dog_app

May 14, 2021

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 [67]: 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

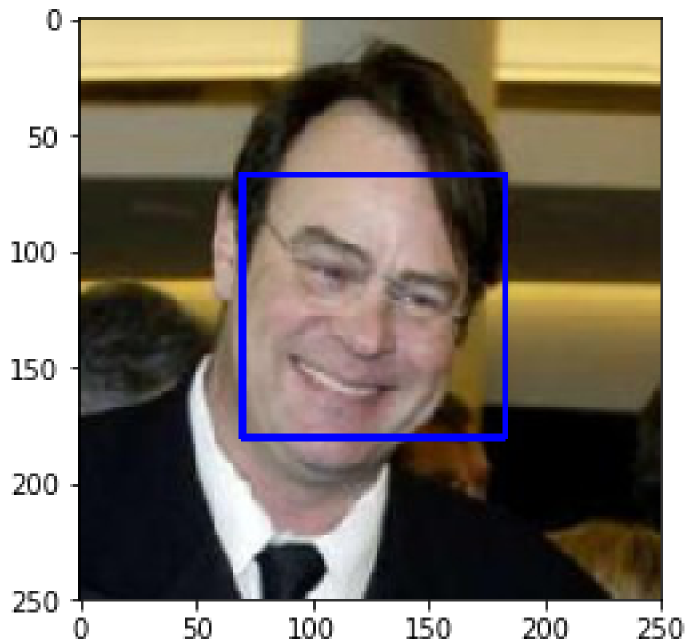
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [68]: 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 [70]:
```
# 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)
    if len(faces) > 0:
        return True
    return False
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

the accuracy of the algorithm in humen files 0.98
the accuracy of the algorithm in dog files 0.17

In [71]:
```
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.
counter_h=0
counter_d=0

for img in human_files_short:
    humen=face_detector(img)
    if humen :
        counter_h += 1


for img in dog_files_short :
    humen=face_detector(img)
    if not humen:
```

```
            counter_d += 1


    percent_in_humen_files=counter_h/len(human_files_short)

    percent_in_dog_files=1-(counter_d/len(dog_files_short))

    print("the accuracy of the algorithm in humen files {}".format(percent_in_humen_files))
    print("the accuracy of the algorithm in dog files {}".format(percent_in_dog_files))
```

```
the accuracy of the algorithm in humen files 0.98
the accuracy of the algorithm in dog files 0.17000000000000004
```

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 [72]: import torch
         import torchvision.models as models

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

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

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


         print(VGG16.classifier[0].in_features)
         VGG16
```

```
25088
```

```
Out[72]: VGG(
           (features): Sequential(
             (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (1): ReLU(inplace)
             (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (3): ReLU(inplace)
             (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (6): ReLU(inplace)
             (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (8): ReLU(inplace)
             (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (11): ReLU(inplace)
             (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (13): ReLU(inplace)
             (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (15): ReLU(inplace)
             (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (18): ReLU(inplace)
             (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (20): ReLU(inplace)
             (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (22): ReLU(inplace)
             (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
             (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (25): ReLU(inplace)
             (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (27): ReLU(inplace)
             (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (29): ReLU(inplace)
             (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           )
           (classifier): Sequential(
             (0): Linear(in_features=25088, out_features=4096, bias=True)
             (1): ReLU(inplace)
             (2): Dropout(p=0.5)
             (3): Linear(in_features=4096, out_features=4096, bias=True)
             (4): ReLU(inplace)
             (5): Dropout(p=0.5)
             (6): Linear(in_features=4096, out_features=1000, bias=True)
           )
         )
```

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

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

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

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

In [73]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image


    data_transform = transforms.Compose([ transforms.Resize((224,224)),transforms.ToTen
    img= Image.open(img_path)
    img_t=data_transform(img)
    batch_t = torch.unsqueeze(img_t, 0)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    VGG16.eval()
    output= VGG16(batch_t.to(device))
    output=output.to(device)
    _, preds_tensor = torch.max(output, 1)
    preds_tensor=preds_tensor.to(torch.device("cpu"))
    preds = np.squeeze(preds_tensor.numpy())
    return preds # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [74]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             output=VGG16_predict(img_path)
             if output>=151 and output<=268 :
                 return True
             else:
                 return False # 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:

```
In [75]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         count_dog=0
         for img in human_files_short :
             output= dog_detector(img)
             if output == True :
                 count_dog+=1
         print("the accuracy of the algorithm in humen files {}".format(count_dog/len(human_file

the accuracy of the algorithm in humen files 0.0
```

```
In [76]: count_dog=0
         for img in dog_files_short :
             output= dog_detector(img)
             if output == True :
                 count_dog+=1
         print("the accuracy of the algorithm in dog files {}".format((count_dog/len(dog_files_s

the accuracy of the algorithm in dog files 0.97
```

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 [77]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

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

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

In [78]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms

from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 16

# convert data to a normalized torch.FloatTensor

transform_train = transforms.Compose([
    transforms.RandomRotation(45),

  transforms.Resize((512,512)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

transform_valid = transforms.Compose([
  transforms.Resize((512,512)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

transform_test = transforms.Compose([
  transforms.Resize((512,512)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])


train_dataset = datasets.ImageFolder(root='/data/dog_images/train',
                                           transform=transform_train)

test_dataset = datasets.ImageFolder(root='/data/dog_images/test',
                                           transform=transform_test)

valid_dataset = datasets.ImageFolder(root='/data/dog_images/valid',
                                           transform=transform_valid)

train_dataloader=torch.utils.data.DataLoader(train_dataset,
```

```
                                                          batch_size=batch_size, shuffle=True,
                                                          num_workers=num_workers)

valid_dataloader=torch.utils.data.DataLoader(valid_dataset,
                                                          batch_size=batch_size,
                                                          num_workers=num_workers)

test_dataloader=torch.utils.data.DataLoader(test_dataset,
                                                          batch_size=batch_size, shuffle=True,
                                                          num_workers=num_workers)



loaders_scratch=[train_dataloader,valid_dataloader,test_dataloader]

# loaders_scratch = {"train" : torch.utils.data.DataLoader(train_dataset,
#                                                           batch_size=batch_size, shuffle=True,
#                                                           num_workers=num_workers)
#                    ,"valid" : torch.utils.data.DataLoader(valid_dataset,
#                                                           batch_size=batch_size,
#                                                           num_workers=num_workers)

#                    ,"test" : torch.utils.data.DataLoader(test_dataset,
#                                                           batch_size=4, shuffle=True,
#                                                           num_workers=num_workers)}
```

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:
The first question
I resize the images by stretching to 512. After researching and several attempts, I found that the best size is to be 512 by 512 pixels, but of course I can use a larger scale or less, but if I try to use a greater resolution, this will consume more than the RAM and you will need more time to learn, and if I try to reduce it more, I may reach a fun in which the relationships between Pixels, and you will not be able to identify the type of dog

--

second question
Yes, I used RandomRotation(Randomize values for the Rotation)

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [79]:
```python
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
        ## input is 512*512*3

        self.pool=nn.MaxPool2d(4,4)

        self.conv1_1=nn.Conv2d(3,16,3,padding=1)
        self.conv1_2=nn.Conv2d(16,16,3,padding=1)
        self.conv1_3=nn.Conv2d(16,16,3,padding=1)

        self.Batch_1=nn.BatchNorm2d(16)

        self.conv2_1=nn.Conv2d(16,32,3,padding=1)
        self.conv2_2=nn.Conv2d(32,32,3,padding=1)
        self.conv2_3=nn.Conv2d(32,32,3,padding=1)

        self.Batch_2=nn.BatchNorm2d(32)

        self.conv3_1=nn.Conv2d(32,64,3,padding=1)
        self.conv3_2=nn.Conv2d(64,64,3,padding=1)
        self.conv3_3=nn.Conv2d(64,64,3,padding=1)

        self.Batch_3=nn.BatchNorm2d(64)

        self.drop=nn.Dropout(p=0.5)
        ## output is 8*8*64
        self.fc1=nn.Linear(8*8*64,2500)
        self.fc2=nn.Linear(2500,1000)
        self.fc3=nn.Linear(1000,133)

        self.Batch_linear=nn.BatchNorm1d(2500)

    def forward(self, x):
        ## Define forward behavior

        x=F.relu(self.conv1_1(x))
        x=F.relu(self.conv1_2(x))
        x=F.relu(self.conv1_3(x))

        x=self.pool(x)
        x=self.Batch_1(x)
```

```
        x=F.relu(self.conv2_1(x))
        x=F.relu(self.conv2_2(x))
        x=F.relu(self.conv2_3(x))

        x=self.pool(x)
        x=self.Batch_2(x)

        x=F.relu(self.conv3_1(x))
        x=F.relu(self.conv3_2(x))
        x=F.relu(self.conv3_3(x))

        x=self.pool(x)
        x=self.Batch_3(x)

        x=x.view(-1,8*8*64)
        x=self.drop(F.relu(self.fc1(x)))
        x=self.Batch_linear(x)
        x=self.drop(F.relu(self.fc2(x)))
        x= self.fc3(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:

At first I looked for examples similar to what I wanted to do. At the beginning I applied three main layers, each one containing three convolutional layers the first one extracts raw information such as color and fonts. The second one discovers more complex information, such as the relationship of colors and shapes together, and the last one to identify the shapes of dogs to differentiate them .After each main layer I put 2d BatchNorm to improve the speed of learning and ensure that all data is on the same scal, then i used 3 linear layers for classification and 1d BatchNorm after the first layer to ensure that all data is on the same scal

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 [80]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.0001)
```

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 [81]: 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[0]):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     optimizer.zero_grad()

                     output = model(data)

                     loss =  criterion(output,target)

                     loss.backward()

                     optimizer.step()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     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[1]):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                output = model(data)
                loss = criterion(output,target)
                ## update the average validation loss
                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 :
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                torch.save(model.state_dict(), save_path)
                valid_loss_min=valid_loss
        return model
```

In [ ]:
```
# train the model
model_scratch = train(40, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.831641         Validation Loss: 4.663046
Validation loss decreased (inf --> 4.663046).  Saving model ...
Epoch: 2         Training Loss: 4.544766         Validation Loss: 4.406909
Validation loss decreased (4.663046 --> 4.406909).  Saving model ...
Epoch: 3         Training Loss: 4.343581         Validation Loss: 4.235242
Validation loss decreased (4.406909 --> 4.235242).  Saving model ...
Epoch: 4         Training Loss: 4.197054         Validation Loss: 4.150343
Validation loss decreased (4.235242 --> 4.150343).  Saving model ...
Epoch: 5         Training Loss: 4.072188         Validation Loss: 4.041636
Validation loss decreased (4.150343 --> 4.041636).  Saving model ...
Epoch: 6         Training Loss: 3.961784         Validation Loss: 3.872406
Validation loss decreased (4.041636 --> 3.872406).  Saving model ...
Epoch: 7         Training Loss: 3.845782         Validation Loss: 3.829316
Validation loss decreased (3.872406 --> 3.829316).  Saving model ...
Epoch: 8         Training Loss: 3.764273         Validation Loss: 3.748256
Validation loss decreased (3.829316 --> 3.748256).  Saving model ...
Epoch: 9         Training Loss: 3.685254         Validation Loss: 3.727593
Validation loss decreased (3.748256 --> 3.727593).  Saving model ...
```

```
Epoch: 10 	Training Loss: 3.600853 	Validation Loss: 3.611072
Validation loss decreased (3.727593 --> 3.611072).  Saving model ...
Epoch: 11 	Training Loss: 3.507229 	Validation Loss: 3.627371
Epoch: 12 	Training Loss: 3.426002 	Validation Loss: 3.523224
Validation loss decreased (3.611072 --> 3.523224).  Saving model ...
Epoch: 13 	Training Loss: 3.353586 	Validation Loss: 3.453768
Validation loss decreased (3.523224 --> 3.453768).  Saving model ...
Epoch: 14 	Training Loss: 3.300133 	Validation Loss: 3.510188
Epoch: 15 	Training Loss: 3.211167 	Validation Loss: 3.379157
Validation loss decreased (3.453768 --> 3.379157).  Saving model ...
Epoch: 16 	Training Loss: 3.152943 	Validation Loss: 3.403621
Epoch: 17 	Training Loss: 3.091194 	Validation Loss: 3.326652
Validation loss decreased (3.379157 --> 3.326652).  Saving model ...
Epoch: 18 	Training Loss: 3.054357 	Validation Loss: 3.398203
Epoch: 19 	Training Loss: 2.989785 	Validation Loss: 3.275832
Validation loss decreased (3.326652 --> 3.275832).  Saving model ...
Epoch: 20 	Training Loss: 2.943362 	Validation Loss: 3.290088
Epoch: 21 	Training Loss: 2.849448 	Validation Loss: 3.288027
Epoch: 22 	Training Loss: 2.822857 	Validation Loss: 3.267239
Validation loss decreased (3.275832 --> 3.267239).  Saving model ...
Epoch: 23 	Training Loss: 2.768841 	Validation Loss: 3.247331
Validation loss decreased (3.267239 --> 3.247331).  Saving model ...
Epoch: 24 	Training Loss: 2.710224 	Validation Loss: 3.262791
Epoch: 25 	Training Loss: 2.647130 	Validation Loss: 3.242076
Validation loss decreased (3.247331 --> 3.242076).  Saving model ...
Epoch: 26 	Training Loss: 2.633497 	Validation Loss: 3.231065
Validation loss decreased (3.242076 --> 3.231065).  Saving model ...
Epoch: 27 	Training Loss: 2.580645 	Validation Loss: 3.234056
Epoch: 28 	Training Loss: 2.535292 	Validation Loss: 3.211452
Validation loss decreased (3.231065 --> 3.211452).  Saving model ...
Epoch: 29 	Training Loss: 2.482041 	Validation Loss: 3.176485
Validation loss decreased (3.211452 --> 3.176485).  Saving model ...
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

In [82]:
```
model_scratch.load_state_dict(torch.load('model_scratch.pt'))

def test(loaders, model, criterion, use_cuda):

    # monitor test loss and accuracy
    test_loss = 0.0
    correct = 0.0
    total = 0.0
```

```
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders[2]):
            # 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: 3.224298


Test Accuracy: 24% (201/836)
```

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 [83]: ## TODO: Specify data loaders
         import os
         from torchvision import datasets
         from PIL import ImageFile
```

```
ImageFile.LOAD_TRUNCATED_IMAGES = True
# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 16
# convert data to a normalized torch.FloatTensor
transform = transforms.Compose([
transforms.RandomRotation(45),
transforms.Resize((224,224)),
transforms.ToTensor(),
transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset = datasets.ImageFolder(root='/data/dog_images/train',
transform=transform)
test_dataset = datasets.ImageFolder(root='/data/dog_images/test',
transform=transform)
valid_dataset = datasets.ImageFolder(root='/data/dog_images/valid',
transform=transform)
data = {"train" : train_dataset, "valid" : valid_dataset, "test" : test_dataset}
loaders_transfer = {0 : torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, shuffle=True,
num_workers=num_workers)
,1 : torch.utils.data.DataLoader(valid_dataset,
batch_size=batch_size,
num_workers=num_workers)
,2 : torch.utils.data.DataLoader(test_dataset,
batch_size=4, shuffle=True,
num_workers=num_workers)}
```

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

         ## TODO: Specify model architecture

         model_transfer = models.vgg16(pretrained=True)



         for param in model_transfer.features.parameters():
             param.requires_grad = False

         model_transfer.classifier[6]=torch.nn.Linear(4096,133)
```

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

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

Answer: I used the same VGG model with the same weights on the convolutional layers and drop layers but i have used different finel linear layer with 133 output of dog bread because of the small size of my data and the fact that it is close to the data that I was previously trained on VGG model

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 [85]: import torch.optim as optim

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

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 [ ]: # train the model
        model_transfer =train(50, loaders_transfer, model_transfer, optimizer_transfer, criterio

        # 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: 3.971319        Validation Loss: 2.657659
Validation loss decreased (inf --> 2.657659).  Saving model ...
Epoch: 2         Training Loss: 2.095121        Validation Loss: 1.419118
Validation loss decreased (2.657659 --> 1.419118).  Saving model ...
Epoch: 3         Training Loss: 1.418794        Validation Loss: 1.060492
Validation loss decreased (1.419118 --> 1.060492).  Saving model ...
Epoch: 4         Training Loss: 1.195815        Validation Loss: 0.943234
Validation loss decreased (1.060492 --> 0.943234).  Saving model ...
Epoch: 5         Training Loss: 1.024925        Validation Loss: 0.903657
Validation loss decreased (0.943234 --> 0.903657).  Saving model ...
Epoch: 6         Training Loss: 0.932685        Validation Loss: 0.809092
Validation loss decreased (0.903657 --> 0.809092).  Saving model ...
Epoch: 7         Training Loss: 0.875713        Validation Loss: 0.797572
Validation loss decreased (0.809092 --> 0.797572).  Saving model ...
Epoch: 8         Training Loss: 0.819681        Validation Loss: 0.780811
Validation loss decreased (0.797572 --> 0.780811).  Saving model ...
Epoch: 9         Training Loss: 0.746925        Validation Loss: 0.759027
```

```
Validation loss decreased (0.780811 --> 0.759027).  Saving model ...
Epoch: 10         Training Loss: 0.737224         Validation Loss: 0.729879
Validation loss decreased (0.759027 --> 0.729879).  Saving model ...
Epoch: 11         Training Loss: 0.685718         Validation Loss: 0.748925
Epoch: 12         Training Loss: 0.669677         Validation Loss: 0.725773
Validation loss decreased (0.729879 --> 0.725773).  Saving model ...
Epoch: 13         Training Loss: 0.645981         Validation Loss: 0.747540
Epoch: 14         Training Loss: 0.622783         Validation Loss: 0.721736
Validation loss decreased (0.725773 --> 0.721736).  Saving model ...
Epoch: 15         Training Loss: 0.593957         Validation Loss: 0.702301
Validation loss decreased (0.721736 --> 0.702301).  Saving model ...
Epoch: 16         Training Loss: 0.569698         Validation Loss: 0.728802
Epoch: 17         Training Loss: 0.547321         Validation Loss: 0.690065
Validation loss decreased (0.702301 --> 0.690065).  Saving model ...
Epoch: 18         Training Loss: 0.535204         Validation Loss: 0.668215
Validation loss decreased (0.690065 --> 0.668215).  Saving model ...
Epoch: 19         Training Loss: 0.522946         Validation Loss: 0.681632
Epoch: 20         Training Loss: 0.510970         Validation Loss: 0.663390
Validation loss decreased (0.668215 --> 0.663390).  Saving model ...
Epoch: 21         Training Loss: 0.488621         Validation Loss: 0.672864
Epoch: 22         Training Loss: 0.483402         Validation Loss: 0.655389
Validation loss decreased (0.663390 --> 0.655389).  Saving model ...
Epoch: 23         Training Loss: 0.475028         Validation Loss: 0.697595
Epoch: 24         Training Loss: 0.475395         Validation Loss: 0.658401
Epoch: 25         Training Loss: 0.455119         Validation Loss: 0.646250
Validation loss decreased (0.655389 --> 0.646250).  Saving model ...
Epoch: 26         Training Loss: 0.442444         Validation Loss: 0.688119
Epoch: 27         Training Loss: 0.428311         Validation Loss: 0.669664
Epoch: 28         Training Loss: 0.401939         Validation Loss: 0.683290
Epoch: 29         Training Loss: 0.398759         Validation Loss: 0.668991
Epoch: 30         Training Loss: 0.402228         Validation Loss: 0.665745
Epoch: 31         Training Loss: 0.381120         Validation Loss: 0.666035
```

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%.

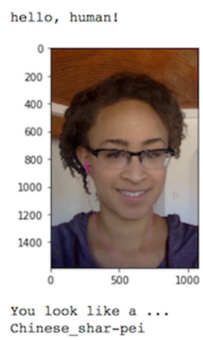
```
In [86]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))

         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.670501


Test Accuracy: 79% (667/836)
```

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 [87]:
```
### 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]

data_transfer=data

class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]

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

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

    img= Image.open(img_path)
    img_t=transform(img)
    batch_t = torch.unsqueeze(img_t, 0)

    model_transfer = models.vgg16(pretrained=True)



    for param in model_transfer.features.parameters():
        param.requires_grad = False

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

    model_transfer.load_state_dict(torch.load('model_transfer.pt'))

    output=model_transfer(batch_t)

    _, preds_tensor = torch.max(output, 1)
    preds_tensor=preds_tensor.to(torch.device("cpu"))
    preds = np.squeeze(preds_tensor.numpy())

    for dog_type in range(len(class_names)) :
        if dog_type==preds:
            return class_names[dog_type]

    return None
```



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 [88]:
```
### 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
    plt.subplots()
    img = Image.open(img_path)
    plt.imshow(img)


    predict=face_detector(img_path)
    if predict:
        human_dog_breed= predict_breed_transfer(img_path)
        plt.title("hello ,humen you look like {}".format(human_dog_breed))

    predict=dog_detector(img_path)

    if predict:
        dog_breed= predict_breed_transfer(img_path)
        plt.title("the dog bread is {}".format(dog_breed))
```

```
            return "oh It seems that I did not know the content of the image Are you sure it is
```

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!

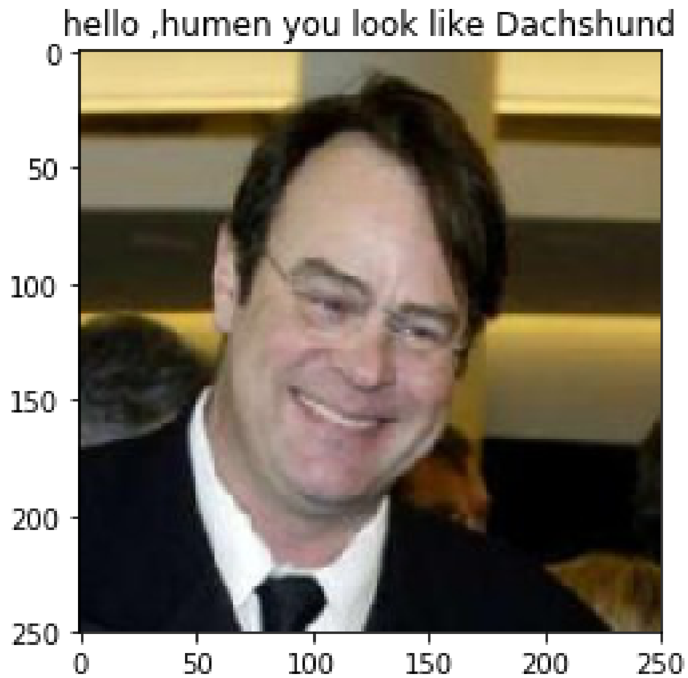
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

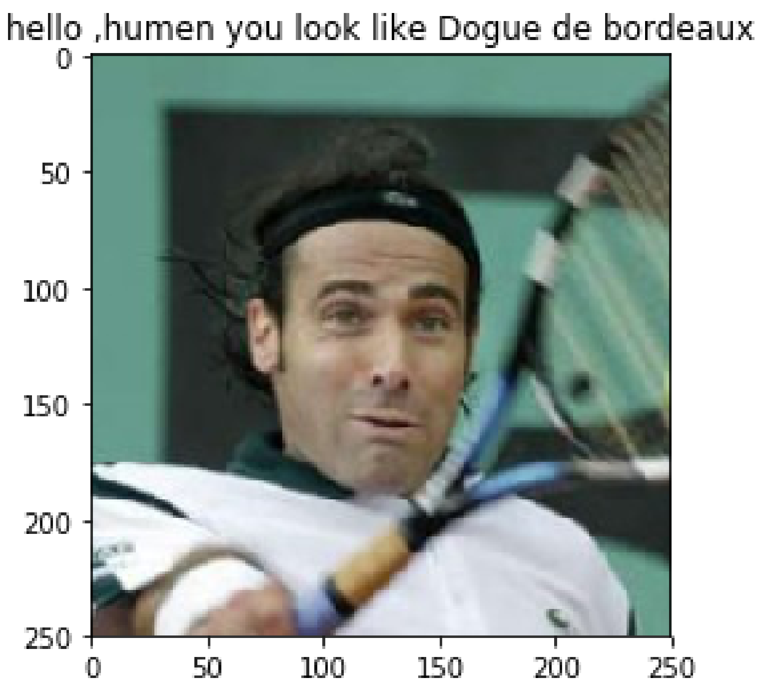
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

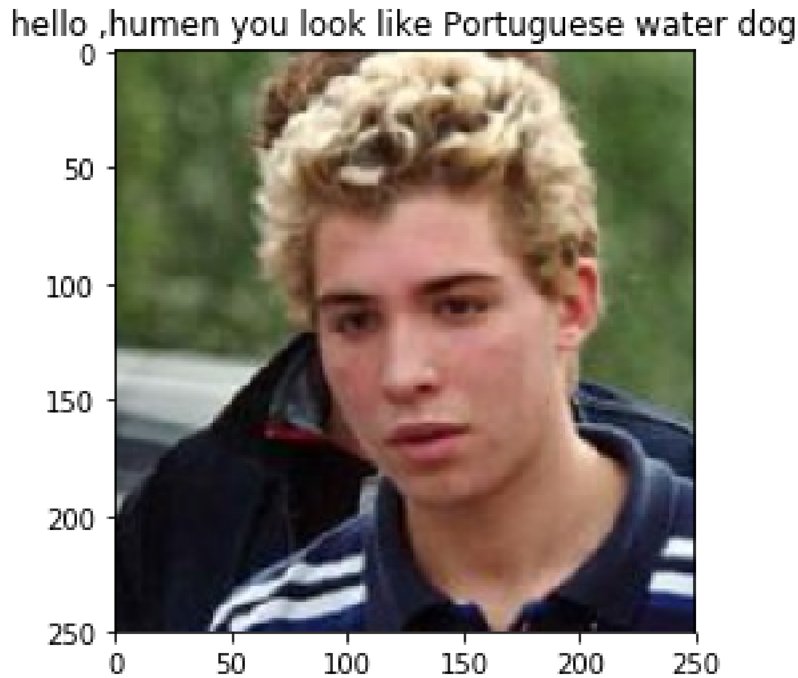
Answer: (Three possible points for improvement)

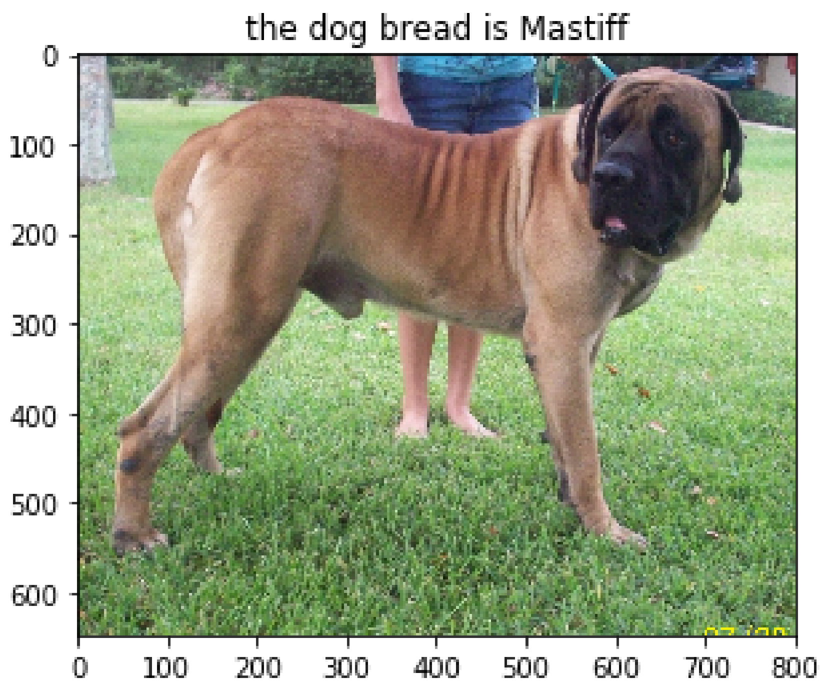
The results are very good.
1- I think that I can do the exercise of the dog type recognition model more.
2- use more accurate facial recognition .
3- make a transfer that integrates the closest image of the human face of the type of dogs with it and displays it.

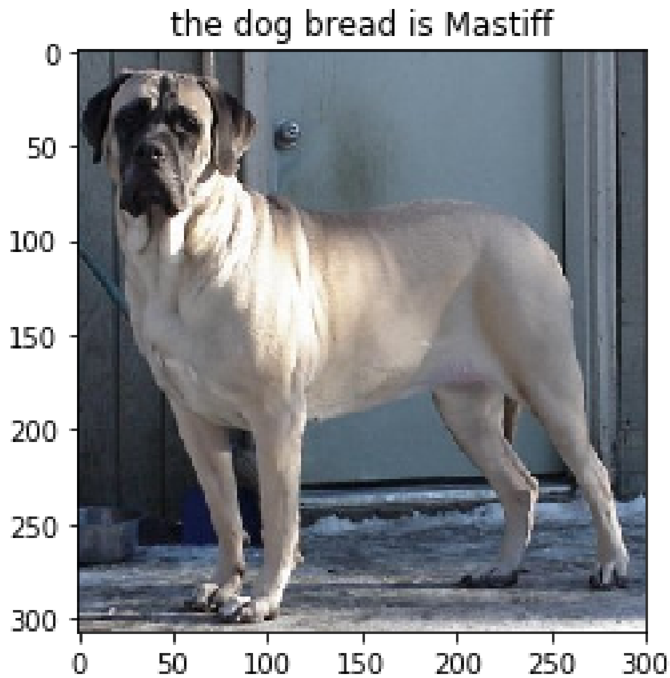
```
In [89]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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

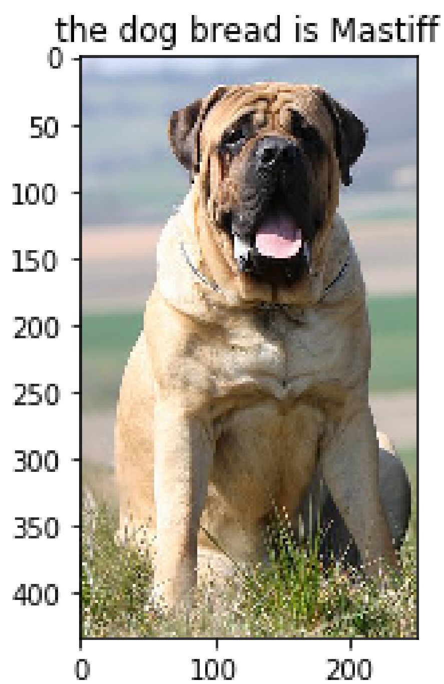

hello ,humen you look like Dachshund
0
50
100
150
200
250
0
50
100
150
200
250


hello ,humen you look like Dogue de bordeaux
0
50
100
150
200
250
0
50
100
150
200
250


hello ,humen you look like Portuguese water dog


the dog bread is Mastiff


the dog bread is Mastiff
0
50
100
150
200
250
300
0
50
100
150
200
250
300


the dog bread is Mastiff
0
50
100
150
200
250
300
350
400
0
100
200