dog_app

May 9, 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 [16]: 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

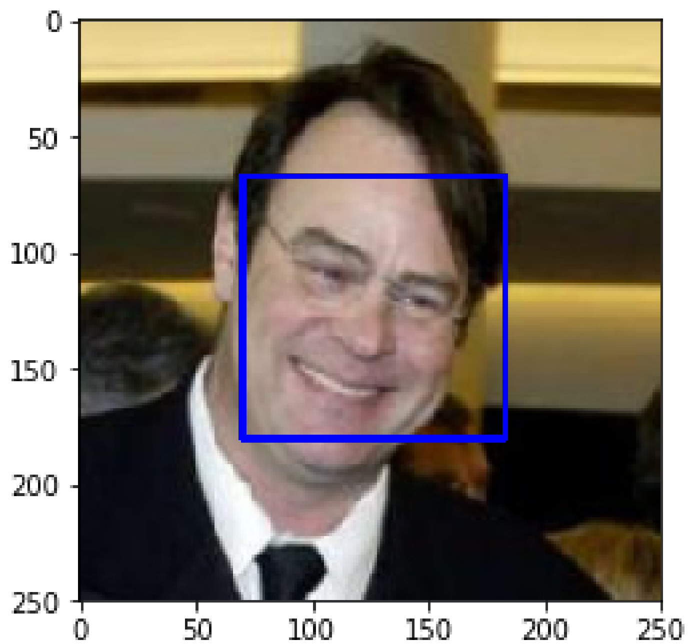
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [17]: import cv2
         import matplotlib.pyplot as plt
         %matplotlib inline

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [18]: # 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)
the accuracy of face_detector on humans=====> 98%
the accuracy of face_detector on dog=====> 17%

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

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

        #-#-# Do NOT modify the code above this line. #-#-#
        def acc_test(files):
         d = 0
         for i in files:
            if face_detector(i)==False:
                d+=1
         return 100-d
        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        print('the accuracy of face_detector on humans=====>',acc_test(human_files_short))
        print('the accuracy of face_detector on dog=====>',acc_test(dog_files_short))
```

```
the accuracy of face_detector on humans=====> 98
the accuracy of face_detector on dog=====> 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 [19]: 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()
         VGG16
```

```
Out[19]: VGG(
           (features): Sequential(
             (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (1): ReLU(inplace)
             (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (3): ReLU(inplace)
             (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (6): ReLU(inplace)
             (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (8): ReLU(inplace)
             (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (11): ReLU(inplace)
             (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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

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

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

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

```
In [5]: from PIL import Image
        import torchvision.transforms as transforms

        def VGG16_predict(img_path):
            '''
            Use pre-trained VGG-16 model to obtain index corresponding to
```

```
        predicted ImageNet class for image at specified path

        Args:
            img_path: path to an image

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

        ## TODO: Complete the function.
        ## Load and pre-process an image from the given img_path
        ## Return the *index* of the predicted class for that image
        img = Image.open(img_path)
        transform = transforms.Compose(
            [transforms.Resize((224, 224)),
             transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

        img = transform(img)
        if use_cuda:
                img = img.cuda()
        VGG16.eval()
        with torch.no_grad():
            out = VGG16(img.unsqueeze_(0))
            pred = out.cpu().data.numpy().argmax()
        VGG16.train()


        return pred # predicted class index
```

In [6]:
```
####### draft
```

In [7]:
```
###testing the functtion
VGG16_predict(human_files[8])
```

Out[7]: 906

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.

    return VGG16_predict(img_path) in np.arange(151, 269) # true/false
```

```
In [9]: ##### testing the function
        dog_detector(human_files[5])
```

```
Out[9]: 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:
the accuracy of dog_detector on humans=====> 0%
the accuracy of dog_detector on dog=====> 100%

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_files_short = human_files[:100]
         dog_files_short = dog_files[:100]
         def acc_test2(files):
          d = 0
          for i in files:
             if dog_detector(i)==False:
                 d+=1
          return 100-d
         print('the accuracy of dog_detector on humans=====>',acc_test2(human_files_short))
         print('the accuracy of dog_detector on dog=====>',acc_test2(dog_files_short))
```

