dog_app

January 23, 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 [14]: 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

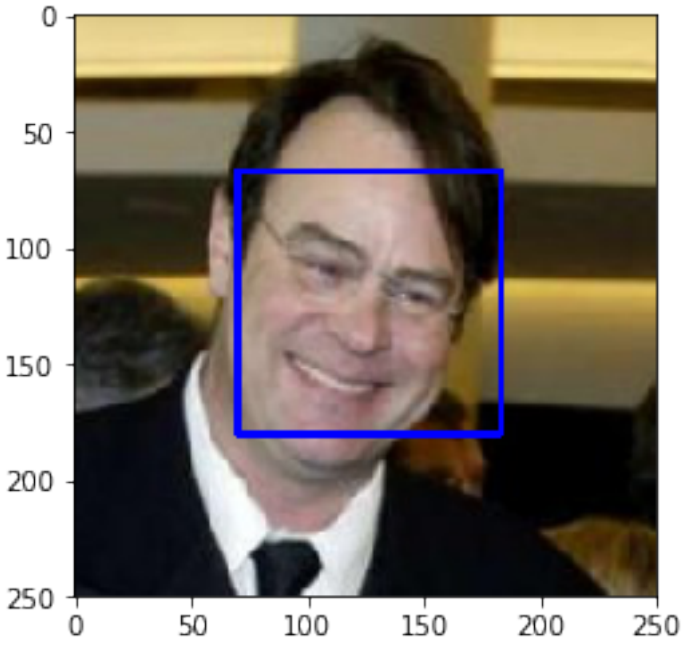
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [15]: 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 [16]: # 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:

The Percentage is 98.00% Faces Detected in the Human Dataset.

The Percentage is 17.00% Faces Detected in the Dog Dataset.

```
In [17]: 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.
         faces_vzie = np.vectorize(face_detector)

         # faces Detection in each Dataset
         human_faces_detect = faces_vzie(human_files_short)
         dog_faces_detect = faces_vzie(dog_files_short)

         # Calculation of the percentage of faces in each Dataset
         print('The Percentage is {:.2f}% Faces Detected in the Human Dataset.'.format((sum(huma
         print('The Percentage is {:.2f}% Faces Detected in the Dog Dataset.'.format((sum(dog_fa
```

```
The Percentage is 98.00% Faces Detected in the Human Dataset.
The Percentage is 17.00% Faces Detected in the Dog Dataset.
```

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 [18]:
```
### (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()

if not use_cuda:
    print('CUDA is not available.')
else:
    print('CUDA is available.')

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

VGG16.eval() # eval mode
```

```
CUDA is available.
```

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 in-

dex 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 [20]: from PIL import Image
         import torchvision.transforms as transforms
         import io

         # Set PIL to be tolerant of image files that are truncated.
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
In [21]: 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)

             data_transform = transforms.Compose([transforms.Resize(256),
                                                  transforms.CenterCrop(224),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize([0.485, 0.456, 0.406], [0.22

             img_tensor = data_transform(img)
             img_tensor.unsqueeze_(0)

             output = VGG16(img_tensor.cuda())

             _, predict_tensor = torch.max(output, 1)
             predict = np.squeeze(predict_tensor.numpy()) if not use_cuda else np.squeeze(predic

             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 [22]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             test = VGG16_predict(img_path)

             result = (test >= 151) and (test <= 268)

             return result
```

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:
Percentage of the Detected dog in human file 0.00%
Percentage of the Detected dog in dog file 100.00%

```
In [23]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         count_human_files_len = len(human_files_short)
         count_dog_files_len = len(dog_files_short)

         dog_in_human_file = sum( [dog_detector(image) for image in human_files_short] )

         dog_in_dog_file = sum( [dog_detector(image) for image in dog_files_short] )

         print('Percentage of the Detected dog in human file {:.2f}% '.format((dog_in_human_file
         print('Percentage of the Detected dog in dog file {:.2f}% '.format((dog_in_dog_file/cou
```

```
Percentage of the Detected dog in human file 0.00%
Percentage of the Detected dog in dog file 100.00%
```

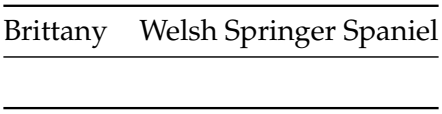
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

```
In [24]: ### (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)

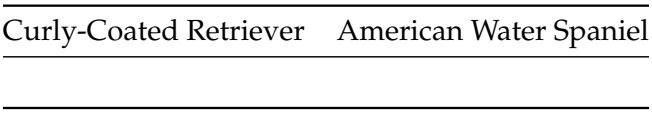
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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



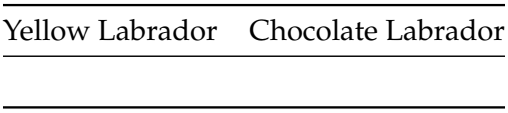
Brittany Welsh Springer Spaniel

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 [25]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_transforms = {
    'training_transforms' : transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),

    'validation_transforms' : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),

    'testing_transforms' : transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),
}

data_dir = '/data/dog_images'
train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

# Load the datasets with ImageFolder
image_datasets = {
    'training_datasets' : datasets.ImageFolder(root=train_dir,transform=data_transforms
    'validation_datasets' : datasets.ImageFolder(root=valid_dir,transform=data_transfor
    'testing_datasets' : datasets.ImageFolder(root=test_dir,transform=data_transforms['
}

# define the dataloaders
batch_size = 20
data_loaders = {
    'trainloader' : torch.utils.data.DataLoader(image_datasets['training_datasets'],bat
    'validloader' : torch.utils.data.DataLoader(image_datasets['validation_datasets'],b
    'testloader' : torch.utils.data.DataLoader(image_datasets['testing_datasets'],batch
```

