dog_app

November 3, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

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

        dog_files[-5]
```

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

```
Out[1]: '/data/dog_images/valid/092.Keeshond/Keeshond_06238.jpg'
```

Step 1: Detect Humans

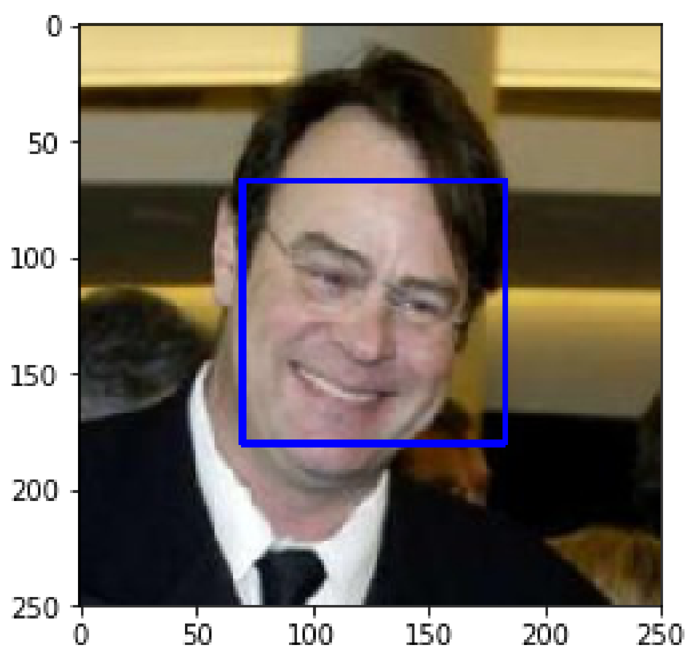
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        print('Gray array type is {} and shape is {}'.format(type(gray), gray.shape))
```

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

print('Face array type is {} and shape is {}'.format(type(faces), faces.shape))

# 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()
```

```
Gray array type is <class 'numpy.ndarray'> and shape is (250, 250)
Face array type is <class 'numpy.ndarray'> and shape is (1, 4)
Number of faces detected: 1
```



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        human_face_count = 0
        dog_face_count = 0
        for i in range(len(human_files_short)):
            if face_detector(human_files_short[i]):
                human_face_count += 1
            if face_detector(dog_files_short[i]):
                dog_face_count += 1

        human_face_percent = (human_face_count / len(human_files_short))*100
        dog_face_percent = (dog_face_count / len(dog_files_short))*100
```

```
print('The percent of human face detected are {}'.format(human_face_percent))
print('The percent of dog face detected are {}'.format(dog_face_percent))
```

```
The percent of human face detected are 98.0
The percent of dog face detected are 17.0
```

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

In [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 [5]:
```
import torch
import torchvision.models as models

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

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

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

print(VGG16)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:09<00:00, 57998247.28it/s]
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

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

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

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

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

```
In [6]: from PIL import Image
        import torchvision.transforms as transforms
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

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

            Args:
                img_path: path to an image

            Returns:
                Index corresponding to VGG-16 model's prediction
            '''
            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            ## Return the *index* of the predicted class for that image
            image = Image.open(img_path)
            normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])
            transform = transforms.Compose([
             transforms.Resize(256),
             transforms.CenterCrop(224),
             transforms.ToTensor(),
             normalize
            ])
            t_img = transform(image)
            batch_t = torch.unsqueeze(t_img, 0)
            VGG16.eval()



            if use_cuda:
                out = VGG16(batch_t.cuda())
            else:
                out = VGG16(batch_t)
            _, index = torch.max(out,1)
```

```
        index = index.item()

        return index # predicted class index


    # for testing the method
    VGG16_predict(dog_files_short[0])
```

Out[6]: 243

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            pred_key = VGG16_predict(img_path)
            return 151 <= pred_key <= 268 # true/false

        # for testing the method
        dog_detector(dog_files_short[0])
```

Out[7]: True

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

- The percentage of the images in human_files_short have detected 0% dogs. - The percentage of the images in dog_files_short have detected 100% dogs

```
In [8]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        human_face_count = 0
        dog_face_count = 0
        for i in range(len(human_files_short)):
            if dog_detector(human_files_short[i]):
                human_face_count += 1
            if dog_detector(dog_files_short[i]):
                dog_face_count += 1
```

```python
human_face_percent = (human_face_count / len(human_files_short))*100
dog_face_percent = (dog_face_count / len(dog_files_short))*100

print('The percent of human face detected are {}'.format(human_face_percent))
print('The percent of dog face detected are {}'.format(dog_face_percent))
```

```
The percent of human face detected are 0.0
The percent of dog face detected are 100.0
```

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 [9]:
```python
### (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 [9]: import os
        from torchvision import datasets

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes

        print(glob("/data/dog_images/*/")) # checking for all available directories
        print('Number of dogs breeds - {}'.format(len(glob("/data/dog_images/train/*/"))))


        normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                         std=[0.229, 0.224, 0.225])


        data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.ToTensor(),
                                         normalize]),
                           'val': transforms.Compose([transforms.Resize(256),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         normalize]),
                           'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                         transforms.ToTensor(),
                                         normalize])
                          }


        train_data = datasets.ImageFolder('/data/dog_images/train/',transform = data_transforms[
        test_data = datasets.ImageFolder('/data/dog_images/test/', transform = data_transforms['
        valid_data = datasets.ImageFolder('/data/dog_images/valid/', transform = data_transforms

        batch_size = 20
```

```
train_loader = torch.utils.data.DataLoader(train_data, batch_size = batch_size, shuffle
test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, shuffle =
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size= batch_size, shuffle =
```

```
['/data/dog_images/train/', '/data/dog_images/test/', '/data/dog_images/valid/']
Number of dogs breeds - 133
```

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: - I have used to transform to centor crop the image and picked up the size of 224 as it will be the ideal size for the VGG16 input dataset and will be standard size for all the images in test, train and validation dataset - I haven't augmented the data as we are having enough images and we don't want to overfit the dataset by augmenting the dataset.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [23]: 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__()
                 self.conv1 = nn.Conv2d(3, 32, 3, padding = 1)
                 self.conv2 = nn.Conv2d(32, 64, 3, padding = 1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding = 1)
                 self.conv4 = nn.Conv2d(128,64,3, padding =1)
                 self.conv5 = nn.Conv2d(64, 20, 3, padding =1)
                 self.pool = nn.MaxPool2d(2,2)
                 self.drop = nn.Dropout(0.3)
                 self.fc1 = nn.Linear(20*7*7, 8000)
                 self.fc2 = nn.Linear(8000, 4000)
                 self.fc3 = nn.Linear(4000, 2000)
                 self.fc4 = nn.Linear(2000, 1000)
                 self.fc5 = nn.Linear(1000, 500)
                 self.fc6 = nn.Linear(500, 133)
                 ## Define layers of a CNN

             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)))
```

```
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))
        x = x.view(-1, 20*7*7)
        x = self.drop(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.drop(x)
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = self.drop(x)
        x = F.relu(self.fc5(x))
        x = self.drop(x)
        x = self.fc6(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()

print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(64, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (drop): Dropout(p=0.3)
  (fc1): Linear(in_features=980, out_features=8000, bias=True)
  (fc2): Linear(in_features=8000, out_features=4000, bias=True)
  (fc3): Linear(in_features=4000, out_features=2000, bias=True)
  (fc4): Linear(in_features=2000, out_features=1000, bias=True)
  (fc5): Linear(in_features=1000, out_features=500, bias=True)
  (fc6): Linear(in_features=500, out_features=133, bias=True)
)
```

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

Answer:

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))

(conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv4): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv5): Conv2d(64, 20, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(drop): Dropout(p=0.3)
(fc1): Linear(in_features=980, out_features=8000, bias=True)
(fc2): Linear(in_features=8000, out_features=4000, bias=True)
(fc3): Linear(in_features=4000, out_features=2000, bias=True)
(fc4): Linear(in_features=2000, out_features=1000, bias=True)
(fc5): Linear(in_features=1000, out_features=500, bias=True)
(fc6): Linear(in_features=500, out_features=133, bias=True)

The first convolution layer will apply 32 filter to the input image by doing the maxpooling and taking the maximum image matrix, the second layer will convert these 32 filtered images to 64, similary third will convert from 64 -> 128, fourth to 128->64 and finally fifth convolution will convert it from 64 to 20 matching the batch size that we have choosed. Each convolution layer is applied to maxpool filter and kernel size is taken as (3,3) with paddin of 1 to maintain consistency.

Then we will flatten the output coming from fifth convolution layer with view(-1, 2077).

Then we will feed the flattened output to the FeedForward network with first layer converting the 2077 nodes to 8000 and apply relu function to it. Similary second fc layer will convert it from 8000 to 4000, third to 4000->2000, forth 2000->1000, fifth->500 and finally the fifth layer will convert it from 500 to 133 which is the output of our classification classes that we have in our network. All the linear layer except the last layer is applied to ReLu function and the dropout of 0.3 is choosed to avoid overfitting of data.

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 [24]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.1)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [17]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             for epoch in range(1, n_epochs+1):
```

```python
    # 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()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        ## 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_loss)

    ######################
    # 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
        valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

        if(valid_loss <= valid_loss_min ):
            torch.save(model.state_dict(), save_path)


    # 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

# return trained model
```

```
            return model
```

```
In [45]: # train the model
         loaders_scratch = {'test': test_loader, 'train': train_loader, 'valid': valid_loader}
         model_scratch = train(40, 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: 4.560718         Validation Loss: 4.269367
Epoch: 2         Training Loss: 4.536169         Validation Loss: 4.653229
Epoch: 3         Training Loss: 4.510549         Validation Loss: 4.796456
Epoch: 4         Training Loss: 4.496562         Validation Loss: 4.554116
Epoch: 5         Training Loss: 4.470639         Validation Loss: 4.588790
Epoch: 6         Training Loss: 4.458082         Validation Loss: 4.539079
Epoch: 7         Training Loss: 4.446216         Validation Loss: 4.297996
Epoch: 8         Training Loss: 4.414851         Validation Loss: 4.494345
Epoch: 9         Training Loss: 4.394037         Validation Loss: 4.470734
Epoch: 10         Training Loss: 4.358184         Validation Loss: 4.466747
Epoch: 11         Training Loss: 4.323321         Validation Loss: 4.454602
Epoch: 12         Training Loss: 4.302419         Validation Loss: 4.477540
Epoch: 13         Training Loss: 4.272484         Validation Loss: 4.188638
Epoch: 14         Training Loss: 4.254348         Validation Loss: 4.308859
Epoch: 15         Training Loss: 4.226531         Validation Loss: 4.028267
Epoch: 16         Training Loss: 4.170674         Validation Loss: 4.306122
Epoch: 17         Training Loss: 4.160899         Validation Loss: 3.912169
Epoch: 19         Training Loss: 4.148218         Validation Loss: 3.699942
Epoch: 20         Training Loss: 4.102117         Validation Loss: 3.951129
Epoch: 21         Training Loss: 4.076646         Validation Loss: 4.125452
Epoch: 22         Training Loss: 4.052876         Validation Loss: 4.055559
Epoch: 23         Training Loss: 4.032739         Validation Loss: 4.115960
Epoch: 24         Training Loss: 4.014862         Validation Loss: 4.003699
Epoch: 25         Training Loss: 3.991220         Validation Loss: 4.238549
Epoch: 26         Training Loss: 3.963936         Validation Loss: 4.415935
Epoch: 27         Training Loss: 3.959662         Validation Loss: 3.598539
Epoch: 28         Training Loss: 3.917161         Validation Loss: 3.710966
Epoch: 29         Training Loss: 3.896533         Validation Loss: 3.820806
Epoch: 30         Training Loss: 3.886880         Validation Loss: 4.099259
Epoch: 31         Training Loss: 3.908242         Validation Loss: 3.817397
Epoch: 32         Training Loss: 3.864399         Validation Loss: 3.878365
Epoch: 33         Training Loss: 3.855880         Validation Loss: 4.012229
Epoch: 34         Training Loss: 3.814482         Validation Loss: 3.722645
Epoch: 35         Training Loss: 3.802733         Validation Loss: 3.849428
Epoch: 36         Training Loss: 3.796173         Validation Loss: 3.582818
Epoch: 37         Training Loss: 3.759976         Validation Loss: 3.793834
Epoch: 38         Training Loss: 3.776301         Validation Loss: 4.376365
Epoch: 39         Training Loss: 3.756077         Validation Loss: 3.504594
```

```
Epoch: 40        Training Loss: 3.758795        Validation Loss: 3.965331
```

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 [18]: 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))
```

```
In [46]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.786480


Test Accuracy: 11% (93/836)
```

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 [15]: ## TODO: Specify data loaders
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

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

         # TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)
         for param in model_transfer.features.parameters():
             param.requires_grad = False
         model_transfer.classifier[6] = nn.Sequential(
                                 nn.Linear(4096, 1048),
                                 nn.ReLU(),
                                 nn.Dropout(0.3),
                                 nn.Linear(1048, 512),
                                 nn.ReLU(),
                                 nn.Dropout(0.3),
                                 nn.Linear(512, 256),
                                 nn.ReLU(),
                                 nn.Dropout(0.3),
                                 nn.Linear(256, 133))

         fc_parameters = model_transfer.classifier.parameters()
         for param in fc_parameters:
             param.requires_grad = True

         if use_cuda:
             model_transfer = model_transfer.cuda()

         model_transfer
```

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 have choosed the vgg16 pretrained model and extended the last feedforward layer with sequential network where the first layer is Linear layer with 4096 input(this is the input that we are getting from the previous layer) and converted it to 1048, the second layer to 1048->512, third layer from 512->256 and last layer from 256 to 133 which is the output of the network. I have set the autograd property of parameters to True so that the FeedForward network can learn from the data using Backpropogation.

Vgg16 is a good pretrained model with already having trained data on dog breeds, so this model is ideal for our image classification problem, where we are trying to detect the dog breed. As we can see below with just 20 epochs of traning our network, we were able to get 84% accuracy which is quite good given the different classes of dog breeds.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [20]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         # for vgg 16 model
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr = 0.01)
```

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 [61]: # train the model
         loaders_transfer = {'test': test_loader, 'train': train_loader, 'valid': valid_loader}
         model_transfer =  train(25, loaders_transfer, model_transfer, optimizer_transfer, crite

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

```
        ---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-61-7ea2407df4fc> in <module>()
          4
          5 loaders_transfer = {'test': test_loader, 'train': train_loader, 'valid': valid_loade
    ----> 6 model_transfer =  train(25, loaders_transfer, model_transfer, optimizer_transfer, cr
          7
          8 # load the model that got the best validation accuracy (uncomment the line below)


        <ipython-input-17-87821e0b85d0> in train(n_epochs, loaders, model, optimizer, criterion,
         13          ###################
         14          model.train()
    ---> 15          for batch_idx, (data, target) in enumerate(loaders['train']):
         16              # move to GPU
```

```
     17             if use_cuda:


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    101         sample = self.loader(path)
    102         if self.transform is not None:
--> 103             sample = self.transform(sample)
    104         if self.target_transform is not None:
    105             target = self.target_transform(target)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
     47     def __call__(self, img):
     48         for t in self.transforms:
---> 49             img = t(img)
     50         return img
     51


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
     74             Tensor: Converted image.
     75         """
---> 76         return F.to_tensor(pic)
     77
     78     def __repr__(self):


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transform
     79     # put it from HWC to CHW format
     80     # yikes, this transpose takes 80% of the loading time/CPU
---> 81     img = img.transpose(0, 1).transpose(0, 2).contiguous()
     82     if isinstance(img, torch.ByteTensor):
```

```
     83         return img.float().div(255)


    KeyboardInterrupt:
```

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

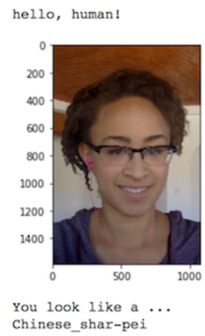
```
Test Loss: 0.502491


Test Accuracy: 84% (709/836)
```

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 [51]: ### 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 loaders_transfer['train'].dataset

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = Image.open(img_path).convert('RGB')
             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                              transforms.ToTensor(),
                                              normalize])

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

             model_transfer.cpu()
             idx = torch.argmax(model_transfer(image))
             return class_names[idx]

         #for testing
         predict_breed_transfer(dog_files[0])
```

```
Out[51]: 'Mastiff'
```



Sample Human Output

Step 5: Write your Algorithm
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [52]:
```
### 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 dog_detector(img_path) is True:
        prediction = predict_breed_transfer(img_path)
        print("Dogs Detected!\nIt looks like a {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(img_path)
        print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predic
    else:
        print("Error! Can't detect anything..")
```

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.

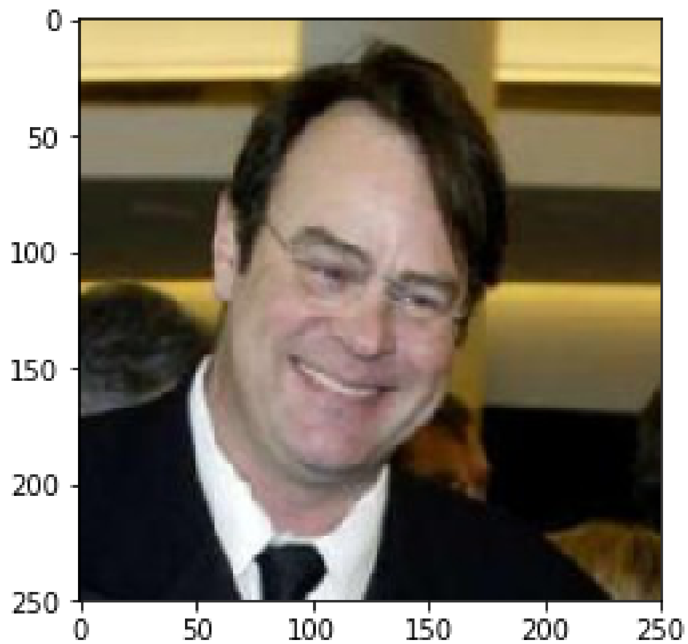
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: The output is certainly better, I wan't expecting 84% score for images. The model can be further improved- - We can provide more data so that our model can train much better - We can train our model for other transfer learning models such as ResNet50 - We can tune the hyperparameter futher to get the more better solution for our datasets

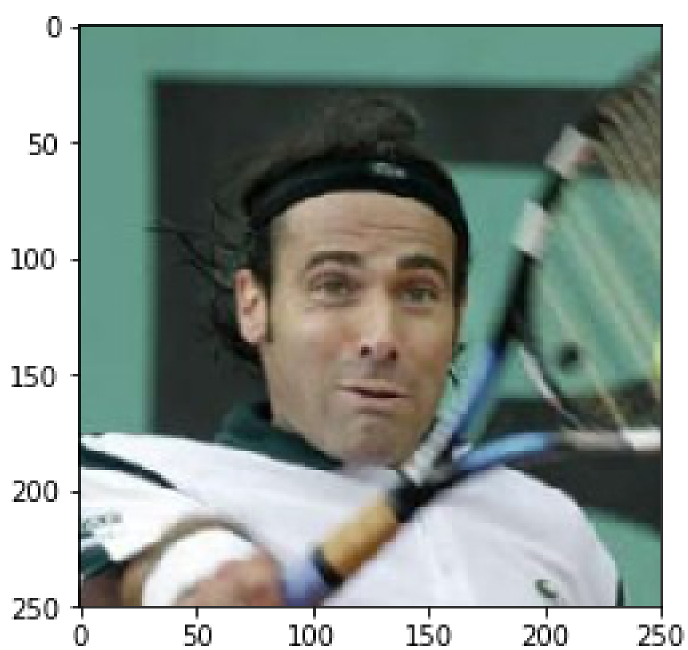
```
In [3]: human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/test/*/*"))


        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 836 total dog images.
```

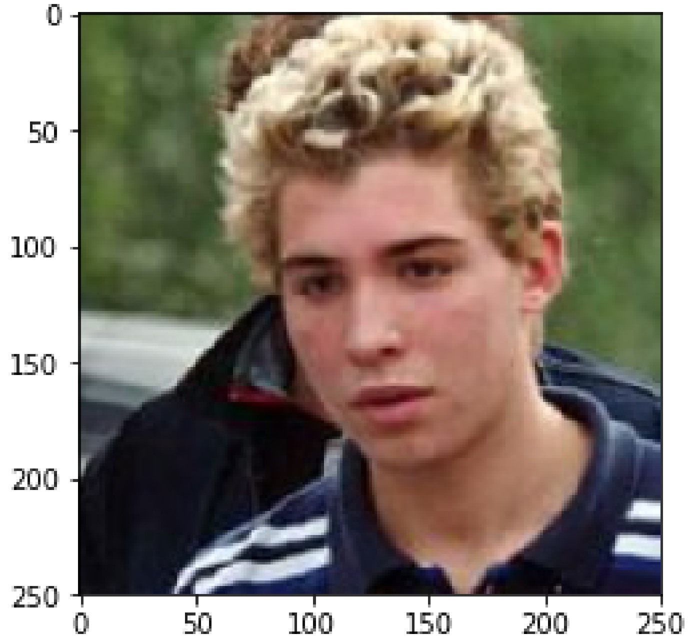
```
In [53]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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



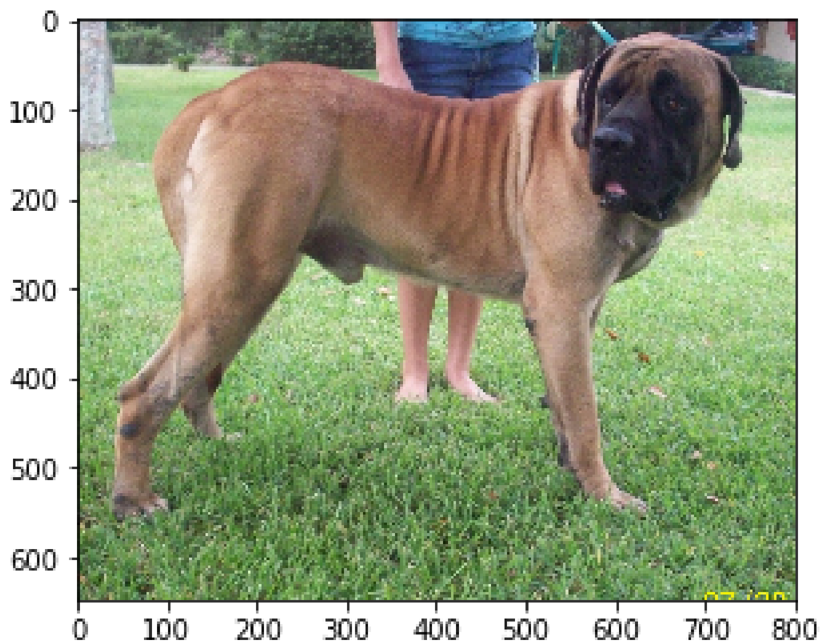
```
Hello, human!
If you were a dog..You may look like a Beagle
```



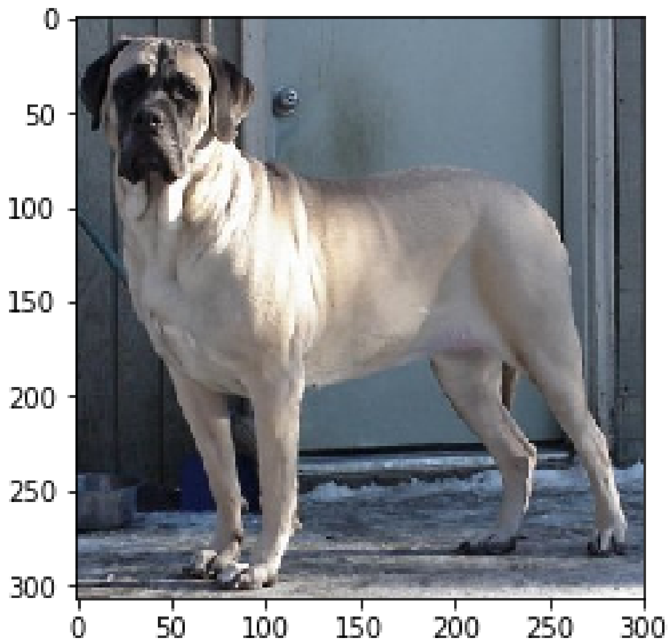
```
Hello, human!
If you were a dog..You may look like a Ibizan hound
```



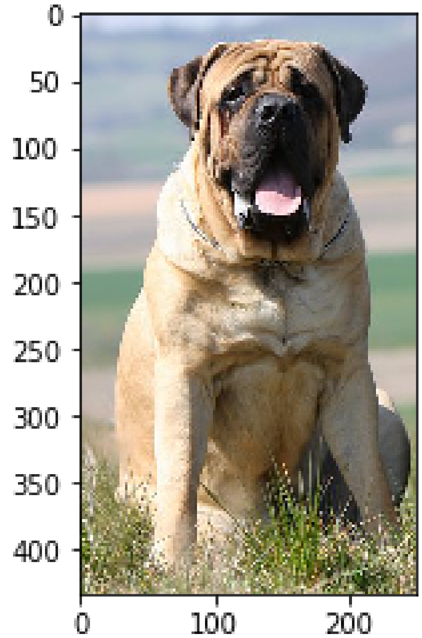

```
Hello, human!
If you were a dog..You may look like a Irish water spaniel
```



```
Dogs Detected!
It looks like a Mastiff
```




```
Dogs Detected!
It looks like a Mastiff
```




```
Dogs Detected!
It looks like a Bullmastiff
```

```
In [ ]:
```