```
the accuracy of dog_detector on humans=====> 0
the accuracy of dog_detector on dog=====> 100
```

```
In [13]: ####draft####
```

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 [14]: ### (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 [13]:
```
import os
import torch
from torchvision import datasets,transforms
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
train_dir = '/data/dog_images/train'
valid_dir = '/data/dog_images/valid'
test_dir = '/data/dog_images/test'
train_transforms = transforms.Compose([transforms.Resize(300),
                                       transforms.CenterCrop(256),
                                       transforms.RandomRotation(30),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                       ])
tv_transforms = transforms.Compose([transforms.Resize(300),
                                    transforms.CenterCrop(256),
                                  transforms.ToTensor(),
                                  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True)

test_data = datasets.ImageFolder(test_dir, transform=tv_transforms)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32, shuffle=True)

valid_data = datasets.ImageFolder(valid_dir, transform=tv_transforms)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=32, shuffle=True)

loaders_scratch = {'train':trainloader ,'test':testloader ,'valid':validloader}
```

In [14]:
```
##draft
```

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:

1)i resized the image and done some operation like rotation and random sizing to generalize our modle you you might notice that i made the pictuers size 300*300 and crop it from the center to be 256*256 i choose this tensor size becauce when i searched the internet for a good arctict ure i found the one i implemented below so i choose this tensor to fit in the fully connected layer

2)i choose to add random rotation and RandomHorizontalFlip to add some generalization for the model

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [15]:
```
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, 3, padding=1)
        self.conv2 = nn.Conv2d(64, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 512, 3, padding=1)
        # max pooling layer
        self.pool = nn.MaxPool2d(4, 4)

        self.fc1 = nn.Linear(512 * 4 * 4, 1500)
        # linear layer (500 -> 10)
        self.fc2 = nn.Linear(1500, 133)


        self.dropout = nn.Dropout(0.2)
    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 512 * 4 * 4)
        # add dropout layer
        x = self.dropout(F.relu(self.fc1(x)))
        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()
```

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

Answer: i started with 3 convlution layers and max pool with sride and kernel of 4 i applied relu and max pool to every conlution layer the i flatenned the out of the convlution layer to pass it to the fully connected layer we have two layers although one is enought but i think to take the input in two steps is better and find some pattern in the previous layer.

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 [16]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.04)
```

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 [14]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            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 parameters
            loss.backward()
        # perform a single optimization step (parameter update)
            optimizer.step()
        # update training loss
            train_loss += loss.item()*data.size(0)

        ######################
```

```python
            # 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
                output = model(data)
            # calculate the batch loss
                loss = criterion(output, target)
            # update average validation loss
                valid_loss += loss.item()*data.size(0)

            # print training/validation statistics
            train_loss = train_loss/len(trainloader.sampler)
            valid_loss = valid_loss/len(validloader.sampler)

            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 ...'.forma
             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(15, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1        Training Loss: 4.879693         Validation Loss: 4.862168
Validation loss decreased (inf --> 4.862168).  Saving model ...
Epoch: 2        Training Loss: 4.839317         Validation Loss: 4.777234
Validation loss decreased (4.862168 --> 4.777234).  Saving model ...
Epoch: 3        Training Loss: 4.679975         Validation Loss: 4.598844
Validation loss decreased (4.777234 --> 4.598844).  Saving model ...
```

```
Epoch: 4         Training Loss: 4.468937        Validation Loss: 4.454929
Validation loss decreased (4.598844 --> 4.454929).  Saving model ...
Epoch: 5         Training Loss: 4.339468        Validation Loss: 4.316132
Validation loss decreased (4.454929 --> 4.316132).  Saving model ...
Epoch: 6         Training Loss: 4.228187        Validation Loss: 4.226393
Validation loss decreased (4.316132 --> 4.226393).  Saving model ...
Epoch: 7         Training Loss: 4.141958        Validation Loss: 4.301202
Epoch: 8         Training Loss: 4.043466        Validation Loss: 4.125783
Validation loss decreased (4.226393 --> 4.125783).  Saving model ...
Epoch: 9         Training Loss: 3.956284        Validation Loss: 4.147149
Epoch: 10         Training Loss: 3.878084        Validation Loss: 4.056465
Validation loss decreased (4.125783 --> 4.056465).  Saving model ...
Epoch: 11         Training Loss: 3.777106        Validation Loss: 4.045132
Validation loss decreased (4.056465 --> 4.045132).  Saving model ...
Epoch: 12         Training Loss: 3.678199        Validation Loss: 3.956378
Validation loss decreased (4.045132 --> 3.956378).  Saving model ...
Epoch: 13         Training Loss: 3.593989        Validation Loss: 3.944545
Validation loss decreased (3.956378 --> 3.944545).  Saving model ...
Epoch: 14         Training Loss: 3.490375        Validation Loss: 3.788721
Validation loss decreased (3.944545 --> 3.788721).  Saving model ...
Epoch: 15         Training Loss: 3.382321        Validation Loss: 3.846833
```

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 [17]:
```python
import numpy as np
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
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: 3.765960


Test Accuracy: 12% (104/836)
```

2 ---

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.

2.0.1 (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 [9]: ## TODO: Specify data loaders
        import torch
        import numpy as np
        import os
        from torchvision import datasets,transforms
        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
        train_dir = '/data/dog_images/train'
        valid_dir = '/data/dog_images/valid'
        test_dir = '/data/dog_images/test'
        train_transforms = transforms.Compose([transforms.Resize(300),
```

```
                                         transforms.CenterCrop(224),
                                         transforms.RandomRotation(30),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
                                         ])
tv_transforms = transforms.Compose([transforms.Resize(300),
                                      transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))

train_data = datasets.ImageFolder(train_dir, transform=train_transforms)
trainloader = torch.utils.data.DataLoader(train_data, batch_size=20, shuffle=True)

test_data = datasets.ImageFolder(test_dir, transform=tv_transforms)
testloader = torch.utils.data.DataLoader(test_data, batch_size=20, shuffle=True)

valid_data = datasets.ImageFolder(valid_dir, transform=tv_transforms)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=20, shuffle=True)

loaders = {'train':trainloader ,'test':testloader ,'valid':validloader}
```

2.0.2 (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 [10]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)
         for param in model_transfer.features.parameters():
             param.requires_grad = False
         model_transfer.classifier[6] = nn.Linear(in_features=4096, out_features=133)

         for param in model_transfer.classifier.parameters():
             param.requires_grad = True
         # check if CUDA is available
         use_cuda = torch.cuda.is_available()



         if use_cuda:
             model_transfer = model_transfer.cuda()
         #print(model_transfer.classifier[6].out_features)
         model_transfer
```

```
Out[10]: 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=133, bias=True)
  )
)
```

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

Answer:

i turned of the grad for the convlution layer changed the last fully connected layer this arctiture is suitbale because it pretraind on image classfication including dogs.

2.0.3 (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 [11]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001, momentum=0.9)
```

2.0.4 (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 [4]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            # initialize tracker for minimum validation loss
            valid_loss_min = np.Inf

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

                ###################
                # train the model #
                ###################
                model.train()
                for batch_idx, (data, target) in enumerate(loaders['train']):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    optimizer.zero_grad()
                # 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 parameters
                    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
            output = model(data)
        # calculate the batch loss
            loss = criterion(output, target)
        # update average validation loss
            valid_loss += loss.item()*data.size(0)

        # print training/validation statistics
        train_loss = train_loss/len(trainloader.sampler)
        valid_loss = valid_loss/len(validloader.sampler)

        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 ...'.format
         valid_loss_min,
         valid_loss))
         torch.save(model.state_dict(), save_path)
         valid_loss_min = valid_loss
    # return trained model
    return model
```

```
model_transfer = train(5, loaders, model_transfer, optimizer, criterion, use_cuda, 'mode
# train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, criterion_transf

# 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: 2.359423 	Validation Loss: 0.919298
Validation loss decreased (inf --> 0.919298).  Saving model ...
```

```
Epoch: 2         Training Loss: 1.202030        Validation Loss: 0.823602
Validation loss decreased (0.919298 --> 0.823602).  Saving model ...
Epoch: 3         Training Loss: 1.005974        Validation Loss: 0.706248
Validation loss decreased (0.823602 --> 0.706248).  Saving model ...
Epoch: 4         Training Loss: 0.876427        Validation Loss: 0.685346
Validation loss decreased (0.706248 --> 0.685346).  Saving model ...
Epoch: 5         Training Loss: 0.772746        Validation Loss: 0.664473
Validation loss decreased (0.685346 --> 0.664473).  Saving model ...
```

2.0.5 (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 [10]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         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))
         ##########################################################################
         test(loaders, model_transfer, criterion, use_cuda)
```

```
Test Loss: 0.696334
```

```
Test Accuracy: 78% (657/836)
```

2.0.6 (IMPLEMENTATION) Predict Dog Breed with the Model

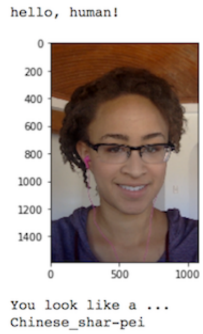
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [12]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         #model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         # 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 loaders['train'].dataset.classes]


         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path)


             transform = transforms.Compose([transforms.Resize(300),
                                                 transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)

             img = transform(img)
             if use_cuda:
                     img = img.cuda()
             model_transfer.eval()
             with torch.no_grad():
                 out = model_transfer(img.unsqueeze_(0))
                 pred = out.cpu().data.numpy().argmax()
             return class_names[pred]
         ############################################
         print(class_names)

['Affenpinscher', 'Afghan hound', 'Airedale terrier', 'Akita', 'Alaskan malamute', 'American esk
```

```
In [20]: predict_breed_transfer(dog_files[0])

Out[20]: 'Bullmastiff'
```

Step 5: Write your Algorithm

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



Sample Human Output

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

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

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

2.0.7 (IMPLEMENTATION) Write your Algorithm

```
In [21]: predict_breed_transfer(human_files[0])
```

```
Out[21]: 'Dachshund'
```

```
In [22]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         from PIL import ImageFile
         import torchvision.transforms as transforms
         def run_app(img_path):
             ## handle cases for a human face, dog, and neither

             img = Image.open(img_path)
             transform = transforms.Compose(
                 [transforms.Resize((224, 224)),
                  transforms.ToTensor(),
                  transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
             img = transform(img)
             if dog_detector(img_path) == True:
                 pred = predict_breed_transfer(img_path)
                 x = 'the breed of this dog is '
                 print(x+pred)
                 plt.imshow(Image.open(img_path))
                 plt.show()

             elif face_detector(img_path) == True:
                 pred = predict_breed_transfer(img_path)
```

```
            x = ''''hi human
            you look like a '''
            print(x + pred)
            plt.imshow(Image.open(img_path))
            plt.show()
        else:
            pred = 'neither human nor dog'
            x = ''
            print(pred + x)
            plt.imshow(Image.open(img_path))
            plt.show()
```

```
In [19]: #draft
```

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?

2.0.8 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

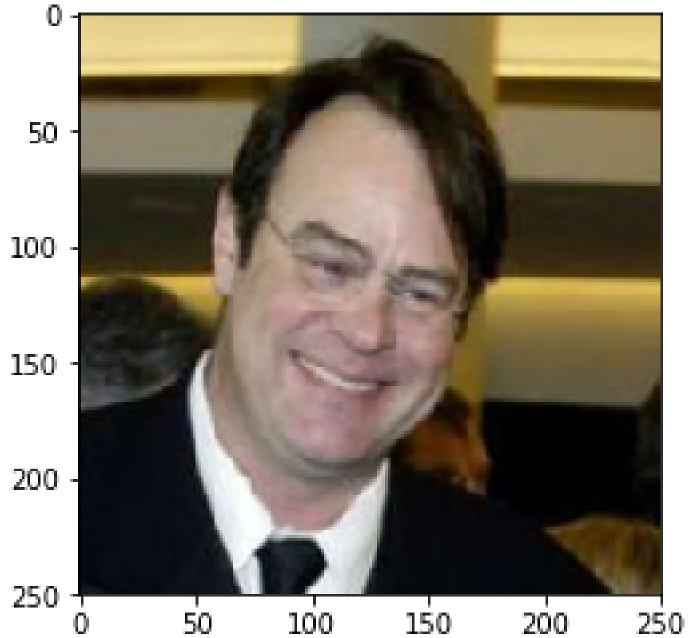
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

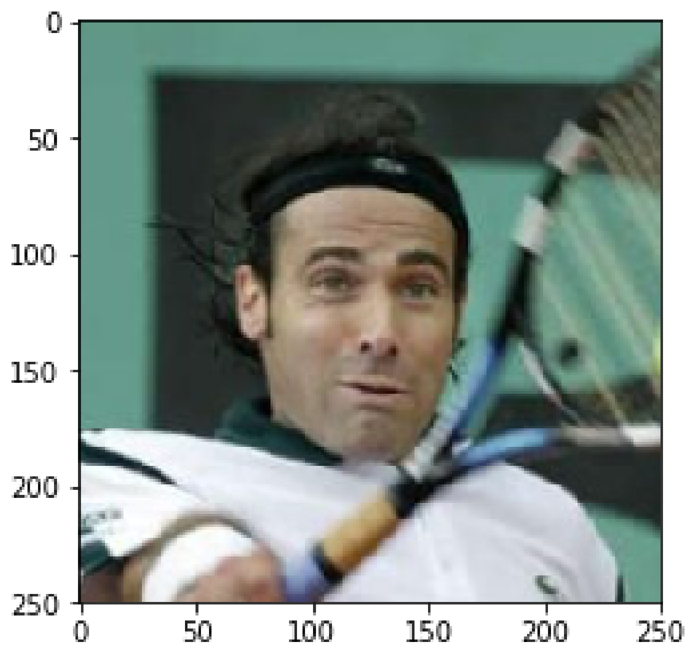
Answer:
the output is actually good it even get the breed of the pictures i uploaded right the model has done well for 5 epochs of training to make it better:-
1)train it for more epochs
2)add more randoumness in the data
3)add more data

```
In [23]: ## 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)
```

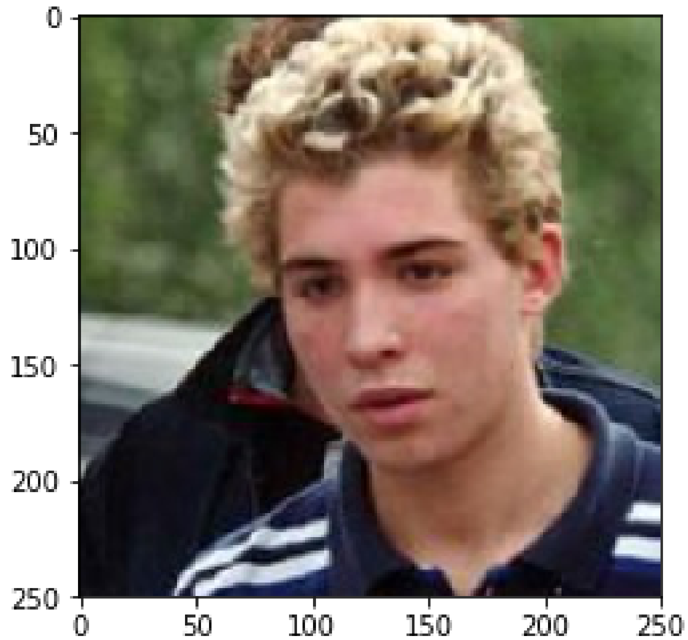
```
'hi human
        you look like a Dachshund
```



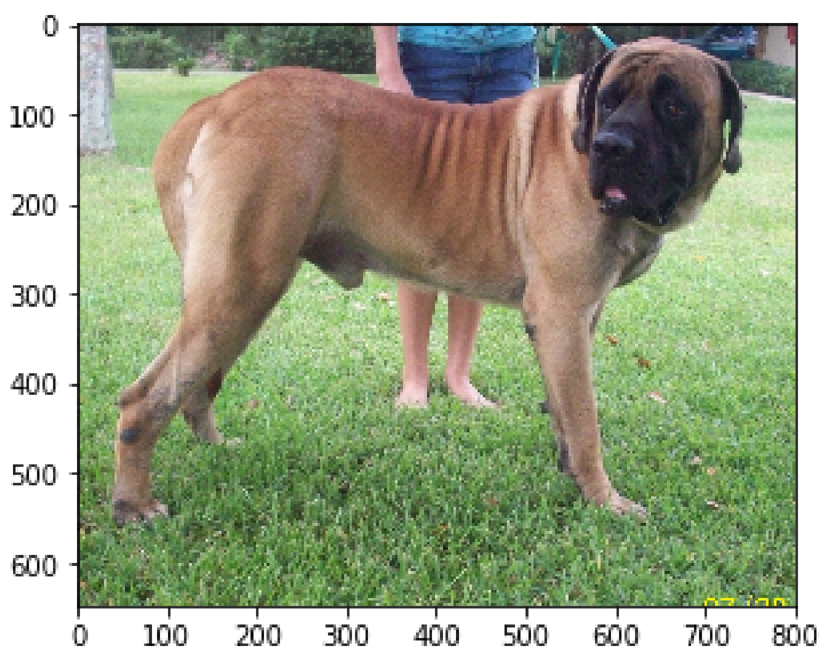
```
'hi human
        you look like a Dachshund
```



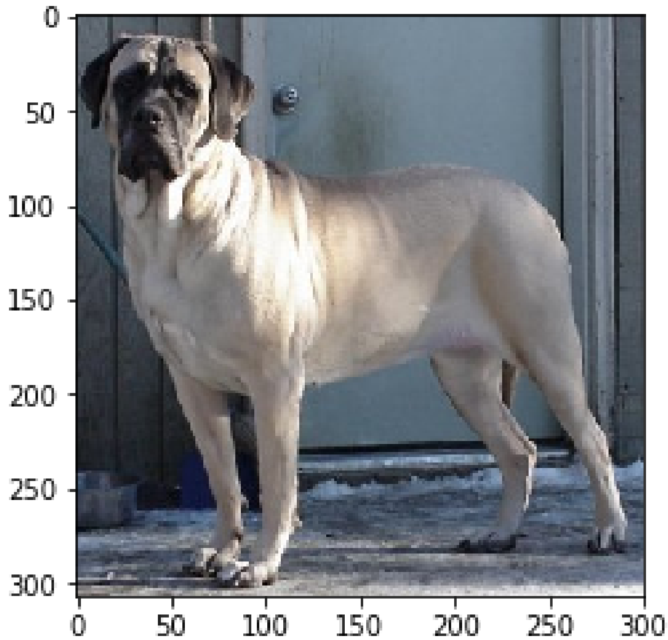
```
'hi human
        you look like a Chinese crested
```



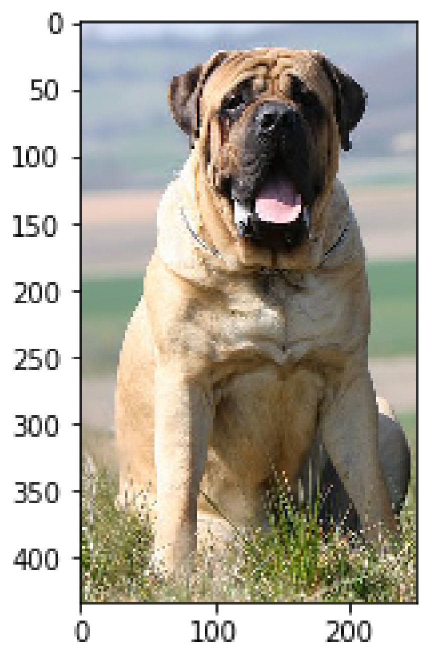