```
        }

        print('Total Number of Train images: ', len(data_loaders['trainloader']) )
        print('Total Number of Valid images: ', len(data_loaders['validloader']) )
        print('Total Number of Test images: ', len(data_loaders['testloader']) )
```

```
Total Number of Train images:  334
Total Number of Valid images:  42
Total Number of Test images:  42
```

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:By Using transforms to Random Resize Crop (224), Normalize (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) The size of the input tensor is 224x224 pixles .

Answer 2:Yes I did ,augment the dataset by using RandomHorizontalFlip and RandomRotation(10), to Prevent overfitting and Increase Accuracy.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [26]: 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, 16, 3)
                 self.conv2 = nn.Conv2d(16, 32, 3)
                 self.conv3 = nn.Conv2d(32, 64, 3)
                 self.conv4 = nn.Conv2d(64, 128, 3)
                 self.conv5 = nn.Conv2d(128, 256, 3)

                 # input channels = 9216 = 256*6*6 , Output channels = 133 dog breeds
                 self.fc1 = nn.Linear(256 * 6 * 6, 133)

                 self.max_pool = nn.MaxPool2d(2, 2,ceil_mode=True)

                 self.dropout = nn.Dropout(0.20)

                 self.conv_bn1 = nn.BatchNorm2d(224,3)
                 self.conv_bn2 = nn.BatchNorm2d(16)
```

```
        self.conv_bn3 = nn.BatchNorm2d(32)
        self.conv_bn4 = nn.BatchNorm2d(64)
        self.conv_bn5 = nn.BatchNorm2d(128)
        self.conv_bn6 = nn.BatchNorm2d(256)

    def forward(self, x):

        x = F.relu(self.conv1(x))
        x = self.max_pool(x)
        x = self.conv_bn2(x)

        x = F.relu(self.conv2(x))
        x = self.max_pool(x)
        x = self.conv_bn3(x)

        x = F.relu(self.conv3(x))
        x = self.max_pool(x)
        x = self.conv_bn4(x)

        x = F.relu(self.conv4(x))
        x = self.max_pool(x)
        x = self.conv_bn5(x)

        x = F.relu(self.conv5(x))
        x = self.max_pool(x)
        x = self.conv_bn6(x)

        x = x.view(-1, 256 * 6 * 6)

        x = self.dropout(x)
        x = self.fc1(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: CNN Model Architecture is Constructed by 5 convolution layers with kernel size of 3, stride of 1 and padding of 0 with input kernal size 224x224 pixle images the result is a model contains 133 output channel to Successfully calssify the dog breeds.

The Construction of each layer as following:

First Layer: has input channels = 3 and output channels = 16
Second Layer: has input channels = 16 and output channels = 32
Third Layer: has input channels = 32 and output channels = 64
Fourth Layer: has input channels = 64 and output channels = 128
Fifth Layer: has input channels = 128 and output channels = 256

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 [27]:
```
import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

In [28]:
```
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['trainloader']):
            # 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['validloader']):
            # 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)


        # calculate average losses
        train_loss = train_loss/len(loaders['trainloader'].dataset)
        valid_loss = valid_loss/len(loaders['validloader'].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
        ## 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(12, data_loaders, model_scratch, optimizer_scratch, criterion_scr

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

```
Epoch: 1         Training Loss: 4.842657         Validation Loss: 4.650559
Validation loss decreased (inf --> 4.650559).  Saving model ...
Epoch: 2         Training Loss: 4.633876         Validation Loss: 4.453996
Validation loss decreased (4.650559 --> 4.453996).  Saving model ...
Epoch: 3         Training Loss: 4.496328         Validation Loss: 4.347566
Validation loss decreased (4.453996 --> 4.347566).  Saving model ...
Epoch: 4         Training Loss: 4.423706         Validation Loss: 4.267768
Validation loss decreased (4.347566 --> 4.267768).  Saving model ...
Epoch: 5         Training Loss: 4.374692         Validation Loss: 4.221386
Validation loss decreased (4.267768 --> 4.221386).  Saving model ...
Epoch: 6         Training Loss: 4.315807         Validation Loss: 4.168962
Validation loss decreased (4.221386 --> 4.168962).  Saving model ...
Epoch: 7         Training Loss: 4.267079         Validation Loss: 4.148462
Validation loss decreased (4.168962 --> 4.148462).  Saving model ...
Epoch: 8         Training Loss: 4.232829         Validation Loss: 4.105970
Validation loss decreased (4.148462 --> 4.105970).  Saving model ...
Epoch: 9         Training Loss: 4.197503         Validation Loss: 4.062980
Validation loss decreased (4.105970 --> 4.062980).  Saving model ...
Epoch: 10         Training Loss: 4.171079         Validation Loss: 4.027392
Validation loss decreased (4.062980 --> 4.027392).  Saving model ...
Epoch: 11         Training Loss: 4.119433         Validation Loss: 3.999543
Validation loss decreased (4.027392 --> 3.999543).  Saving model ...
Epoch: 12         Training Loss: 4.096230         Validation Loss: 3.966234
Validation loss decreased (3.999543 --> 3.966234).  Saving model ...
```

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 [35]: 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['testloader']):
```

```
            # 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 += loss.item()*data.size(0)
            # 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)

        # calculate average loss
        test_loss = test_loss/len(loaders['testloader'].dataset)

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

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

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

```
Test Loss: 3.946302


Test Accuracy: 10% (89/836)
```

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 [36]: ## TODO: Specify data loaders
         # Use the same data loaders
         # take a copy
         loaders_transfer = data_loaders.copy()
```

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

         ## TODO: Specify model architecture
         model_transfer = models.vgg19(pretrained=True)

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

```
In [38]: for param in model_transfer.parameters():
             param.requires_grad = False

         model_transfer.classifier[6]= nn.Linear(4096 , 133, bias=True)

         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: Becase this architecture is a keras model with 19 layer network that has an input size of 224X224 pixel image, these model detect features in the images they were not trained on. it is required to use transfer learning to train the network to classify the dogs breed, VGG19 will allow me to get the best-performing Image Classification results and High Accuracy , Accuarcy reached 83%.

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 [39]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(filter(lambda t: t.requires_grad,model_transfer.paramete
```

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 [40]: # train the model
         # change the epochs number to get a Better Performance
         n_epochs = 15

         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,c
                                use_cuda,'model_transfer.pt')
```

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

```
Epoch: 1         Training Loss: 4.484979         Validation Loss: 3.471476
Validation loss decreased (inf --> 3.471476).  Saving model ...
Epoch: 2         Training Loss: 3.536921         Validation Loss: 2.511420
Validation loss decreased (3.471476 --> 2.511420).  Saving model ...
Epoch: 3         Training Loss: 2.915362         Validation Loss: 1.940120
Validation loss decreased (2.511420 --> 1.940120).  Saving model ...
Epoch: 4         Training Loss: 2.530745         Validation Loss: 1.591755
Validation loss decreased (1.940120 --> 1.591755).  Saving model ...
Epoch: 5         Training Loss: 2.262126         Validation Loss: 1.362016
Validation loss decreased (1.591755 --> 1.362016).  Saving model ...
Epoch: 6         Training Loss: 2.093935         Validation Loss: 1.201391
Validation loss decreased (1.362016 --> 1.201391).  Saving model ...
Epoch: 7         Training Loss: 1.946214         Validation Loss: 1.084078
Validation loss decreased (1.201391 --> 1.084078).  Saving model ...
Epoch: 8         Training Loss: 1.825524         Validation Loss: 0.991052
Validation loss decreased (1.084078 --> 0.991052).  Saving model ...
Epoch: 9         Training Loss: 1.762176         Validation Loss: 0.922221
Validation loss decreased (0.991052 --> 0.922221).  Saving model ...
Epoch: 10         Training Loss: 1.688612         Validation Loss: 0.865788
Validation loss decreased (0.922221 --> 0.865788).  Saving model ...
Epoch: 11         Training Loss: 1.604387         Validation Loss: 0.819207
Validation loss decreased (0.865788 --> 0.819207).  Saving model ...
Epoch: 12         Training Loss: 1.584135         Validation Loss: 0.781208
Validation loss decreased (0.819207 --> 0.781208).  Saving model ...
Epoch: 13         Training Loss: 1.554222         Validation Loss: 0.746185
Validation loss decreased (0.781208 --> 0.746185).  Saving model ...
Epoch: 14         Training Loss: 1.511546         Validation Loss: 0.717742
Validation loss decreased (0.746185 --> 0.717742).  Saving model ...
Epoch: 15         Training Loss: 1.458214         Validation Loss: 0.695670
Validation loss decreased (0.717742 --> 0.695670).  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 [41]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.693232


Test Accuracy: 83% (702/836)
```

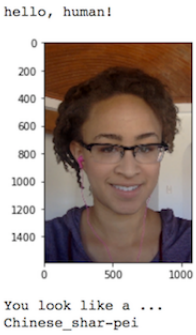
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

```
In [44]: def process_image_to_tensor(image):
             ''' Scales, crops, and normalizes a PIL image for a PyTorch model,
                 returns an tensor array
             '''
             # define transforms for the training data and testing data
             prediction_transforms = transforms.Compose([transforms.Resize(224),
                                                  transforms.CenterCrop(224),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize([0.485, 0.456, 0.406],
                                                                       [0.229, 0.224, 0.225])])

             img = Image.open( image ).convert('RGB')
             img_tensor = prediction_transforms(img)[:3,:,:].unsqueeze(0)

             return img_tensor

In [45]: ### 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 data_loaders['trainloader'].datas

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

             if use_cuda:
                 image_tensor = image_tensor.cuda()

             # get sample outputs
             output = model_transfer(image_tensor)
             # convert output probabilities to predicted class
             _, preds_tensor = torch.max(output, 1)
             pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso

             return class_names[pred]
```

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.



Sample Human Output

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 [46]: def display_image(img_path, title="Title"):
             ''' display the image to the user
             '''
             image = Image.open(img_path)
             plt.title(title)
             plt.imshow(image)
             plt.show()

In [47]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             if(face_detector(img_path)):
                 print("Human is Detected")
                 predicted_breed = predict_breed_transfer(img_path)
                 print("You look like a ...")
                 print(predicted_breed)

             elif(dog_detector(img_path)):
                 print("Dog is detected")
                 prediction = predict_breed_transfer(img_path)
                 print("Dog Breed: {0}".format(prediction))

             else:
```

```
        print("Neither Dog nor Human is detected")
        print("Error !!!. Try with other Image")
```

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: An Average Result !

Three possible points for improvement:

1- improve the training images, that only the dog is inside the image and the background has not so much details.

2- Increasing the Dataset to improve the performance of the model.

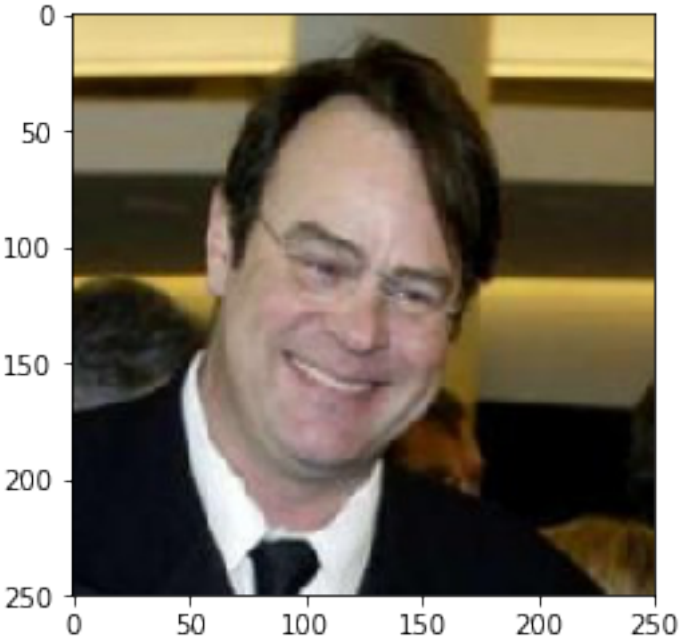
3- Increasing the training time to improve the performance of the model.

4- Enhance the classifier connected layers to train the net faster (dimensions , dropout layers).

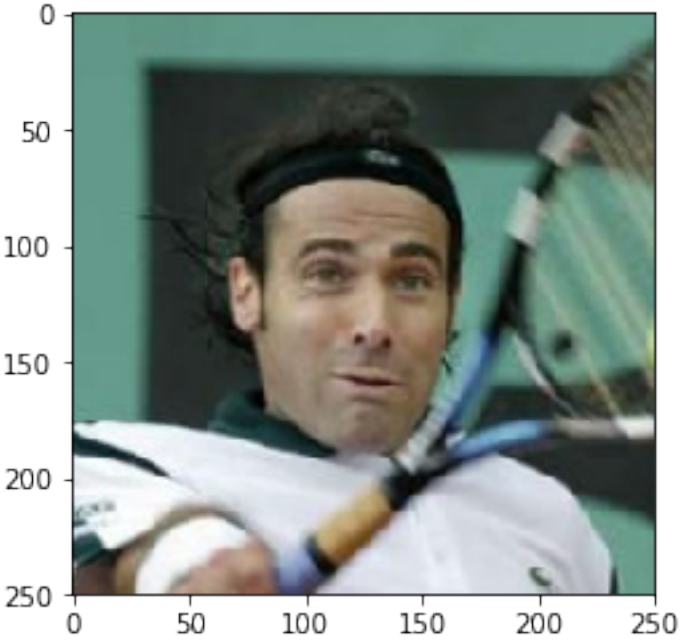
5- Change the CNN Model to improve the performance of the model .

In [49]:
```
## 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
count = 0;
for file in np.hstack((human_files[:5], dog_files[:15])):
    print("Input:",count+1)
    run_app(file)
    count=count+1;
```

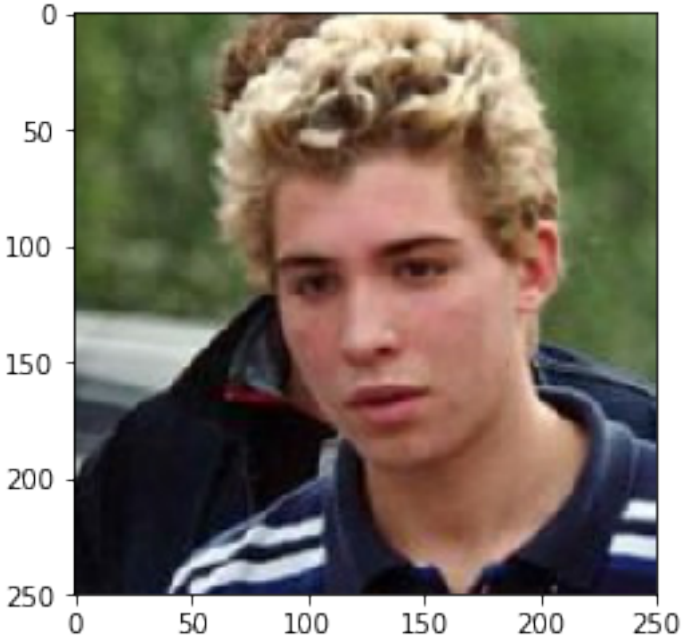
```
Input: 1
```



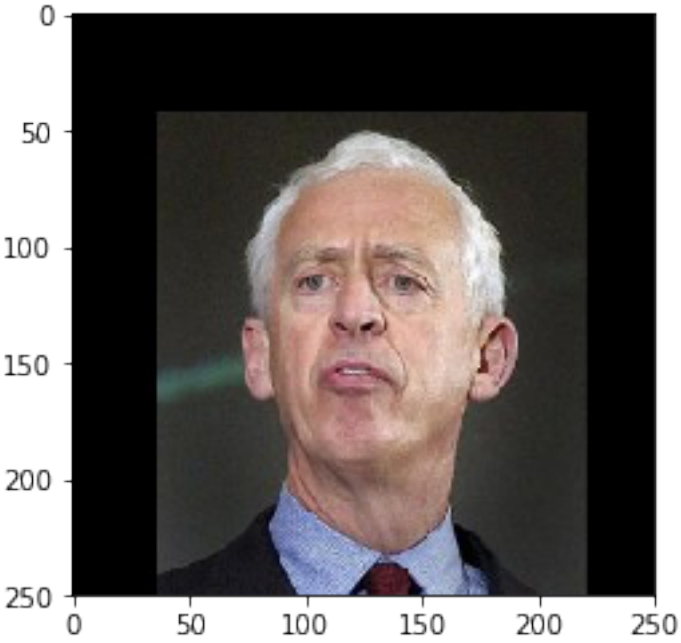

```
Human is Detected
You look like a ...
Collie
Input: 2
```



