dog_app

August 31, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the Jupyter Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to **File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location /dog_images.
- Download the human dataset. Unzip the folder and place it in the home directory, at location /lfw.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays human_files and dog_files.

```
In [1]: import numpy as np
        from glob import glob

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

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

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

Step 1: Detect Humans

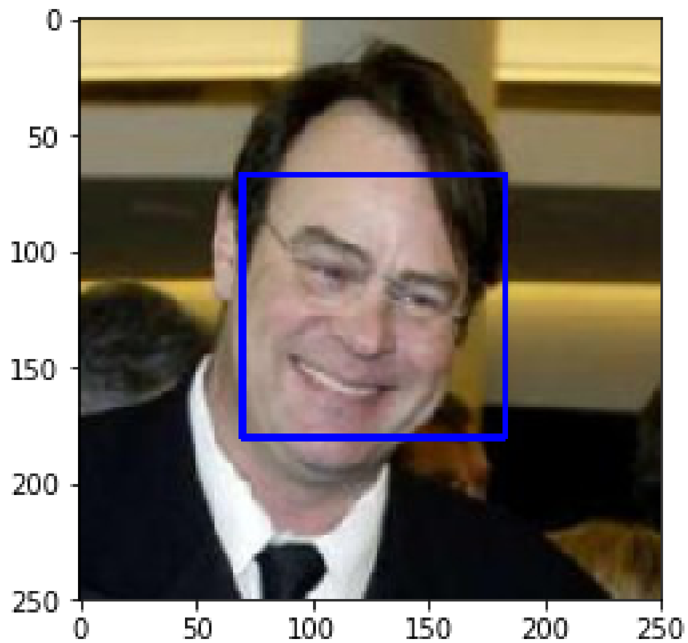
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the haarcascades directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # find faces in image
        faces = face_cascade.detectMultiScale(gray)

        # print number of faces detected in the image
        print('Number of faces detected:', len(faces))
```

```python
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell)

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        human_count = 0
        dog_count = 0

        for image in human_files_short:
            if face_detector(image) == True:
                human_count += 1

        for image in dog_files_short:
            if face_detector(image) == True:
                dog_count += 1

        print(f'Percentage of human images that detected human face: {human_count}%')
        print(f'Percentage of misclassified dog images that detected human faces: {dog_count}%')
```

```
Percentage of human images that detected human face: 98%
Percentage of misclassified dog images that detected human faces: 17%
```

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

In [5]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

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 [6]:
```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 95854442.60it/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 [7]:
```
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')


     # VGG-16 Takes 224x224 images as input, so we resize all of them and convert data t
    in_transform = transforms.Compose([
                        transforms.Resize(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 dimension
    image = in_transform(image)[:3,:,:].unsqueeze(0)

    return image


def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

    Returns:
        Index corresponding to VGG-16 model's prediction
    '''

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image



    image = load_image(img_path)
    if use_cuda:
        image = image.cuda()
    predict = VGG16(image)
    predict = predict.data.cpu().argmax()

    return predict # predicted class index
```

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

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        from tqdm import tqdm


        # Initiallizing:

        dog_percentage_in_human = 0
        dog_percentage_in_dog = 0


        for i in tqdm(range(len(human_files_short))):
            dog_percentage_in_human += int (dog_detector(human_files_short[i]))


        for i in tqdm(range(len(dog_files_short))):
            dog_percentage_in_dog += int (dog_detector(dog_files_short[i]))


        print (' Dog Percentage in Human dataset: {}% \n Dog Percentage in Dog Dataset: {}%'.for

100%|| 100/100 [00:03<00:00, 29.03it/s]
100%|| 100/100 [00:04<00:00, 20.75it/s]
```

```
Dog Percentage in Human dataset: 0%
Dog Percentage in Dog Dataset: 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 [10]: ### (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 [11]:
```
import os
from torchvision import datasets
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes


# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20


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


transform = { 'train'}

data_transforms = {
    'train' : transforms.Compose([transforms.Resize(224),
                                  transforms.CenterCrop(224),
                                  transforms.RandomHorizontalFlip(), # randomly flip an
                                  transforms.RandomRotation(10),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])]),

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

```
    'test' : transforms.Compose([transforms.Resize(224),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                std=[0.229, 0.224, 0.225])])}


image_datasets = {'train' : datasets.ImageFolder(root=train_dir,transform=data_transfor
                  'valid' : datasets.ImageFolder(root=valid_dir,transform=data_transfor
                  'test' : datasets.ImageFolder(root=test_dir,transform=data_transforms
```

```
loaders_scratch = {'train' : torch.utils.data.DataLoader(image_datasets['train'],batch_
                   'valid' : torch.utils.data.DataLoader(image_datasets['valid'],batch
                   'test' : torch.utils.data.DataLoader(image_datasets['test'],batch_s

dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid', 'test']}
class_names = image_datasets['train'].classes
n_classes = len(class_names)

print( len(class_names))
print( len(image_datasets['train']))

print( len(image_datasets['valid']))

print( len(image_datasets['test']))
```

```
133
6680
835
836
```

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: - The images were resized to 244X244 pixels using transforms.Resize(224) to avoid the bias, the long time, and to save memory size in case of the big images size. - Since the dataset size is small, the dataset was augmented by randomly flip using transforms.RandomHorizontalFlip() and images rotation using transforms.RandomRotation(10) functions.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [12]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN

         # convolutional layer (sees 32x32x3 image tensor)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
        # convolutional layer (sees 16x16x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
        # convolutional layer (sees 8x8x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # linear layer (64 * 28 * 28 -> 500)
        self.fc1 = nn.Linear(64 * 28 * 28, 500)
        # linear layer (500 -> 133)
        self.fc2 = nn.Linear(500, 133)
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.25)

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

         # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 64 * 28 * 28)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)

        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 [13]:
```
print(model_scratch)
```

```
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))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (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:

The input RGB image was cropped into 224x224 pixel and the depth is 3 (3 colors), we will have our input with the shape of 3x224x224. The desired no. of output is the same as no. of classes (here we have 133 classes in our images datasets).

Three convolutional layers were created using the relu activation function and one max pooling layer(2,2), the first layer takes (3,224,224) input and converted it into a depth of 16 layers, the filter used was 3x3 with stride of 1 and padding of 1 in order to not get rid of edges:

first convolutional layer (sees 32x32x3 image tensor). second convolutional layer (sees 16x16x16 tensor). third convolutional layer (sees 8x8x32 tensor). one pool layer (2,2) was used in order to reduce the size of the images to half. Then flatten image input using view function to reshape the tensor.

Then, two fully connected Linear layers with relu activation function were added, and dropout layers for the hidden layers with a percentage of 25% to avoid the bias: first linear layer (64 28 28 -> 500). second linear layer (500 -> 133), as the output classes is 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 [14]:
```
import torch.optim as optim

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

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

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 [15]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):
                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     ###
                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
                     # backward pass: compute gradient of the loss with respect to model paramet
                     loss.backward()
                     # perform a single optimization step (parameter update)
                     optimizer.step()
                     # update training loss
                     train_loss += loss.item()*data.size(0)
                     ###

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

                 for batch_idx, (data, target) in enumerate(loaders['valid']):
                     # move to GPU
```

```
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss

                ###
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += loss.item()*data.size(0)
                ###
            train_loss = train_loss/len(loaders['train'].dataset) ###
            valid_loss = valid_loss/len(loaders['valid'].dataset) ###


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

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

        # return trained model
        return model


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

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

```
Epoch: 1         Training Loss: 4.871163         Validation Loss: 4.818644
Validation loss decreased (inf --> 4.818644).  Saving model ...
Epoch: 2         Training Loss: 4.731521         Validation Loss: 4.654351
Validation loss decreased (4.818644 --> 4.654351).  Saving model ...
Epoch: 3         Training Loss: 4.574951         Validation Loss: 4.558323
Validation loss decreased (4.654351 --> 4.558323).  Saving model ...
```

```
Epoch: 4        Training Loss: 4.416839        Validation Loss: 4.387935
Validation loss decreased (4.558323 --> 4.387935).  Saving model ...
Epoch: 5        Training Loss: 4.279013        Validation Loss: 4.327182
Validation loss decreased (4.387935 --> 4.327182).  Saving model ...
Epoch: 6        Training Loss: 4.167768        Validation Loss: 4.263788
Validation loss decreased (4.327182 --> 4.263788).  Saving model ...
Epoch: 7        Training Loss: 4.062796        Validation Loss: 4.239401
Validation loss decreased (4.263788 --> 4.239401).  Saving model ...
Epoch: 8        Training Loss: 3.977595        Validation Loss: 4.154640
Validation loss decreased (4.239401 --> 4.154640).  Saving model ...
Epoch: 9        Training Loss: 3.888110        Validation Loss: 4.125212
Validation loss decreased (4.154640 --> 4.125212).  Saving model ...
Epoch: 10        Training Loss: 3.790417        Validation Loss: 4.112561
Validation loss decreased (4.125212 --> 4.112561).  Saving model ...
Epoch: 11        Training Loss: 3.685548        Validation Loss: 4.122174
Epoch: 12        Training Loss: 3.587138        Validation Loss: 4.107276
Validation loss decreased (4.112561 --> 4.107276).  Saving model ...
Epoch: 13        Training Loss: 3.478142        Validation Loss: 4.090480
Validation loss decreased (4.107276 --> 4.090480).  Saving model ...
Epoch: 14        Training Loss: 3.349853        Validation Loss: 4.070271
Validation loss decreased (4.090480 --> 4.070271).  Saving model ...
Epoch: 15        Training Loss: 3.250282        Validation Loss: 4.148986
Epoch: 16        Training Loss: 3.101890        Validation Loss: 4.118949
Epoch: 17        Training Loss: 2.964403        Validation Loss: 4.197226
Epoch: 18        Training Loss: 2.817492        Validation Loss: 4.159191
Epoch: 19        Training Loss: 2.660893        Validation Loss: 4.266165
Epoch: 20        Training Loss: 2.463330        Validation Loss: 4.367907
Epoch: 21        Training Loss: 2.342002        Validation Loss: 4.378192
Epoch: 22        Training Loss: 2.134953        Validation Loss: 4.409229
Epoch: 23        Training Loss: 1.948040        Validation Loss: 4.597934
Epoch: 24        Training Loss: 1.782230        Validation Loss: 4.726251
Epoch: 25        Training Loss: 1.640498        Validation Loss: 4.645061
```

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

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
```

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

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

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

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

```
Test Loss: 4.041086


Test Accuracy: 11% (98/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

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

```
In [17]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch
```

```
In [18]: print(VGG16)
         print(VGG16.classifier[6].in_features)
         print(VGG16.classifier[6].out_features)
```

```
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)
  )
)
4096
1000
```

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


         ## TODO: Specify model architecture

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

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

         n_inputs = model_transfer.classifier[6].in_features

         # add last linear layer (n_inputs -> 133 classes)
         # new layers automatically have requires_grad = True
         last_layer = nn.Linear(n_inputs, 133)

         model_transfer.classifier[6] = last_layer


         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:

Pre-trained modules are trained to extract patterns and features. The VGG16 models was choseng as a base model. The structure was checked then freezed during training for all "features" layers because if I changed it will take a long time. I changed the last linear layer output to be 133 classes to fit with our classification output. Then train the classifier to get a good result for our application.

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 [20]: 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 [21]:
```
# train the model
model_transfer = train(12, loaders_transfer, model_transfer, optimizer_transfer, criter

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

```
Epoch: 1         Training Loss: 3.782367         Validation Loss: 2.241430
Validation loss decreased (inf --> 2.241430).  Saving model ...
Epoch: 2         Training Loss: 1.748950         Validation Loss: 0.993243
Validation loss decreased (2.241430 --> 0.993243).  Saving model ...
Epoch: 3         Training Loss: 1.068154         Validation Loss: 0.709885
Validation loss decreased (0.993243 --> 0.709885).  Saving model ...
Epoch: 4         Training Loss: 0.832891         Validation Loss: 0.612402
Validation loss decreased (0.709885 --> 0.612402).  Saving model ...
Epoch: 5         Training Loss: 0.710197         Validation Loss: 0.534427
Validation loss decreased (0.612402 --> 0.534427).  Saving model ...
Epoch: 6         Training Loss: 0.624288         Validation Loss: 0.488456
Validation loss decreased (0.534427 --> 0.488456).  Saving model ...
Epoch: 7         Training Loss: 0.583492         Validation Loss: 0.466296
Validation loss decreased (0.488456 --> 0.466296).  Saving model ...
Epoch: 8         Training Loss: 0.522847         Validation Loss: 0.452870
Validation loss decreased (0.466296 --> 0.452870).  Saving model ...
Epoch: 9         Training Loss: 0.499362         Validation Loss: 0.440477
Validation loss decreased (0.452870 --> 0.440477).  Saving model ...
Epoch: 10          Training Loss: 0.460209         Validation Loss: 0.430613
Validation loss decreased (0.440477 --> 0.430613).  Saving model ...
Epoch: 11          Training Loss: 0.455046         Validation Loss: 0.416920
Validation loss decreased (0.430613 --> 0.416920).  Saving model ...
Epoch: 12          Training Loss: 0.419116         Validation Loss: 0.407362
Validation loss decreased (0.416920 --> 0.407362).  Saving model ...
```

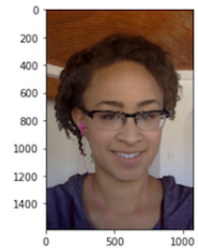
1.1.16 (IMPLEMENTATION) Test the Model

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 [22]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.447678


Test Accuracy: 86% (721/836)
```

hello, human!



You look like a ...
Chinese_shar-pei

Sample Human Output

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 [23]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in image_datasets['train'].classes]


def predict_breed_transfer(img_path):
    # load the image and return the predicted breed


    image = load_image(img_path)
    if use_cuda:
        image = image.cuda()
    predict = model_transfer(image)
    predict = predict.data.cpu().argmax()

    return class_names[predict]
```

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

         def display_img(img_path):
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()



         def run_app(img_path):
             ## handle cases for a human face, dog, and neither

             predicted_breed = predict_breed_transfer (img_path)
             if dog_detector(img_path):
                 print ('\n\n hello, it is a dog ')
                 display_img(img_path)
                 return print ('the predicted breed is:', predicted_breed)


             elif face_detector(img_path):
                 print ('\n\n hello, human')
                 display_img(img_path)
                 return print (' You look like a ', predicted_breed)

             else:
                 display_img(img_path)
                 print ('\n\n error: sorry it is neither human, nor dog ')
             predict_breed_transfer
```

Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

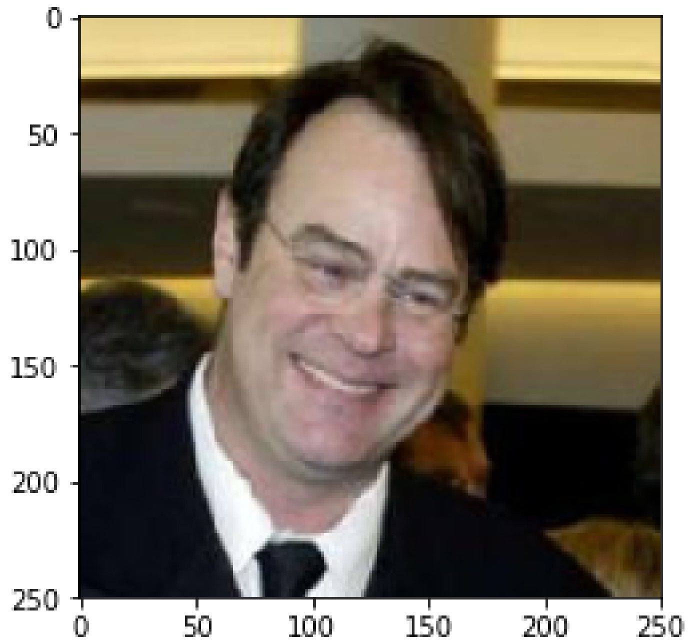
The output is really good.

Points for improvement:

- Train the model with a dataset has bigger number of images it will make an improvement.
- Consider more images augmentaion this will make an improvement.
- Train the model with more number of epochs this could make an improvement.

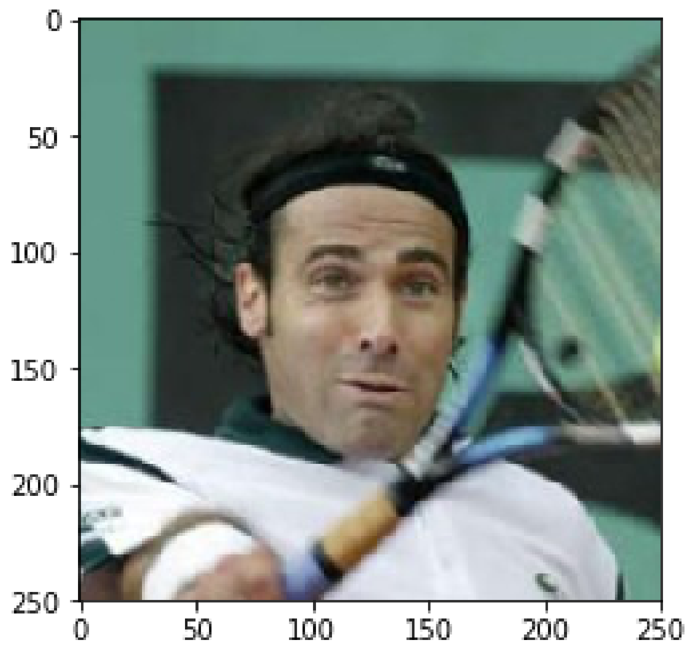
In [25]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

```
hello, human
```



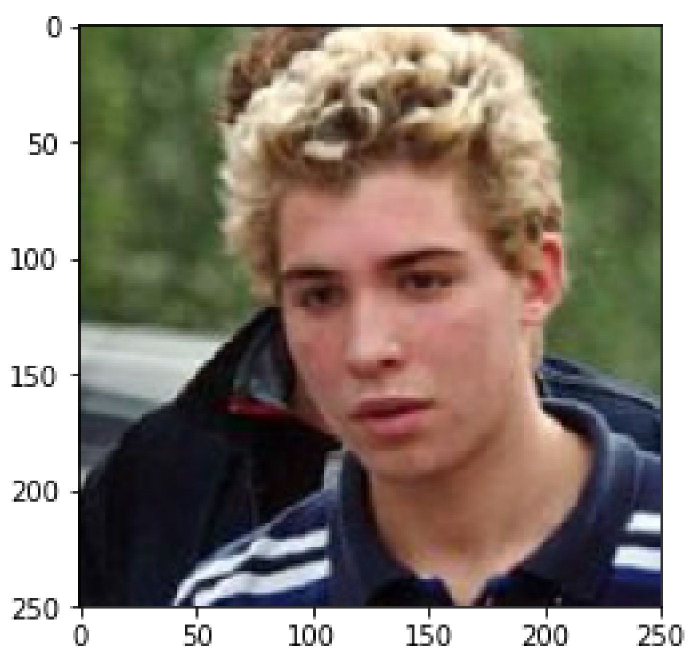

```
You look like a  Brittany
```

```
hello, human
```



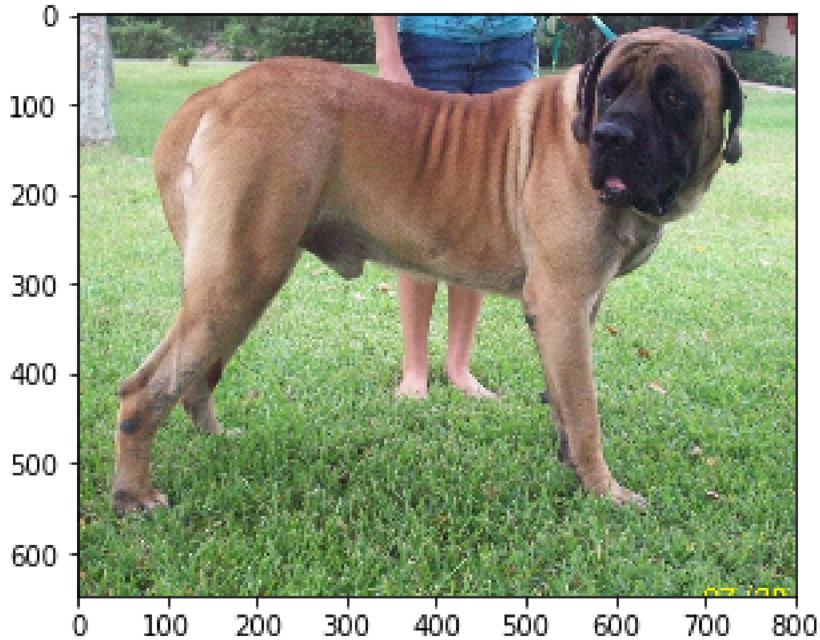
```
You look like a  Greyhound
```

```
hello, human
```



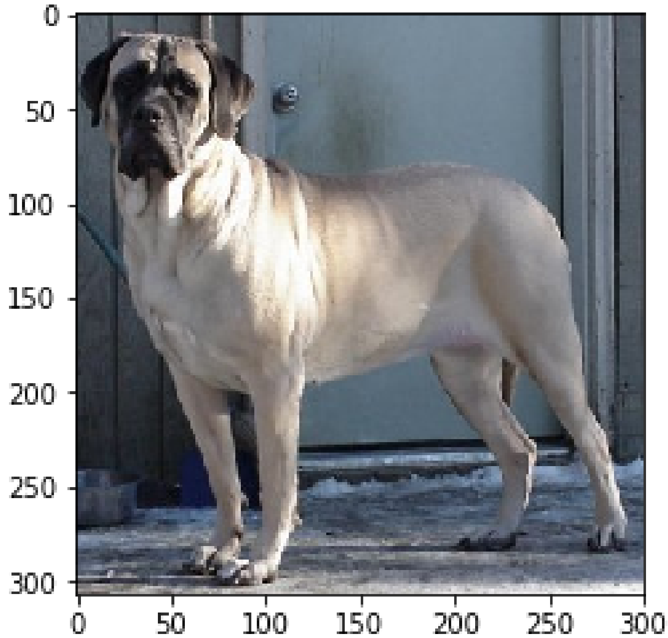
```
You look like a  Afghan hound
```

```
hello, it is a dog
```



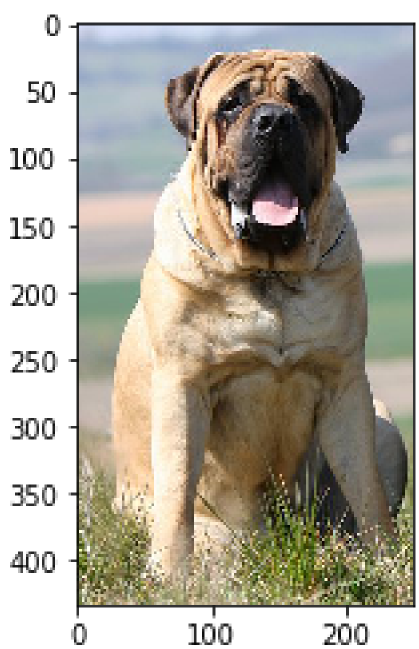

```
the predicted breed is: Bullmastiff
```

```
hello, it is a dog
```




```
the predicted breed is: Mastiff
```

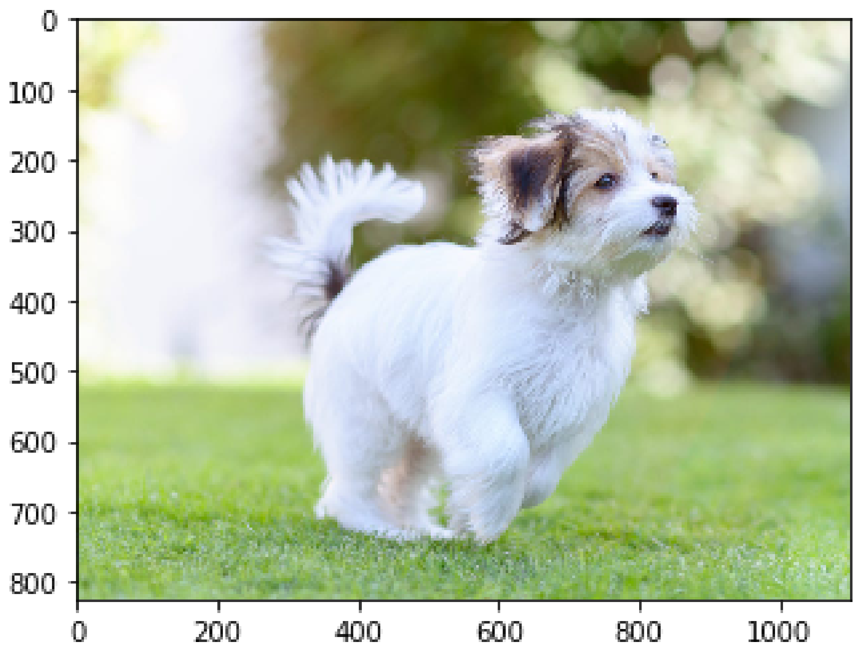
```
 hello, it is a dog
```



the predicted breed is: Mastiff

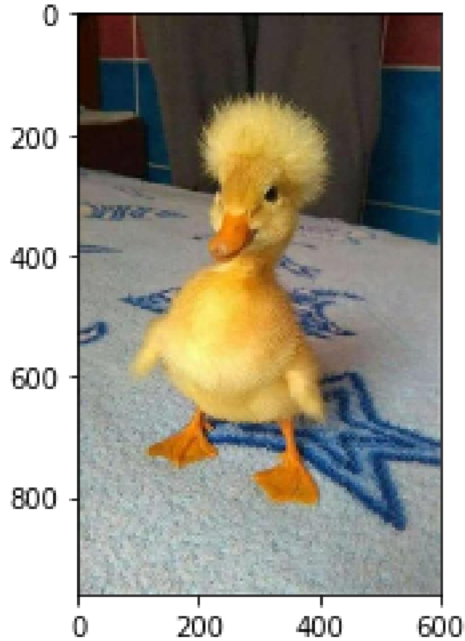
```
In [26]: #Testing the algorithm with at least six images from my computer
         my_specimen = glob("./test imgs/*")
         for file in np.hstack((my_specimen)):
             run_app(file)
```

```
hello, it is a dog
```




```
the predicted breed is: Chihuahua
```

```
hello, it is a dog
```



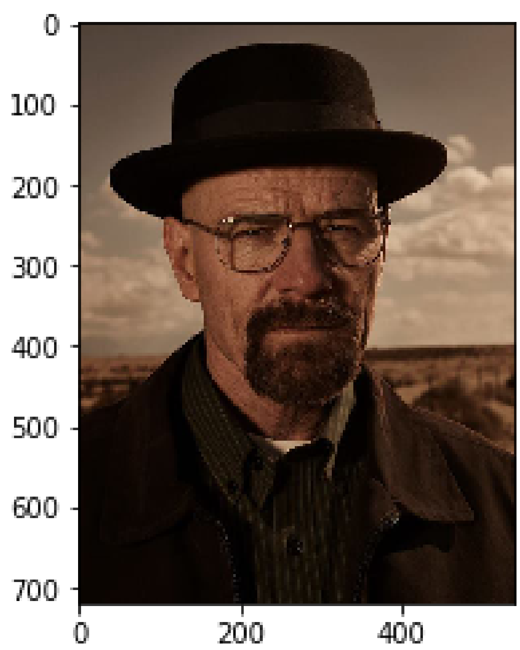
the predicted breed is: Pomeranian

```
error: sorry it is neither human, nor dog

hello, it is a dog
```

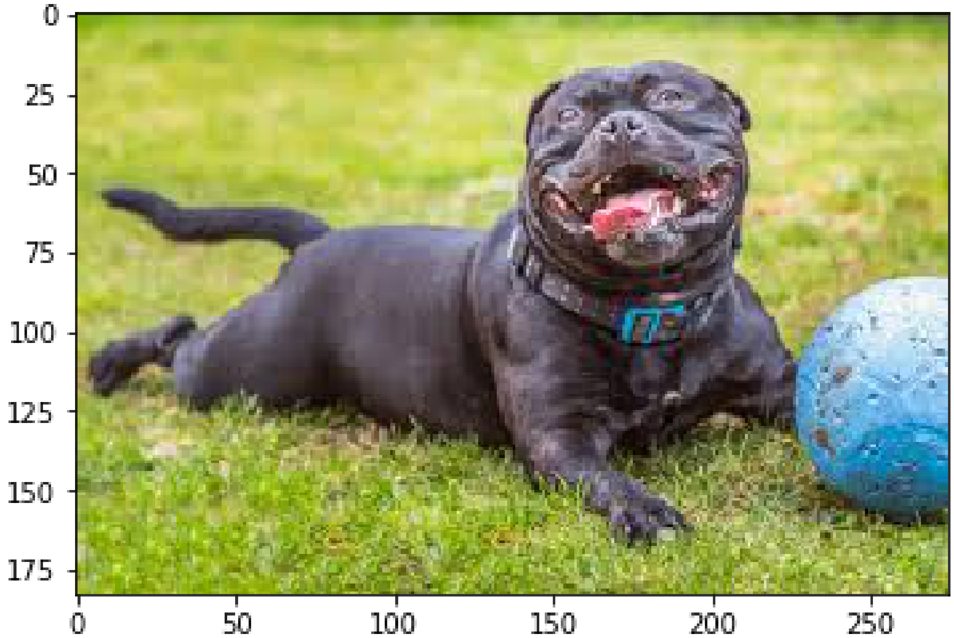

```
the predicted breed is: Golden retriever

hello, human
```

You look like a Greyhound

hello, human



```
You look like a  Nova scotia duck tolling retriever


hello, it is a dog
```




```
the predicted breed is: Cane corso
```

```
error: sorry it is neither human, nor dog
```