```
the breed of this dog is Bullmastiff
```



```
the breed of this dog is Mastiff
```

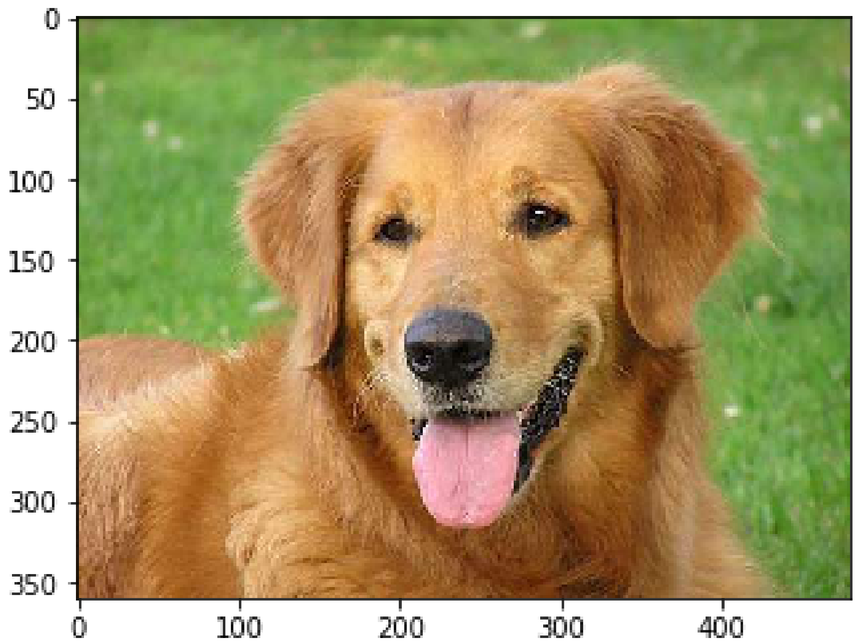



```
the breed of this dog is Mastiff
```

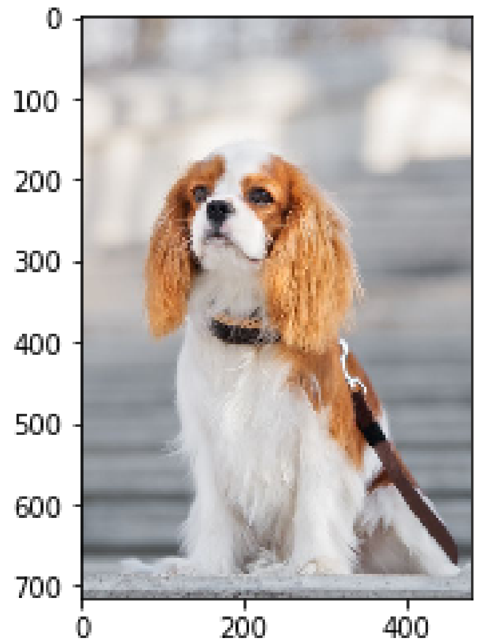


```
In [24]: import numpy as np
         from glob import glob

         # load filenames for human and dog images
         images = ["./img1.jpg","./img2.jpg","./img7.jpg","./img4.jpg","./img5.jpeg","./img6.jpe
         for img in images:
             imagesfile = np.array(glob(img))
             run_app(imagesfile[0])

         #plt.imshow(Image.open(imagesfile[0]))
         #plt.show()
```

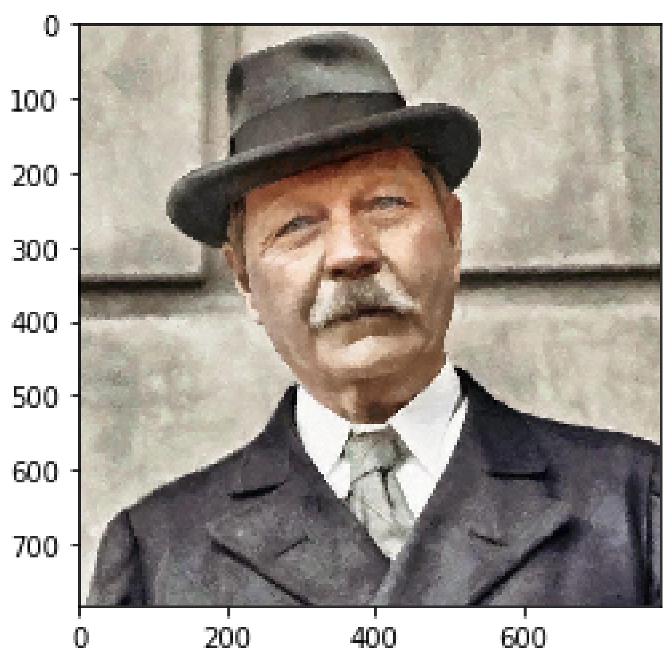
```
the breed of this dog is Golden retriever
```



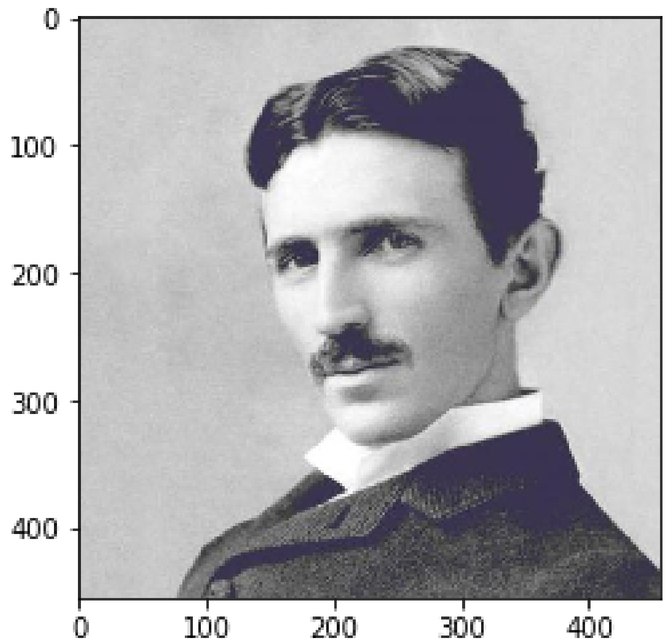

```
the breed of this dog is Cavalier king charles spaniel
```




```
'hi human
        you look like a Pharaoh hound
```

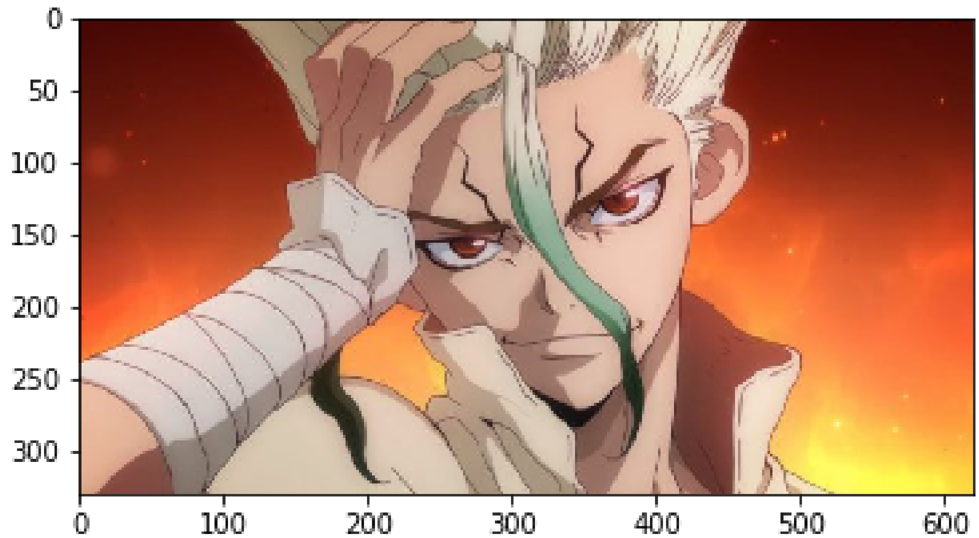


```
'hi human
        you look like a Dachshund
```




```
'hi human
        you look like a Dachshund
```

```
neither human nor dog
```



In [ ]: