dog_app

June 6, 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 [65]: 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

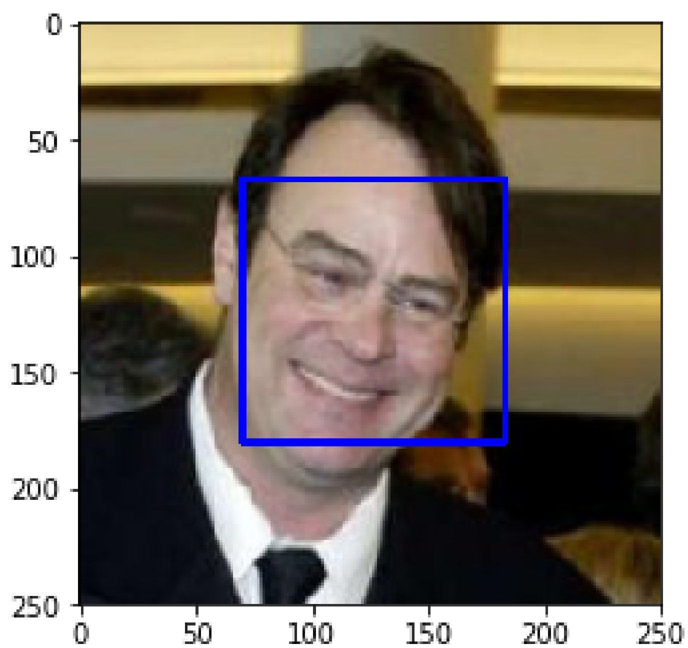
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [18]: 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 [19]:
```
# 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)
Percentage of face detection in human_files : 98.0
Percentage of face detection in dog_files : 17.0

In [20]:
```
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_percentage = 0
dog_percentage = 0

for i in range(100):
    person = face_detector(human_files_short[i])
    if person:
        human_percentage += 1
    dog = face_detector(dog_files_short[i])
    if dog:
        dog_percentage += 1

print("percentage of face detection  in human_files : ",((human_percentage/100)*100))
print("percentage of face detection  in dog_files : ",((dog_percentage/100)*100))
```

```
percentage of face detection  in human_files :  98.0
percentage of face detection  in dog_files :  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 [3]: ### (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 [52]: 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()
```

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 [66]:
```
from PIL import Image
import torchvision.transforms as transforms


if use_cuda:
    VGG16 = VGG16.cuda()
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)


    test_transforms = transforms.Compose([transforms.Resize(255),
                                          transforms.CenterCrop(224),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])

    transformed = test_transforms(img).float()

    transformed = transformed.unsqueeze(0)

    if use_cuda:
        transformed.cuda()

    output = VGG16(test_transforms)


    return output.data.argmax(dim=1)# predicted class index
```

In [85]:
```
def VGG16_predict(img_path):

    img = Image.open(img_path)
```

```
    test_transforms = transforms.Compose([transforms.Resize(255),
                                          transforms.CenterCrop(224),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])

    transformed = test_transforms(img).float()

    images = transformed.unsqueeze(0)


    if use_cuda:
        images = images.cuda()
    output = VGG16(images)


    return output.data.argmax(dim=1)
```

In [86]:
```
VGG16_predict(dog_files_short[64])
```

Out[86]:
```
tensor([ 236], device='cuda:0')
```

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 [87]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    dog = False
    value = VGG16_predict(img_path)
    if value >= 151 and value <= 268:
        dog = True
    return dog
```

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 dog detection in human_files : 1.0
percentage of dog detection in dog_files : 99.0

In [88]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_percentage = 0
dog_percentage = 0
for i in range(100):
    person = dog_detector(human_files_short[i])
    if person:
        human_percentage += 1
    dog = dog_detector(dog_files_short[i])
    if dog:
        dog_percentage += 1

print("percentage of dog detection  in human_files : ",((human_percentage/100)*100))
print("percentage of dog detection  in dog_files : ",((dog_percentage/100)*100))
```

```
percentage of dog detection  in human_files :  1.0
percentage of dog detection  in dog_files :  99.0
```

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 [ ]:
```
### (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

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 [1]: import os
        from torchvision import datasets
        from PIL import ImageFile
        import torchvision.transforms as transforms
        import torch
        import torchvision.models as models
        import numpy as np

        use_cuda = torch.cuda.is_available()

        ImageFile.LOAD_TRUNCATED_IMAGES = True

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

        data_dir = '/data/dog_images'


        batch_size = 20
```

```
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5],
                                                           [0.5, 0.5, 0.5])])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5],
                                                           [0.5, 0.5, 0.5])])


train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
valid_data = datasets.ImageFolder(data_dir + '/valid', transform=valid_transforms)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=Tru
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)
validloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)

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

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 resize all images to 255, crooping to 224 and a batch size of 20 . I based in the previous exercise of the course to select this size for the images.

To augment the dataset first I made a random rotation with a range of 30 degrees,also also made aRandomHorizontalFlip

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [2]: import torch.nn as nn
        import torch.nn.functional as F

        # define the CNN architecture
```

```python
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,padding=1)
        self.conv2 = nn.Conv2d(16,32,3,padding=1)
        self.conv2_2 = nn.Conv2d(32,64,3,padding=1)
        self.conv3 = nn.Conv2d(64,128,3,padding=1)
        self.conv3_2 = nn.Conv2d(128,200,3,padding=1)
        self.conv4 = nn.Conv2d(200,256,3,padding=1)
        self.conv5 = nn.Conv2d(256,512,3,padding=1)

        self.batch1 = nn.BatchNorm2d(16)
        self.batch2 = nn.BatchNorm2d(32)
        self.batch2_2 = nn.BatchNorm2d(64)
        self.batch3 = nn.BatchNorm2d(128)
        self.batch3_2 = nn.BatchNorm2d(200)
        self.batch4 = nn.BatchNorm2d(256)
        self.batch5 = nn.BatchNorm2d(512)

        self.pool = nn.MaxPool2d(2,2)
        self.dropout = nn.Dropout(0.2)
        self.fc1 = nn.Linear(512*7*7, 12240)
        self.fc2 = nn.Linear(12240, 8000)
        self.fc3 = nn.Linear(8000, 512)
        self.fc4 = nn.Linear(512, 133)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.batch1(self.conv1(x)))) # 112*112
        x = F.relu(self.batch2(self.conv2(x))) # 56*56
        x = self.pool(F.relu(self.batch2_2(self.conv2_2(x)))) # 56*56
        x = F.relu(self.batch3(self.conv3(x))) # 28*28
        x = self.pool(F.relu(self.batch3_2(self.conv3_2(x))))
        x = self.pool(F.relu(self.batch4(self.conv4(x)))) # 14*14
        x = self.pool(F.relu(self.batch5(self.conv5(x)))) # 7*7
        x = x.view(-1,7*7*512)
        x = self.dropout(x)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.fc3(x)
        x = F.relu(x)
```

```
        x = self.dropout(x)
        x = self.fc4(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:

1) Convolutional layers . I use 7 of this type of layer because if use a lot of convolutional layers you can found more complex figures. After each layer I use a batch normalization this with the intention of accelerate the learning of the network, and the activation function that i used was RELU.
2) Between some of the convolutional layers I used a Maxpooling layers to reduce the size of the inputs images
3) After the convolutional layers I used 4 full connected layers with a dropout off 20 % to avoid overfiting. And also used a RELU as a activation function

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 [3]: import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

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

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

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


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

    ## TODO: save the model if validation loss has decreased

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

```
                print('Validation loss decreased ({:.6f} --> {:.6f}).   Saving model...'.fo
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))

    model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')
    if use_cuda:
        model_scratch.cuda()

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

```
Epoch: 1         Training Loss: 4.637753         Validation Loss: 4.427267
Validation loss decreased (inf --> 4.427267).    Saving model...
Epoch: 2         Training Loss: 4.522554         Validation Loss: 4.364124
Validation loss decreased (4.427267 --> 4.364124).    Saving model...
Epoch: 3         Training Loss: 4.459027         Validation Loss: 4.251072
Validation loss decreased (4.364124 --> 4.251072).    Saving model...
Epoch: 4         Training Loss: 4.384823         Validation Loss: 4.168324
Validation loss decreased (4.251072 --> 4.168324).    Saving model...
Epoch: 5         Training Loss: 4.299453         Validation Loss: 4.069885
Validation loss decreased (4.168324 --> 4.069885).    Saving model...
Epoch: 6         Training Loss: 4.244790         Validation Loss: 4.204494
Epoch: 7         Training Loss: 4.197648         Validation Loss: 4.135220
Epoch: 8         Training Loss: 4.118144         Validation Loss: 3.897003
Validation loss decreased (4.069885 --> 3.897003).    Saving model...
Epoch: 9         Training Loss: 4.077584         Validation Loss: 3.950442
Epoch: 10         Training Loss: 4.027678         Validation Loss: 3.758258
Validation loss decreased (3.897003 --> 3.758258).    Saving model...
Epoch: 11         Training Loss: 3.978852         Validation Loss: 3.842026
Epoch: 12         Training Loss: 3.935289         Validation Loss: 3.780651
Epoch: 13         Training Loss: 3.886152         Validation Loss: 3.713439
Validation loss decreased (3.758258 --> 3.713439).    Saving model...
Epoch: 14         Training Loss: 3.849075         Validation Loss: 3.550044
Validation loss decreased (3.713439 --> 3.550044).    Saving model...
Epoch: 15         Training Loss: 3.797618         Validation Loss: 3.709570
Epoch: 16         Training Loss: 3.741493         Validation Loss: 3.657865
Epoch: 17         Training Loss: 3.706374         Validation Loss: 3.487592
Validation loss decreased (3.550044 --> 3.487592).    Saving model...
Epoch: 18         Training Loss: 3.658343         Validation Loss: 3.416111
Validation loss decreased (3.487592 --> 3.416111).    Saving model...
Epoch: 19         Training Loss: 3.621717         Validation Loss: 3.385428
```

```
Validation loss decreased (3.416111 --> 3.385428).    Saving model...
Epoch: 20         Training Loss: 3.575951         Validation Loss: 3.718173
```

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

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

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

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

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

        # call test function
        model_scratch.load_state_dict(torch.load('model_scratch.pt'))
        test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.385428


Test Accuracy: 17% (144/836)
```

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 [5]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
from PIL import ImageFile
import torchvision.transforms as transforms
import torch
import torchvision.models as models
import numpy as np

use_cuda = torch.cuda.is_available()

ImageFile.LOAD_TRUNCATED_IMAGES = True

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

data_dir = '/data/dog_images'


batch_size = 20

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                            std=[0.229, 0.224, 0.225])]


test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                           std=[0.229, 0.224, 0.225])]
```

```
valid_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                           std=[0.229, 0.224, 0.225])]



train_data = datasets.ImageFolder(data_dir + '/train', transform=train_transforms)
test_data = datasets.ImageFolder(data_dir + '/test', transform=test_transforms)
valid_data = datasets.ImageFolder(data_dir + '/valid', transform=valid_transforms)

trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=Tru
testloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)
validloader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)

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

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 [6]: 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.required_grad = False

        n_inputs = model_transfer.classifier[6].in_features

        layer = nn.Linear(n_inputs, 133)

        model_transfer.classifier[6] = layer

        print(model_transfer)

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:20<00:00, 26489481.67it/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=133, bias=True)
  )
)
```

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

Answer: I used the model vgg16 and only modify the classifier part for a full connected layer with a input of 25088 and output equal to 133, that is the number of categories to predict.The reason for use this is because this model is the only one I had used before and gave me good results.

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 [7]: import torch.optim as optim

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

```
In [5]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            # initialize tracker for minimum validation loss
            valid_loss_min = np.Inf

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

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


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

            ## TODO: save the model if validation loss has decreased

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

        # return trained model
        return model
```

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 [16]: # train the model
         model_transfer =  train(5, loaders_transfer, model_transfer, optimizer_transfer, criter

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

```
Epoch: 1         Training Loss: 4.283535         Validation Loss: 2.869763
Validation loss decreased (inf --> 2.869763).    Saving model...
Epoch: 2         Training Loss: 2.800040         Validation Loss: 1.312419
Validation loss decreased (2.869763 --> 1.312419).    Saving model...
Epoch: 3         Training Loss: 1.926222         Validation Loss: 0.825291
Validation loss decreased (1.312419 --> 0.825291).    Saving model...
Epoch: 4         Training Loss: 1.554764         Validation Loss: 0.639709
Validation loss decreased (0.825291 --> 0.639709).    Saving model...
Epoch: 5         Training Loss: 1.393747         Validation Loss: 0.590347
Validation loss decreased (0.639709 --> 0.590347).    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 [6]: 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 [18]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.590347


Test Accuracy: 82% (688/836)
```

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

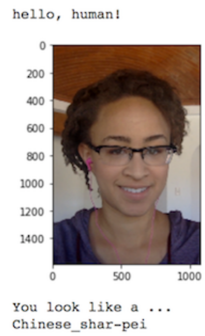
```
# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]
use_cuda = torch.cuda.is_available()

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    model_transfer.load_state_dict(torch.load('model_transfer.pt'))
    img = Image.open(img_path)


    test_transforms = transforms.Compose([transforms.Resize(255),
                                          transforms.CenterCrop(224),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])

    transformed = test_transforms(img).float()

    images = transformed.unsqueeze(0)


    if use_cuda:
        images = images.cuda()
    output = model_transfer(images)

    _, 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]
```

In [ ]:
```
dataiter = iter(loaders_transfer['test'])
images, labels = dataiter.next()
```

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!



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [99]: ### 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
             image = Image.open(img_path)
             if face_detector(img_path):
                 print ("Human")
                 plt.imshow(image)
                 plt.show()
                 pred =  predict_breed_transfer(img_path)
                 print("You look like : " + pred)
             elif dog_detector(img_path):
                 print ("Dog")
                 plt.imshow(image)
                 plt.show()
                 pred =  predict_breed_transfer(img_path)
                 print("Breed : " + pred)
             else:
                 print("error")
                 plt.imshow(image)
                 plt.show()

             print()
             print()
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

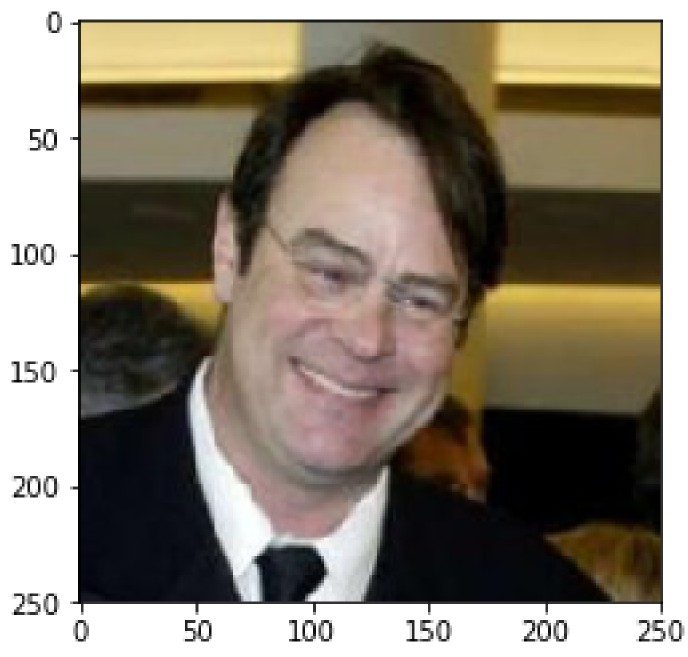
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

Answer: (Three possible points for improvement) 1 Increase the epochs of training to reduse the loss 2 Try another model and compare if it has better performance 3 Also I think the performace can improve witha largest dataset for training

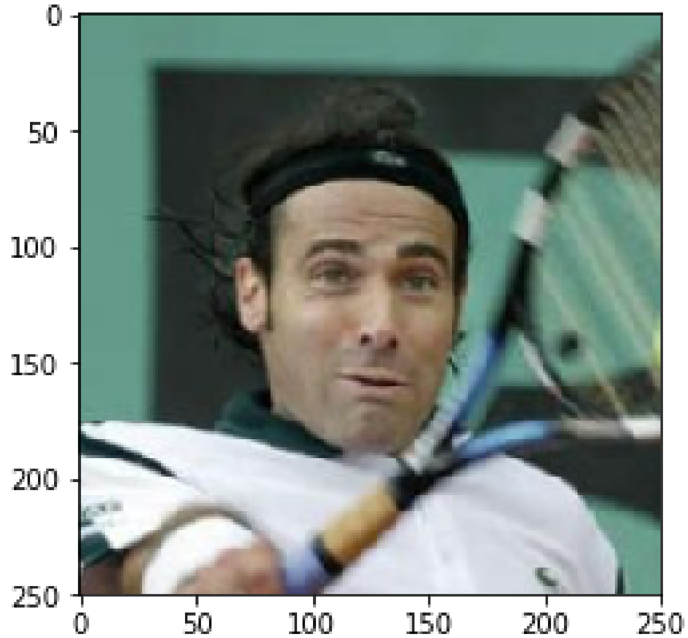
```
In [100]: ## TODO: Execute your algorithm from Step 6 on
          ## at least 6 images on your computer.
          ## Feel free to use as many code cells as needed.

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

```
Human
```



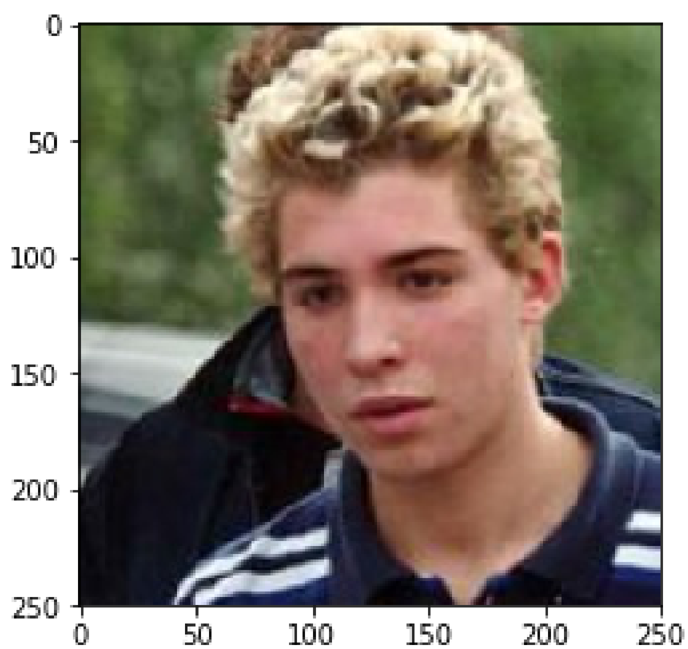

```
You look like : Lowchen
```

```
Human
```



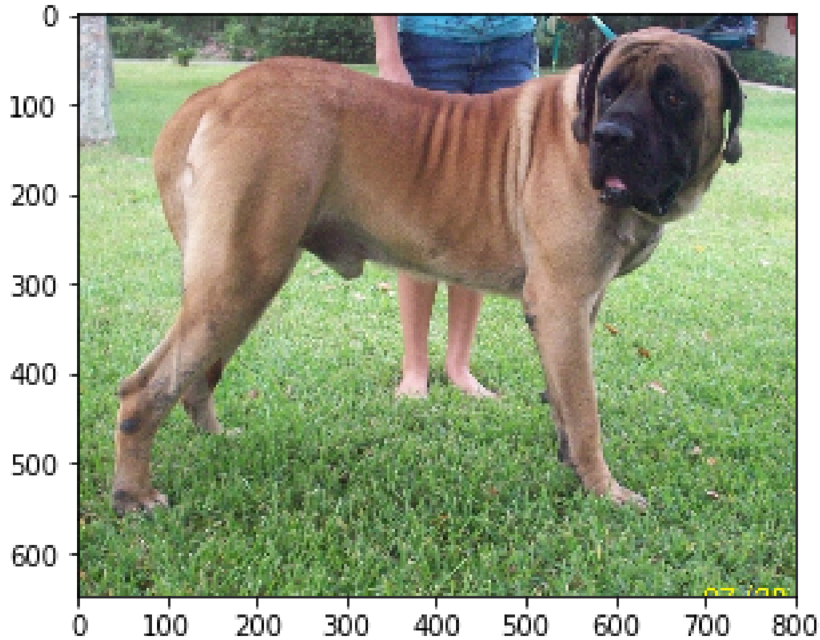

```
You look like : Cavalier king charles spaniel
```

```
Human
```



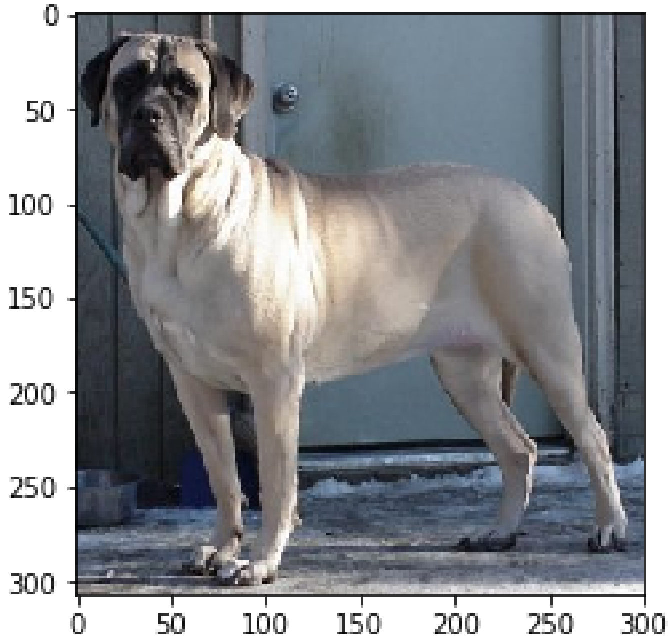
```
You look like : English cocker spaniel
```

```
Dog
```



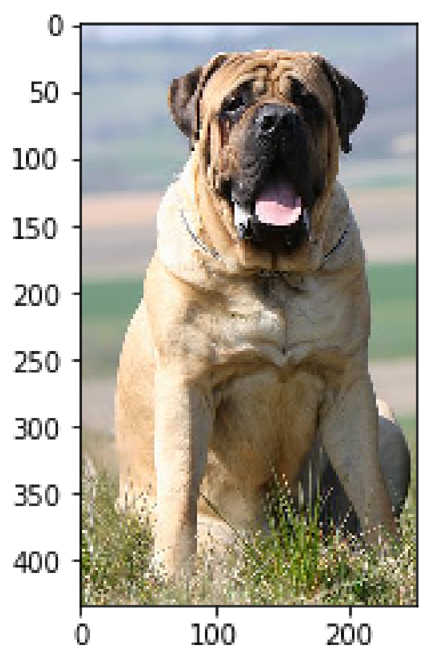

```
Breed : Bullmastiff
```

```
Dog
```




```
Breed : Mastiff
```

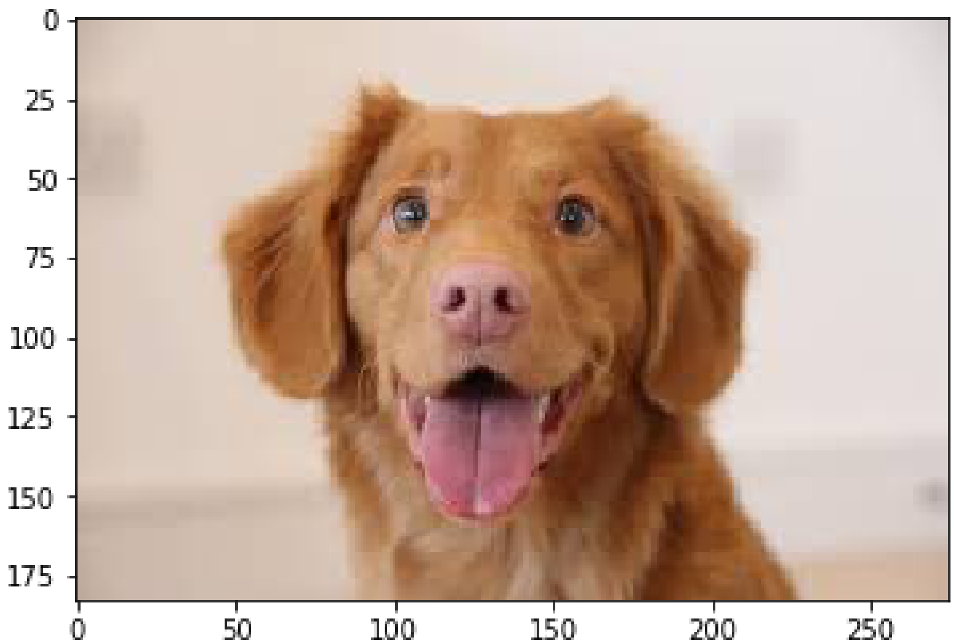
```
Dog
```



```
Breed : Bullmastiff
```

```
In [110]: run_app("imagen2.jpg")
```

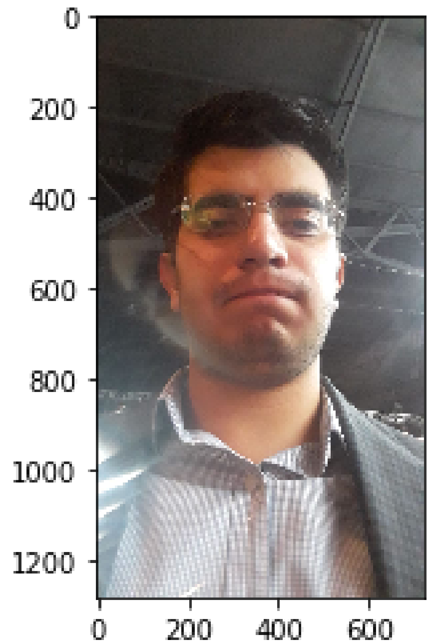
```
Dog
```




```
Breed : Nova scotia duck tolling retriever
```

```
In [113]: run_app("imagen5.jpg")
```

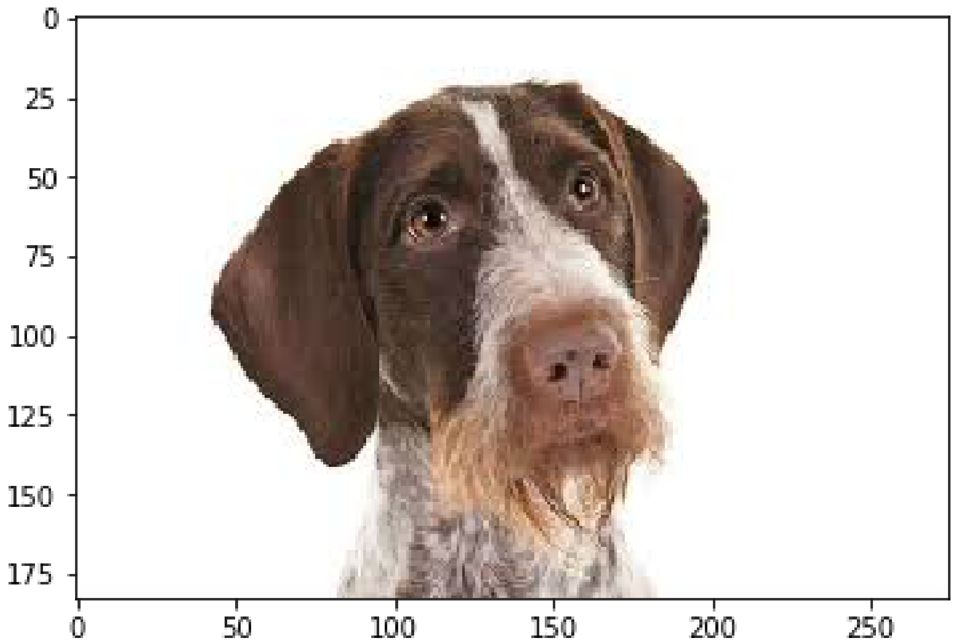
```
Human
```




```
You look like : Afghan hound
```

```
In [114]: run_app("img1-1.jpg")
```

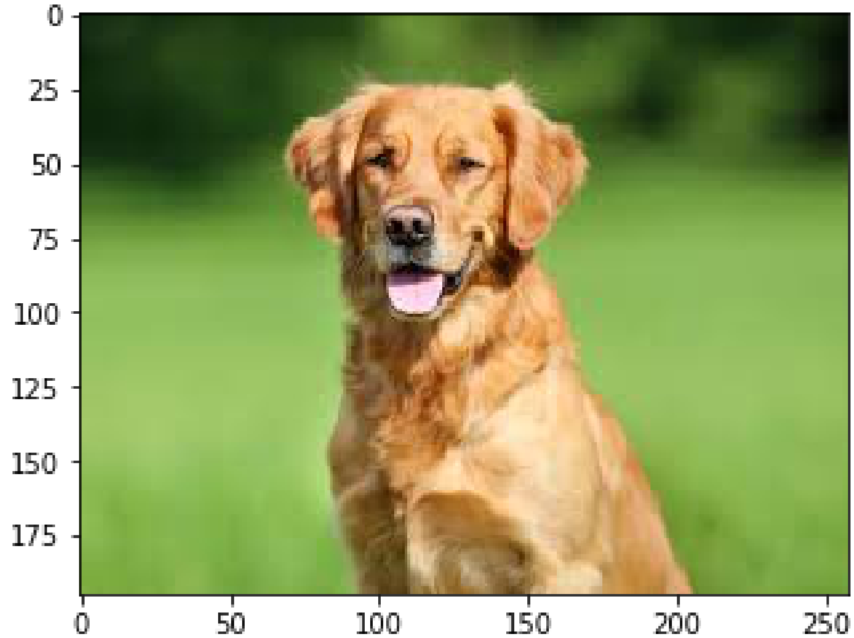
```
Dog
```




```
Breed : German wirehaired pointer
```

```
In [115]: run_app("img1-2.jpg")
```

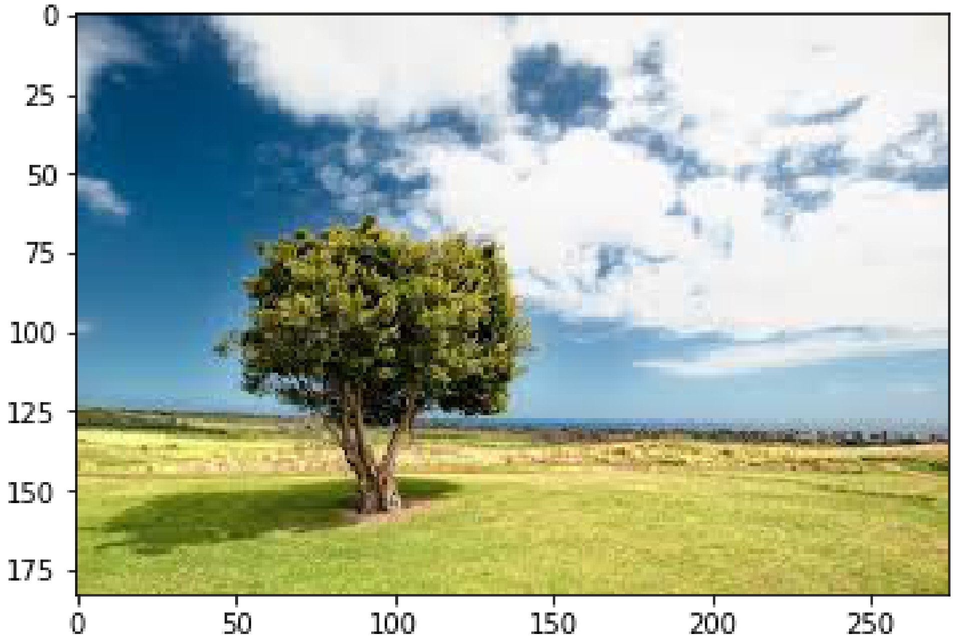
```
Dog
```




```
Breed : Golden retriever
```

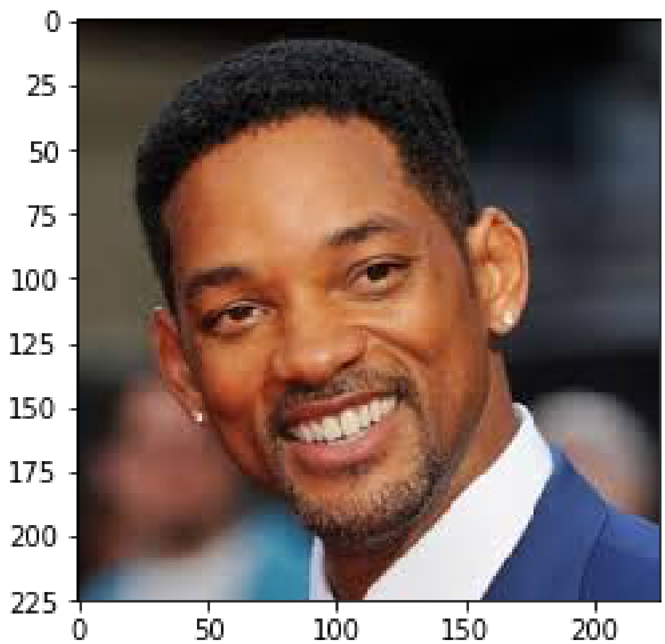
```
In [116]: run_app("img1-3.jpg")
```

```
error
```



```
In [117]: run_app("img1-4.jpg")
```

Human



```
You look like : Brittany
```

```
In [ ]:
```