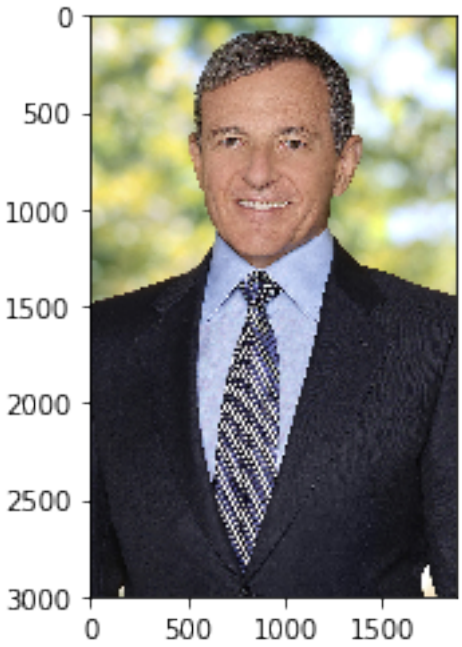

```
tensor([ 823], device='cuda:0')
Hello, human
You resumble a Basenji breed
```


```
tensor([ 423], device='cuda:0')
Hello, human
You resumble a Dachshund breed
```



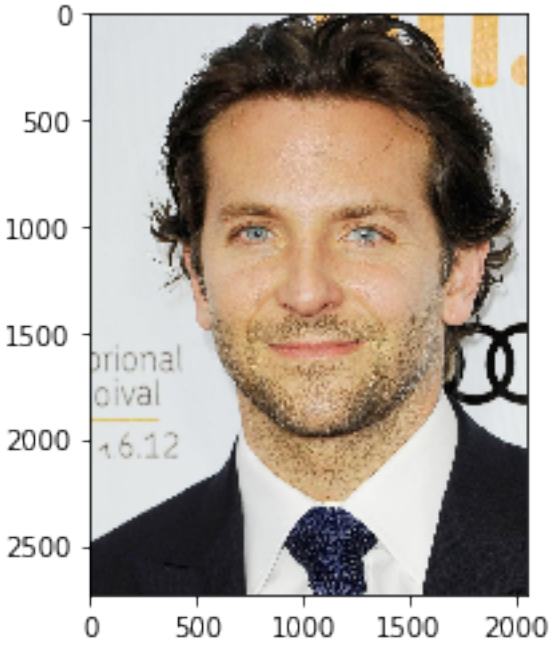
```
tensor([ 824], device='cuda:0')
Hello, human
You resumble a Cavalier king charles spaniel breed
```


```
tensor([ 207], device='cuda:0')
Dog with breed Golden retriever detected
```

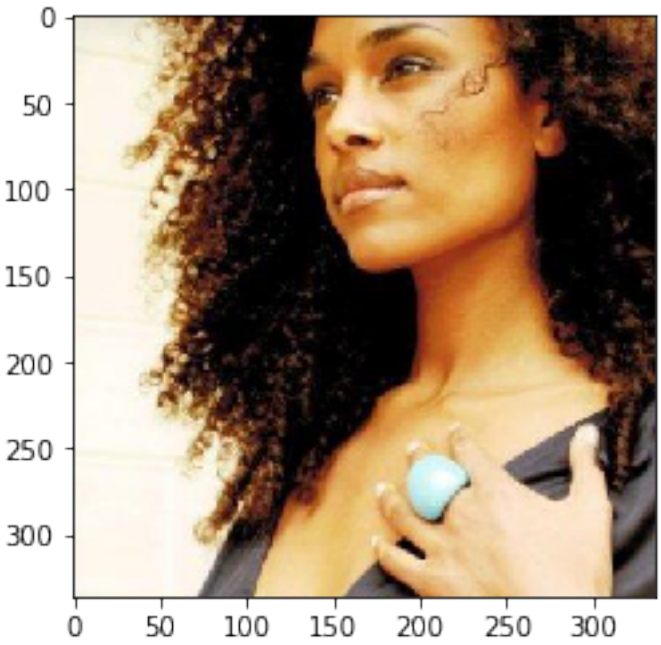



```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Cairn terrier breed
```

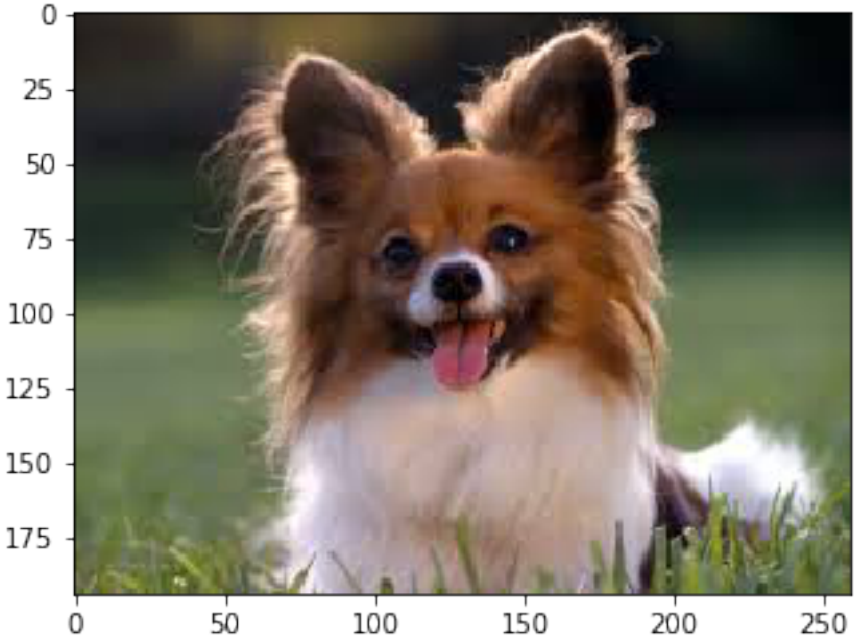
```
tensor([ 167], device='cuda:0')
Dog with breed Beagle detected
```




```
tensor([ 457], device='cuda:0')
Hello, human
You resumble a Cairn terrier breed
```




```
tensor([ 903], device='cuda:0')
Error: no face or dog detected
```

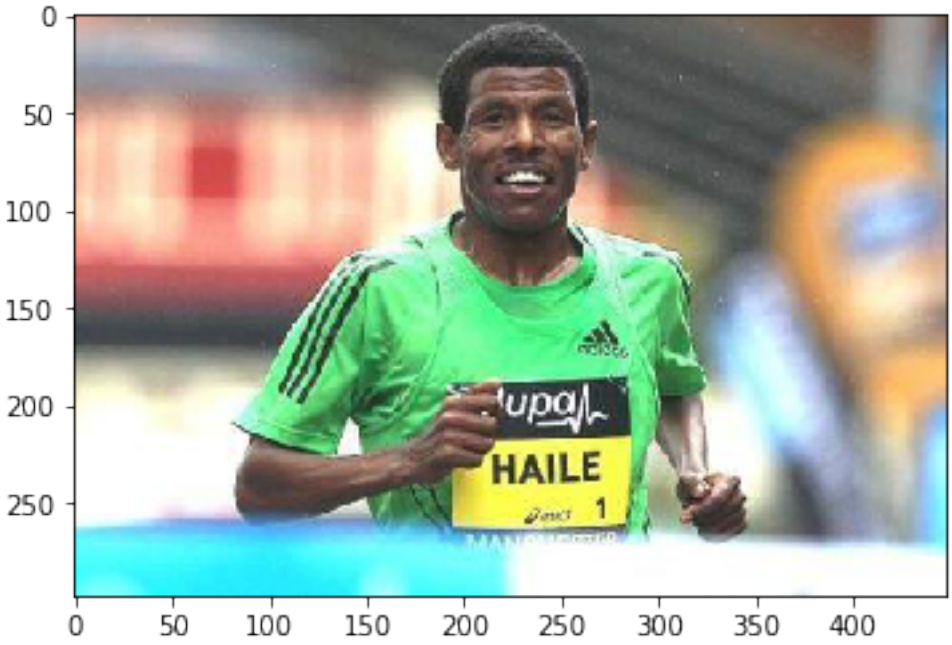


```
tensor([ 157], device='cuda:0')
Dog with breed Papillon detected
```


```
tensor([ 629], device='cuda:0')
Hello, human
You resumble a Boykin spaniel breed
```

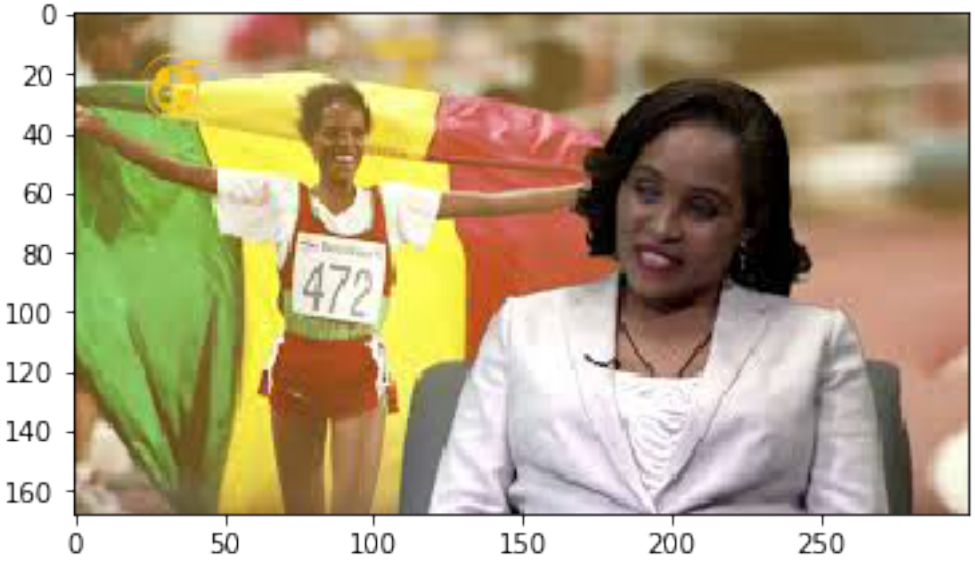

```
tensor([ 568], device='cuda:0')
Hello, human
```

```
You resumble a Irish water spaniel breed
```




```
tensor([ 722], device='cuda:0')
Hello, human
You resumble a Icelandic sheepdog breed
```

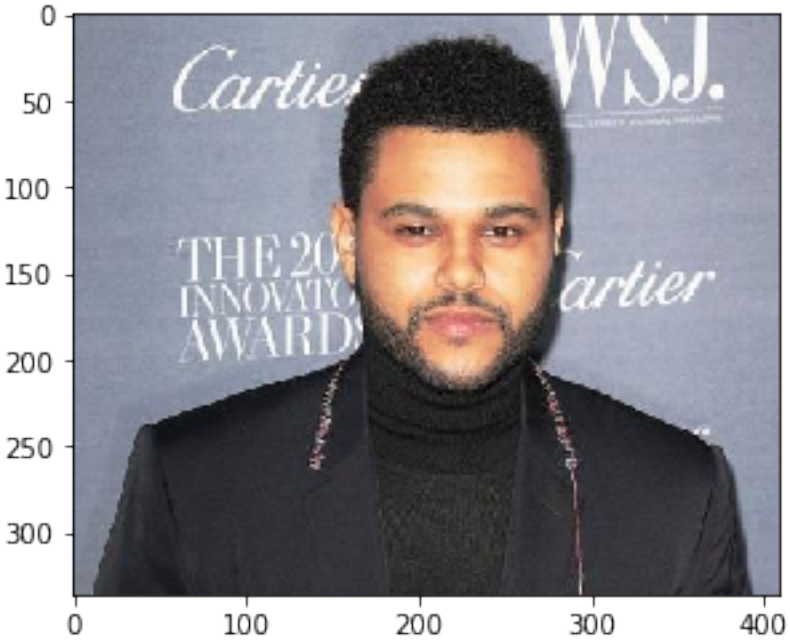
```
tensor([ 617], device='cuda:0')
Hello, human
You resumble a Clumber spaniel breed
```



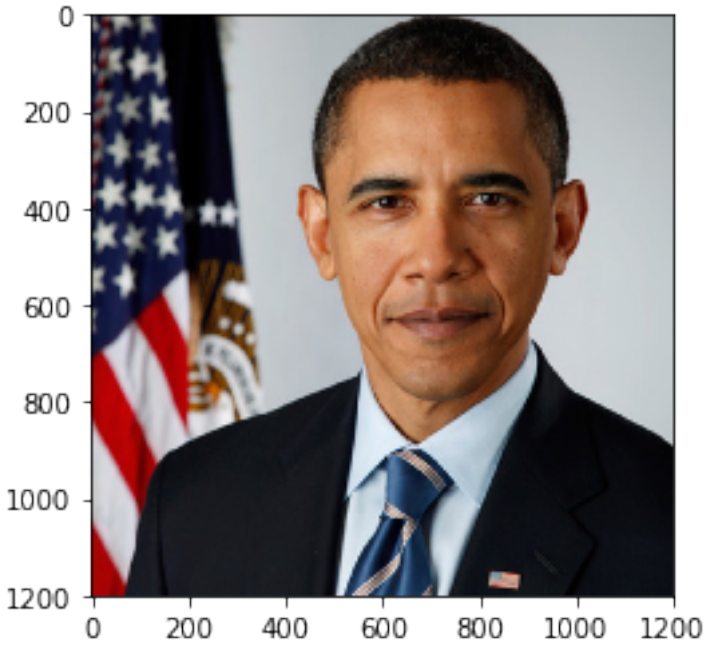

```
tensor([ 862], device='cuda:0')
Hello, human
You resumble a Dalmatian breed
```

```
tensor([ 254], device='cuda:0')
Dog with breed Boston terrier detected
```


```
tensor([ 834], device='cuda:0')
Hello, human
You resumble a Basenji breed
```




```
tensor([ 610], device='cuda:0')
Hello, human
You resumble a Belgian sheepdog breed
```



```
tensor([ 906], device='cuda:0')
Hello, human
You resumble a Cairn terrier breed
```

```
In [37]: celfiles
```

```
Out[37]: array([], dtype=float64)
```