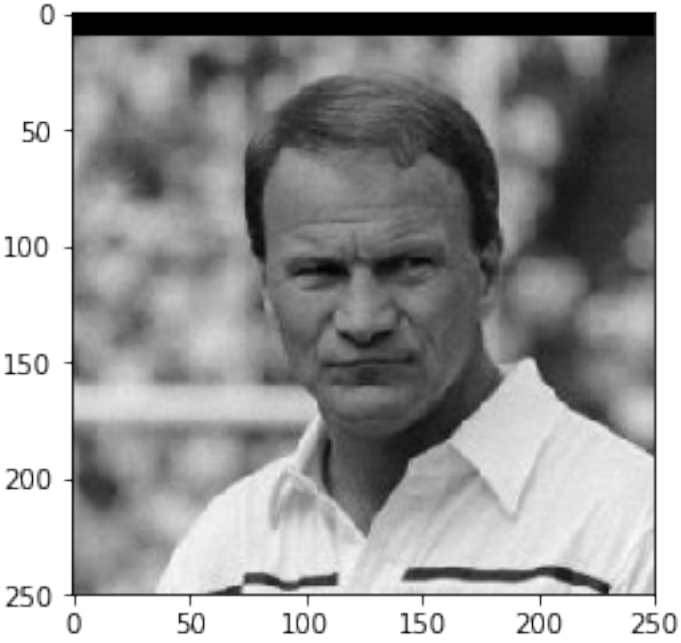
```
Human is Detected
You look like a ...
Dachshund
Input: 3
```



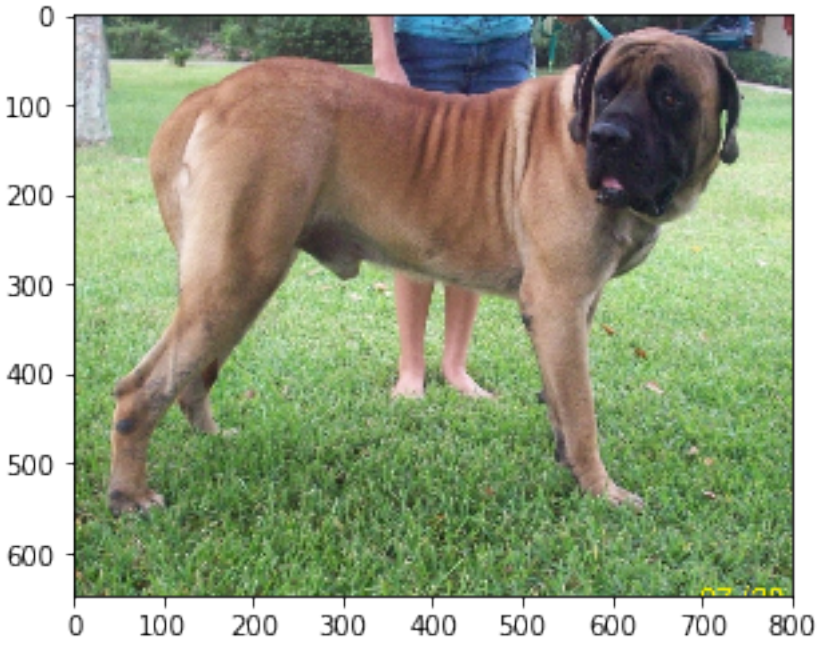

```
Human is Detected
You look like a ...
Poodle
Input: 4
```



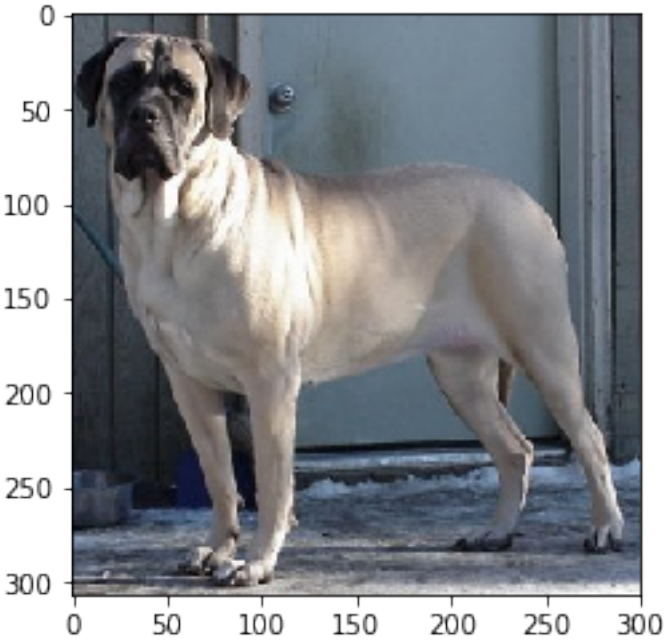

```
Human is Detected
You look like a ...
Italian greyhound
Input: 5
```



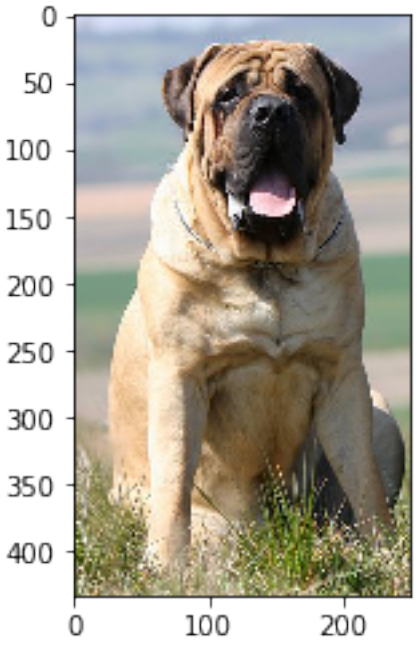
```
Human is Detected
You look like a ...
Bull terrier
Input: 6
```



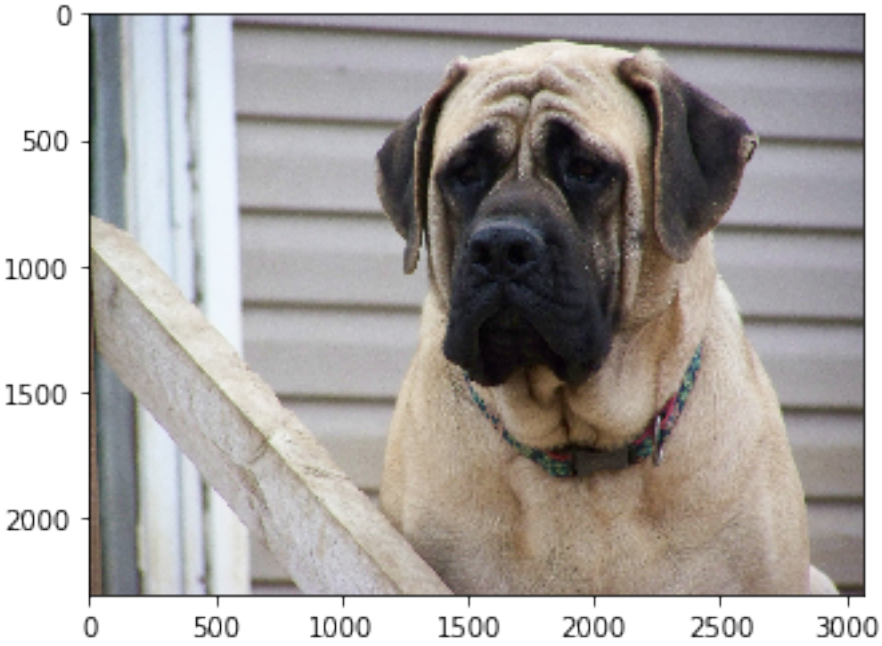

```
Dog is detected
Dog Breed: Bullmastiff
Input: 7
```



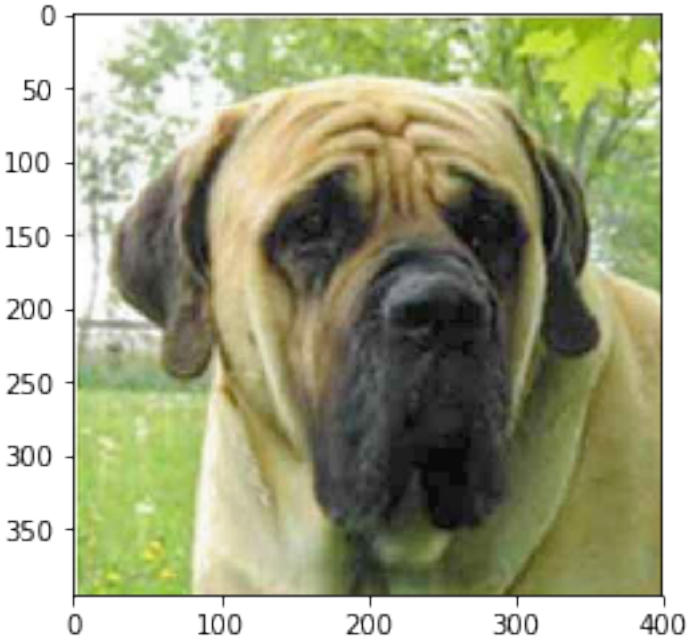

```
Dog is detected
Dog Breed: Bullmastiff
Input: 8
```



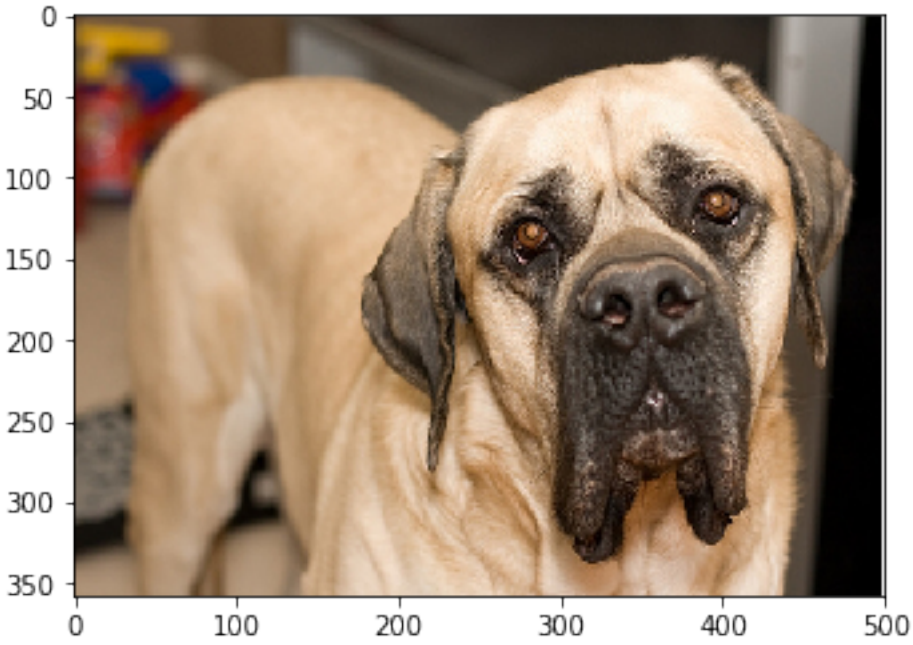
```
Dog is detected
Dog Breed: Bullmastiff
Input: 9
```



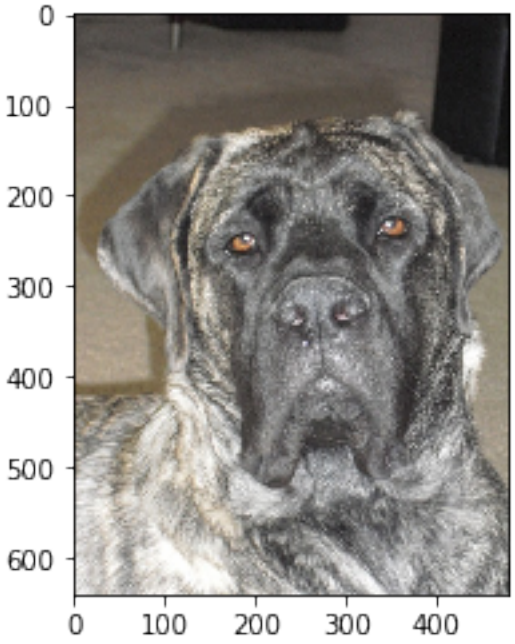

```
Dog is detected
Dog Breed: Mastiff
Input: 10
```




```
Dog is detected
Dog Breed: Mastiff
Input: 11
```

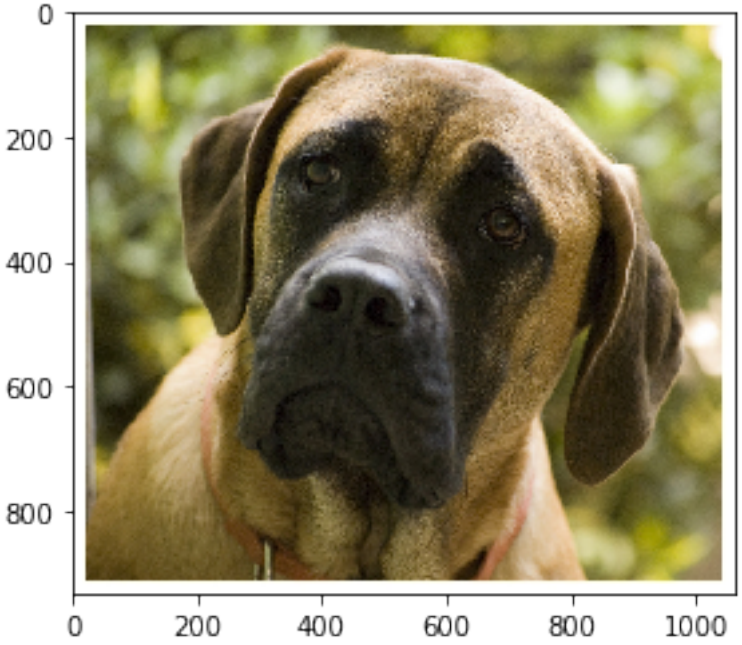


```
Dog is detected
Dog Breed: Mastiff
Input: 12
```

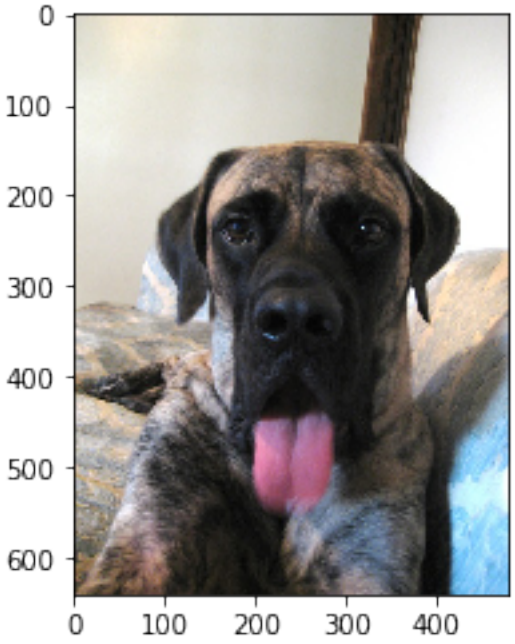



```
Dog is detected
Dog Breed: Mastiff
Input: 13
```

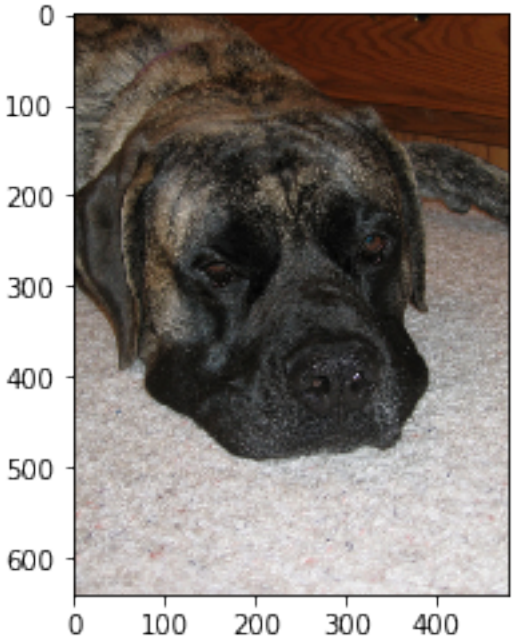

```
Dog is detected
Dog Breed: Mastiff
Input: 14
```



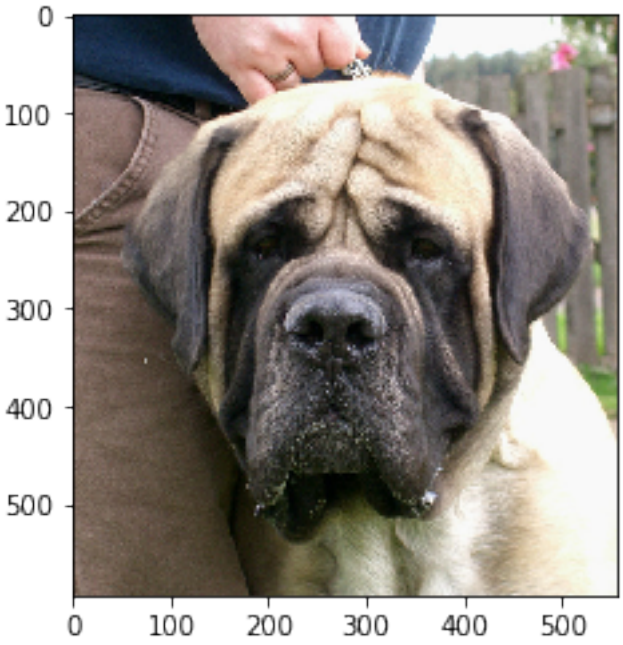
```
Dog is detected
Dog Breed: Mastiff
Input: 15
```



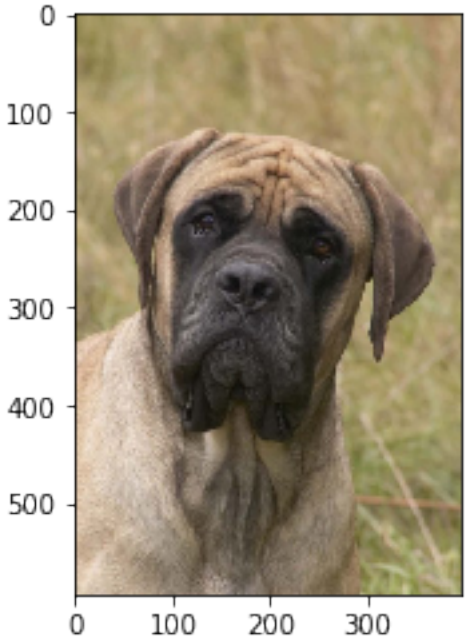

```
Dog is detected
Dog Breed: Mastiff
Input: 16
```



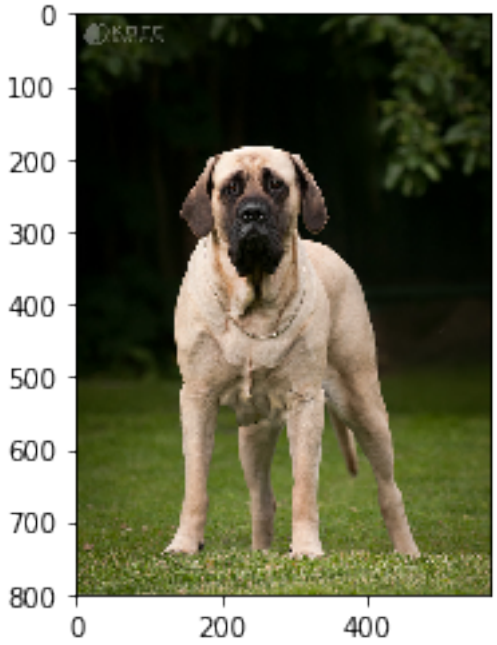

```
Dog is detected
Dog Breed: Mastiff
Input: 17
```



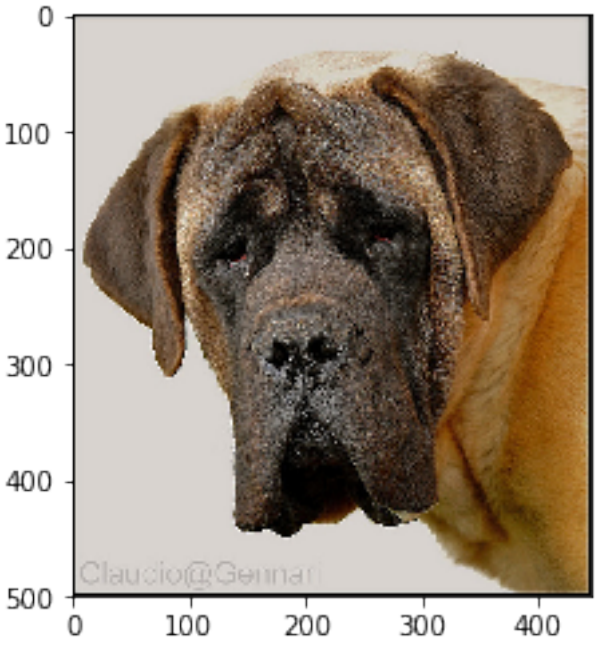
```
Dog is detected
Dog Breed: Mastiff
Input: 18
```




```
Dog is detected
Dog Breed: Bullmastiff
Input: 19
```




```
Dog is detected
Dog Breed: Mastiff
Input: 20
```



```
Dog is detected
Dog Breed: Mastiff
```

In [ ]: