dog_app

May 5, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [43]: 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

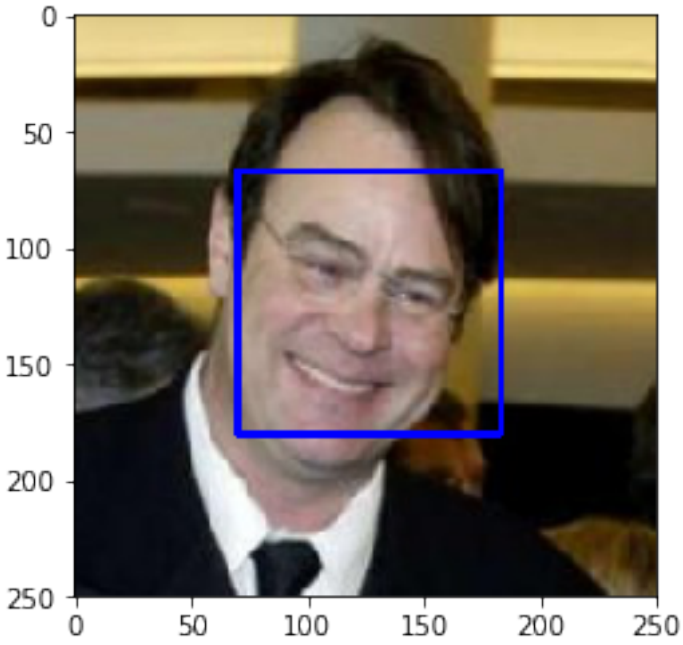
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [44]: 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 [45]: # 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: - percentage of the first 100 images in `human_files` have a detected human face = 98% - percentage of the first 100 images in `dog_files` have a detected human face = 17%

```
In [46]: 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.
         from tqdm import tqdm
         import time
         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_in_human_files = 0
         human_face_in_dog_files = 0
```

```
        with tqdm(total = len(human_files_short) ,desc ="human_files_short") as progress_bar:
            for human_file in human_files_short:
                if face_detector(human_file) == True :
                    human_face_in_human_files += 1
                    progress_bar.update(1)
        percentage_human_face_in_human_files = human_face_in_human_files
        print(' percentage of the first 100 images in human_files have a detected human face =

        time.sleep(3)
        with tqdm(total = len(dog_files_short) ,desc ="dog_files_short ") as progress_bar:
            for dog_file in dog_files_short:
                if face_detector(dog_file) == True :
                    human_face_in_dog_files += 1
                    progress_bar.update(1)
        percentage_human_face_in_dog_files = human_face_in_dog_files
        print(' percentage of the first 100 images in dog_files have a detected human face = '+
```

```
human_files_short:  98%|| 98/100 [00:02<00:00, 35.28it/s]


 percentage of the first 100 images in human_files have a detected human face = 98%


dog_files_short :  17%|        | 17/100 [00:28<01:28,  1.07s/it]

 percentage of the first 100 images in dog_files have a detected human face = 17%
```

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

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 [47]: 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 [48]: from PIL import Image
         import torchvision.transforms as transforms

         # Set PIL to be tolerant of image files that are truncated.
         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).convert('RGB')
    #From pytorch Forums :https://discuss.pytorch.org/t/transfer-learning-usage-with-di
    #VGG 16 takes 224x224 images as input
    preprocessed_image = transforms.Compose([
                                transforms.RandomResizedCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485 , 0.456 ,0.406),
                                                     (0.229 , 0.224 , 0.225))])
    #with the help of these :
    #https://gist.github.com/jkarimi91/d393688c4d4cdb9251e3f939f138876e
    #and lectures from udacity

    # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
    # in other meaning we are preprcess image to 4D tensor where unsqueeze zero adds a
    image = preprocessed_image(image)[:3,:,:].unsqueeze(0)
    if use_cuda:
        image = image.cuda()

    output = VGG16(image)
    predicted_index = torch.argmax(output).item()

    return predicted_index # predicted class index
```

In [49]:
```
#testing the function
VGG16_predict(dog_files_short[1])
```

Out[49]: 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 [50]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    index = VGG16_predict(img_path)
    if index > 150 and index < 269:
        return True
    else :
        return False
    #return None # true/false
```

```
In [51]: #print it out to check
         print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))
```

```
True
False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: . - percentage of the first 100 images in `human_files` have a detected a dog = 1% - percentage of the first 100 images in `dog_files` have a detected a dog = 96%

```
In [52]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.


         human_face_in_human_files = 0
         human_face_in_dog_files = 0
         with tqdm(total = len(human_files_short) ,desc ="human_files_short") as progress_bar:
             for human_file in human_files_short:
                 if dog_detector(human_file) == True :
                     human_face_in_human_files += 1
                     progress_bar.update(1)
         percentage_human_face_in_human_files = human_face_in_human_files
         print('percentage of the images in human_files_short have a detected dog = '+ str(perce

         time.sleep(3)
         with tqdm(total = len(dog_files_short) ,desc ="dog_files_short ") as progress_bar:
             for dog_file in dog_files_short:
                 if dog_detector(dog_file) == True :
                     human_face_in_dog_files += 1
                     progress_bar.update(1)
         percentage_human_face_in_dog_files = human_face_in_dog_files
         print('percentage of the images in dog_files_short have a detected dog = '+ str(percent
```

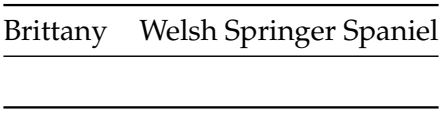
```
human_files_short:   1%|          | 1/100 [00:00<00:59,  1.66it/s]


percentage of the images in human_files_short have a detected dog = 1%


dog_files_short :  96%|| 96/100 [00:04<00:00, 20.58it/s]

percentage of the images in dog_files_short have a detected dog = 96%
```

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`.

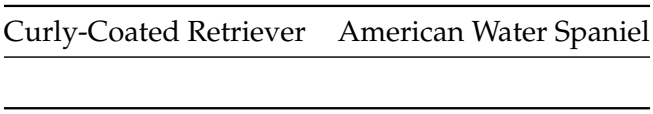
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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.

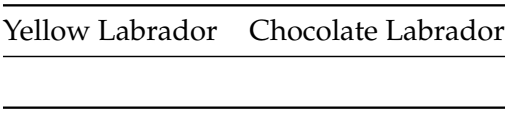
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.



It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).



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.



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 [10]: import os
         from torchvision import datasets

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes3
         #number of subprocesss to use for data loading
         num_workers = 0
         #how many samples per batch to load
         batch_size = 20
         #Data Augmentation
         transform = transforms.Compose([
             transforms.RandomResizedCrop(224),
             transforms.RandomHorizontalFlip(),
             transforms.RandomRotation(10),
             transforms.ToTensor(),
             transforms.Normalize((0.485 , 0.456 ,0.406),(0.229 , 0.224 , 0.225))])

         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir,'train/')
         valid_dir = os.path.join(data_dir,'valid/')
         test_dir = os.path.join(data_dir,'test/')
         train_data = datasets.ImageFolder(train_dir,transform=transform)
         valid_data = datasets.ImageFolder(valid_dir,transform=transform)
         test_data = datasets.ImageFolder(test_dir,transform=transform)


         #prepare data loaders
         train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,shuffle=Tru
         valid_loader = torch.utils.data.DataLoader(valid_data,batch_size=batch_size,shuffle=Tru
         test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size,shuffle=True,
```

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: - So i have cropped the images and resized it to be 224 x224 as we might use it in the future with VGG16 where in the paper,the input should be 224 x224 .i have also chosen a random Horizontal Flip and random rotation bt 10 degrees.

- Yes i have used data augmentation as i have stated before as data augmentaion as it enhances and improves the quality of training data set so that we can build deep learning models in such a good way.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [11]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 #convolutional layer (224x224x3)
                 self.conv1 = nn.Conv2d(3 , 64 ,3 ,padding = 1)
                 #convolutional layer (112x112x64)
                 self.conv2 = nn.Conv2d(64,128,3,padding=1)
                 #convolutional layer (56x56x128)
                 self.conv3 = nn.Conv2d(128,256,3,padding=1)
                 #convolutional layer (28x28x256)
                 self.conv4 = nn.Conv2d(256,512,3,padding=1)
                 #max pooling layer
                 self.pool = nn.MaxPool2d(2,2)
                 #linear layer (512 * 14 * 14 ->1024)
                 self.fc1 = nn.Linear(512 *14 * 14 ,1024)
                 #linear layer (1024 - > 133)
                 self.fc2 = nn.Linear(1024,133)
                 #drop out
                 self.dropout = nn.Dropout(0.20)

             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 = x.view(x.shape[0],-1)
                 x = self.dropout(x)
                 x = F.relu(self.fc1(x))
                 x = self.dropout(x)
                 x = self.fc2(x)
                 return x

         #-#-# You do 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, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=100352, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

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

Answer: - My plan was to decrease the dimenstions x,y as we go further and increase the depth of the layers and this to increase the efficiency . - i have also used drop out function of 20 % to avoid overfitting. - i have used a a max pooling of kernel 2 and stride 2 after each conv. layer to decrease the dimensions by the half which means we will devide by 2 , check the comments between layers on the above cells for more clarification . - I have used two fully connected layer with relu activation function after the first one and the second on will detect the classes of the breeds. - layers : - conv1 -- > (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) - (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - conv2 -- > (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) - (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - conv3 -- > (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) - (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - conv4 -- > (conv4): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) - (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) - (dropout): Dropout(p=0.2, inplace=False) - Two fully connected layer : - fc 1 -- > (fc1): Linear(in_features=131072, out_features=1024, bias=True) - (dropout): Dropout(p=0.2, inplace=False) - fc 2 -- > (fc2): Linear(in_features=1024, out_features=133, bias=True)

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 [12]: import torch.optim as optim

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

```
### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.02)
```

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 [13]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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(train_loader):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            #first we clear the gradients of all optimized variables
            optimizer.zero_grad()
            #forward pass : pass input to model to get output
            output = model(data)
            #calculating batch loss
            loss = criterion(output,target)
            #backward pass : loss of gradient computation
            loss.backward()
            #update parameters by aingle step optimization
            optimizer.step()
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
```

```
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(valid_loader):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()


            #forward pass : pass input to model to get output
            output = model(data)
            #calculating batch loss
            loss = criterion(output,target)
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
            valid_loss += loss.item()*data.size(0)
        ## update the average validation loss
        train_loss = train_loss/len(train_loader.dataset)
        valid_loss = valid_loss/len(valid_loader.dataset)


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

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

loaders_scratch = {'train' : train_loader,
                   'valid' : valid_loader,
                   'test' : test_loader
                     }

# train the model
model_scratch = train(30, 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: 4.877576         Validation Loss: 4.851457
Validation loss decreased (inf --> 4.851457).  Saving model ...
Epoch: 2         Training Loss: 4.807053         Validation Loss: 4.722278
Validation loss decreased (4.851457 --> 4.722278).  Saving model ...
Epoch: 3         Training Loss: 4.647418         Validation Loss: 4.619562
Validation loss decreased (4.722278 --> 4.619562).  Saving model ...
Epoch: 4         Training Loss: 4.568357         Validation Loss: 4.579813
Validation loss decreased (4.619562 --> 4.579813).  Saving model ...
Epoch: 5         Training Loss: 4.512142         Validation Loss: 4.503259
Validation loss decreased (4.579813 --> 4.503259).  Saving model ...
Epoch: 6         Training Loss: 4.465545         Validation Loss: 4.439723
Validation loss decreased (4.503259 --> 4.439723).  Saving model ...
Epoch: 7         Training Loss: 4.422688         Validation Loss: 4.417101
Validation loss decreased (4.439723 --> 4.417101).  Saving model ...
Epoch: 8         Training Loss: 4.361989         Validation Loss: 4.388283
Validation loss decreased (4.417101 --> 4.388283).  Saving model ...
Epoch: 9         Training Loss: 4.296399         Validation Loss: 4.318071
Validation loss decreased (4.388283 --> 4.318071).  Saving model ...
Epoch: 10         Training Loss: 4.248483         Validation Loss: 4.304266
Validation loss decreased (4.318071 --> 4.304266).  Saving model ...
Epoch: 11         Training Loss: 4.183809         Validation Loss: 4.365463
Epoch: 12         Training Loss: 4.139348         Validation Loss: 4.173928
Validation loss decreased (4.304266 --> 4.173928).  Saving model ...
Epoch: 13         Training Loss: 4.059425         Validation Loss: 4.101349
Validation loss decreased (4.173928 --> 4.101349).  Saving model ...
Epoch: 14         Training Loss: 4.005668         Validation Loss: 4.167177
Epoch: 15         Training Loss: 3.967786         Validation Loss: 4.188379
Epoch: 16         Training Loss: 3.906323         Validation Loss: 4.093761
Validation loss decreased (4.101349 --> 4.093761).  Saving model ...
Epoch: 17         Training Loss: 3.854992         Validation Loss: 4.132159
Epoch: 18         Training Loss: 3.824692         Validation Loss: 3.971330
Validation loss decreased (4.093761 --> 3.971330).  Saving model ...
Epoch: 19         Training Loss: 3.776379         Validation Loss: 4.081574
Epoch: 20         Training Loss: 3.706124         Validation Loss: 4.089470
Epoch: 21         Training Loss: 3.688903         Validation Loss: 3.954123
Validation loss decreased (3.971330 --> 3.954123).  Saving model ...
Epoch: 22         Training Loss: 3.656650         Validation Loss: 3.933336
Validation loss decreased (3.954123 --> 3.933336).  Saving model ...
Epoch: 23         Training Loss: 3.588884         Validation Loss: 3.937059
Epoch: 24         Training Loss: 3.546192         Validation Loss: 3.969206
Epoch: 25         Training Loss: 3.476315         Validation Loss: 3.978433
Epoch: 26         Training Loss: 3.445716         Validation Loss: 3.896812
Validation loss decreased (3.933336 --> 3.896812).  Saving model ...
Epoch: 27         Training Loss: 3.393648         Validation Loss: 3.867171
Validation loss decreased (3.896812 --> 3.867171).  Saving model ...
Epoch: 28         Training Loss: 3.319989         Validation Loss: 3.866821
Validation loss decreased (3.867171 --> 3.866821).  Saving model ...
Epoch: 29         Training Loss: 3.289856         Validation Loss: 3.779572
```

```
Validation loss decreased (3.866821 --> 3.779572).  Saving model ...
Epoch: 30         Training Loss: 3.251469         Validation Loss: 3.797541
```

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 [14]:
```
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(test_loader):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

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

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

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

```
Test Loss: 3.781564


Test Accuracy: 13% (109/836)
```

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 [12]:
```
## TODO: Specify data loaders
import os
from torchvision import datasets
num_workers = 0
#how many samples per batch to load
batch_size = 20
#Data Augmentation
transform = transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.ToTensor()])

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
train_data = datasets.ImageFolder(train_dir,transform=transform)
valid_data = datasets.ImageFolder(valid_dir,transform=transform)
test_data = datasets.ImageFolder(test_dir,transform=transform)


#prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data,batch_size=batch_size,shuffle=Tru
valid_loader = torch.utils.data.DataLoader(valid_data,batch_size=batch_size,shuffle=Tru
test_loader = torch.utils.data.DataLoader(test_data,batch_size=batch_size,shuffle=True,
```

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 [13]:
```
import torchvision.models as models
import torch.nn as nn


#loading the vgg16 pretrained model from pytorch
model_transfer= VGG16
print(model_transfer)
```

```
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)
  )
)
```

```
In [14]: #freezing training for all "features " layers
         for param in model_transfer.features.parameters():
```

```
            param.requires_grad = False

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

In [15]:
```
        ## TODO: Specify model architecture
        #change the last layer that maps n_inputs to 133
        #note for myself: new layers automatically have requires_grad = True

        n_inputs =model_transfer.classifier[6].in_features
        last_layer = nn.Linear(n_inputs,133)
        model_transfer.classifier[6] = last_layer

        if use_cuda:
            model_transfer = model_transfer.cuda()

        print (model_transfer.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 have used VGG16 as it has won second place in ImageNet and due to its graeat performance as well as its simplicity . - VGG16 has trained on ImageNet 16 which already has Dogs Images - so basically what i did : - i left VGG16 features as it is and i mean by that that i have freezed the parameters of all convolutional layers - i have only changed the final classification layer to 4094 in_features and 133 out features which is number of classes we have .

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 [16]:
```
        import torch.optim as optim
        criterion_transfer = nn.CrossEntropyLoss()
        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 [21]:
```
# the following import is required for training to be robust to truncated images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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(train_loader):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            #first we clear the gradients of all optimized variables
            optimizer.zero_grad()
            #forward pass : pass input to model to get output
            output = model(data)
            #calculating batch loss
            loss = criterion(output,target)
            #backward pass : loss of gradient computation
            loss.backward()
            #update parameters by aingle step optimization
            optimizer.step()
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(valid_loader):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
```

```
                #forward pass : pass input to model to get output
                output = model(data)
                #calculating batch loss
                loss = criterion(output,target)
                #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
                valid_loss += loss.item()*data.size(0)
            ## update the average validation loss
            train_loss = train_loss/len(train_loader.dataset)
            valid_loss = valid_loss/len(valid_loader.dataset)


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

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

    loaders_transfer = {'train' : train_loader,
                        'valid' : valid_loader,
                        'test' : test_loader
                          }
```

```
In [22]: # train the model
         model_transfer =train(10, loaders_transfer, model_transfer, optimizer_transfer, criteri
```

```
Epoch: 1         Training Loss: 1.215184         Validation Loss: 1.371189
Validation loss decreased (inf --> 1.371189).  Saving model ...
Epoch: 2         Training Loss: 1.165129         Validation Loss: 1.434311
Epoch: 3         Training Loss: 1.150237         Validation Loss: 1.406801
Epoch: 4         Training Loss: 1.150353         Validation Loss: 1.377579
Epoch: 5         Training Loss: 1.097739         Validation Loss: 1.398407
Epoch: 6         Training Loss: 1.047687         Validation Loss: 1.331802
Validation loss decreased (1.371189 --> 1.331802).  Saving model ...
Epoch: 7         Training Loss: 1.056986         Validation Loss: 1.420028
Epoch: 8         Training Loss: 1.033496         Validation Loss: 1.381143
Epoch: 9         Training Loss: 1.007943         Validation Loss: 1.401107
Epoch: 10         Training Loss: 0.984884         Validation Loss: 1.289155
```

```
Validation loss decreased (1.331802 --> 1.289155).  Saving model ...
```

```
In [17]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

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

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

             print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
                 100. * correct / total, correct, total))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.319639


Test Accuracy: 64% (543/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

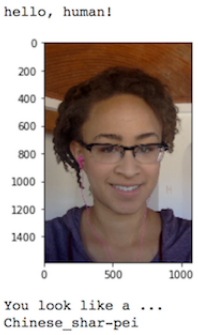
Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

In [18]:
```
### 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]
print(class_names[0])
def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    image = Image.open(img_path).convert('RGB')
    #From pytorch Forums :https://discuss.pytorch.org/t/transfer-learning-usage-with-di
    #VGG 16 takes 224x224 images as input
    preprocessed_image = transforms.Compose([
                                transforms.RandomResizedCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485 , 0.456 ,0.406),
                                                     (0.229 , 0.224 , 0.225))])
    #with the help of these :
    #https://gist.github.com/jkarimi91/d393688c4d4cdb9251e3f939f138876e
    #and lectures from udacity

    # discard the transparent, alpha channel (that's the :3) and add the batch dimensio
    # in other meaning we are preprcess image to 4D tensor where unsqueeze zero adds a
    image = preprocessed_image(image)[:3,:,:].unsqueeze(0)
    if use_cuda:
        image = image.cuda()
    output = model_transfer(image)
    # convert output probabilities to predicted class
    predicted_index = torch.argmax(output).item()
    dog_type = class_names [predicted_index]
    return dog_type
```

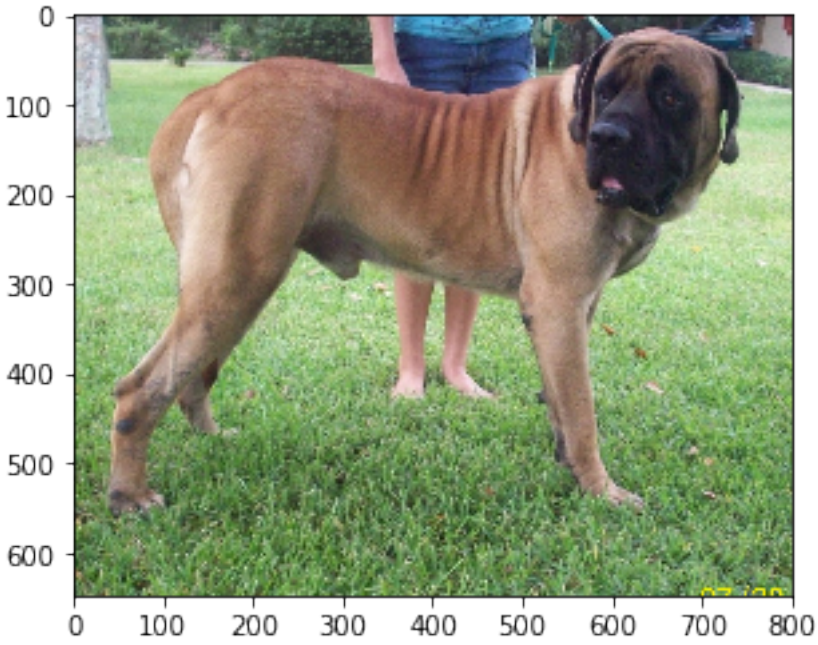
```
Affenpinscher
```

In [19]:
```
#testing my function
image = Image.open(dog_files[0])
print(predict_breed_transfer(dog_files[0]))
plt.imshow(image)
plt.show()
```

```
Belgian malinois
```



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

         def run_app(img_path):
             image = Image.open(img_path)
             ## handle cases for a human face, dog, and neither
             if dog_detector(img_path) == True:
                 print ('It is a dog!')
                 plt.imshow(image)
                 plt.show()
                 print('it looks like a ...')
                 print(predict_breed_transfer(img_path))
                 print('\n----------------------------------\n')
             elif face_detector(img_path) == True :
                 print ('Hello Human! ')
                 plt.imshow(image)
                 plt.show()
                 print('you look like a ...')
                 print(predict_breed_transfer(img_path))
                 print ('\n---------------------\n')
             else:
                 print ('Error ! couldnot detect a human or a dog ')
                 plt.imshow(image)
                 plt.show()
                 print ('\n---------------------\n')
```

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.

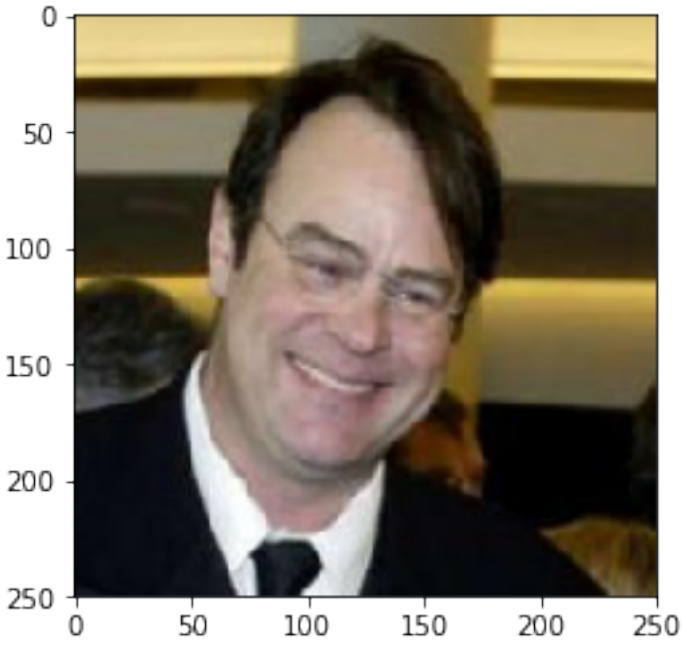
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) - I would say worse " - But may be i should have tried to change a bit more in the classification layers ,drop outs - try out different optimizers like Adam for example ,try out different learning rates values - add more variety to dataset and by that i mean ;trying out different data augmentations - increase number of epochs

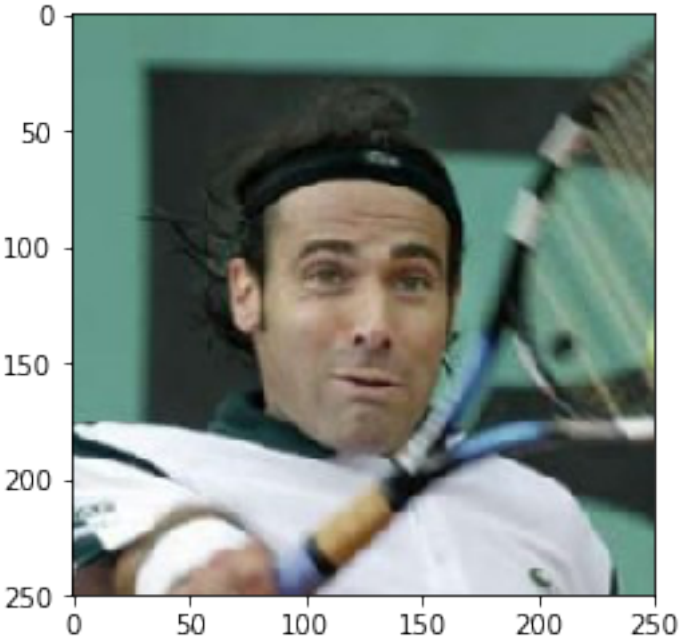
```
In [53]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
```

```
        ## Feel free to use as many code cells as needed.

        ## suggested code, below
        for file in np.hstack((human_files[:3], dog_files[6:9])):
            run_app(file)
```

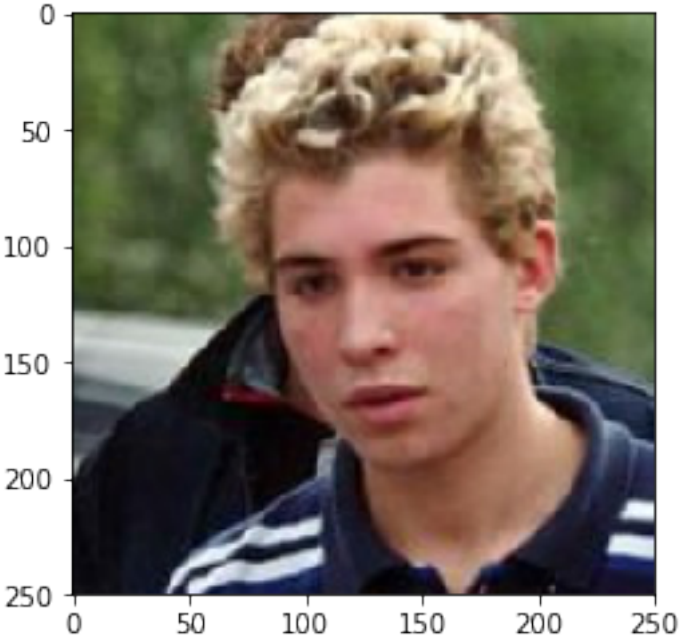
```
Hello Human!
```



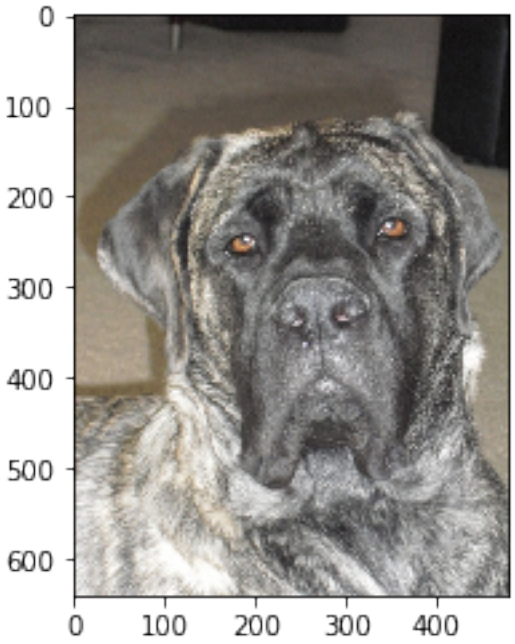

```
you look like a ...
Dachshund

----------------------

Hello Human!
```




```
you look like a ...
Beagle

----------------------

Hello Human!
```

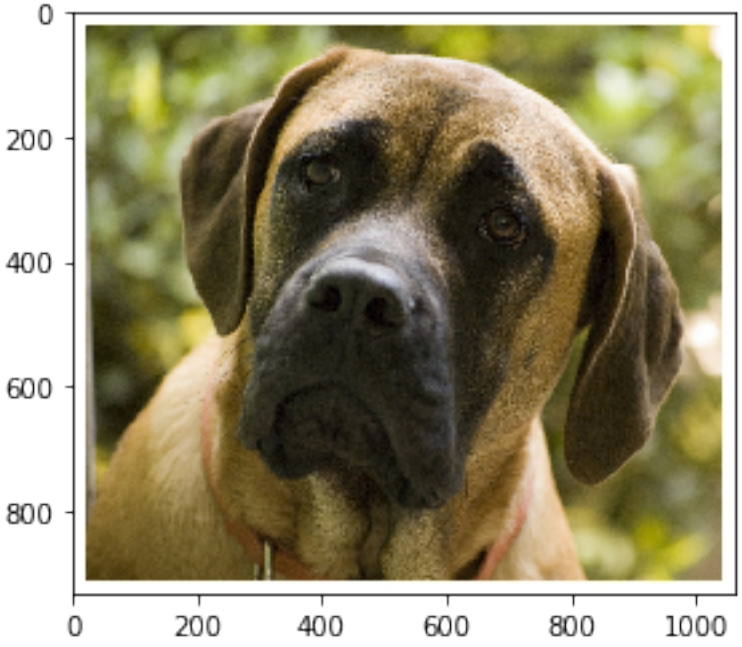


```
you look like a ...
Airedale terrier

----------------------

It is a dog!
```




```
it looks like a ...
Mastiff

-----------------------------------

It is a dog!
```


```
it looks like a ...
Mastiff

-----------------------------------

It is a dog!
```



```
it looks like a ...
Cane corso

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

In [ ]: