dog_app

December 23, 2018

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

- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

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

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

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

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

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

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

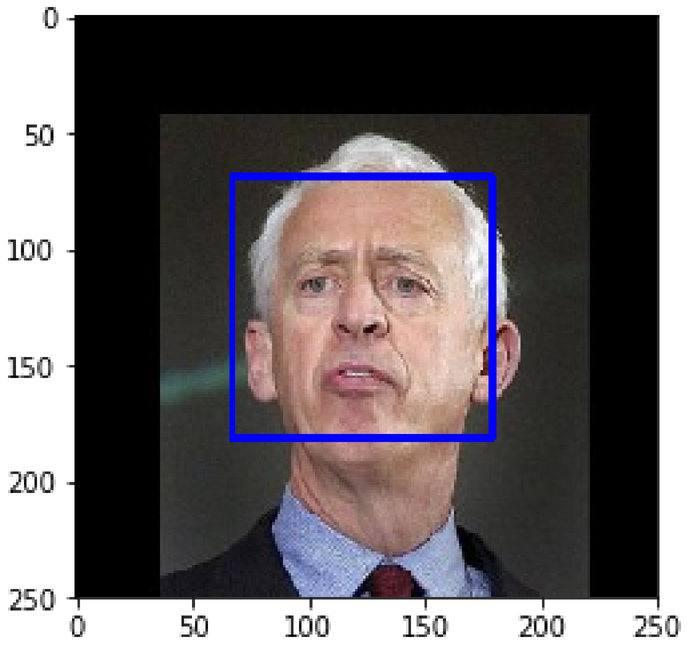
## Step 1: Detect Humans

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

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

        # load color (BGR) image
        img = cv2.imread(human_files[3])
        # 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 [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

```
        img = cv2.imread(img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

        #-#-# Do NOT modify the code above this line. #-#-#
        human_counter,dog_counter = 0,0
        for human_image,dog_image in zip(human_files_short,dog_files_short):
            human_counter+= 1 if face_detector(human_image) else 0
            dog_counter+= 1 if face_detector(dog_image) else 0

        print('human percentage {:3.2f}%'.format((human_counter/len(human_files_short))*100))
        print('dog percentage {:3.2f}%'.format((dog_counter/len(dog_files_short))*100))


        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
```

```
human percentage 98.00%
dog percentage 17.00%
```

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

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

## Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

```
In [6]: import torch
        import torchvision.models as models

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True).to(device)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:04<00:00, 122571278.09it/s]
```

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 [7]: from PIL import Image
        import PIL.Image
        import torchvision.transforms as transforms

        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

        def VGG16_predict(img_path):
            # VGG-16 Takes 224x224 images as input, so we resize all of them
            normalize = transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])

            preprocess = transforms.Compose([
                transforms.Resize(256),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                normalize])

            image = PIL.Image.open(str(img_path))
            image = preprocess(image)
            if torch.cuda.is_available():
                image = image.cuda()
            image.unsqueeze_(0)
            output = VGG16(image)
            probabilities = torch.exp(output)
            top_probability, top_class = probabilities.topk(1, dim=1)
            return top_class.item()
```

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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        DOG_LOWER, DOG_UPPER = 151, 268
        def dog_detector(img_path):
            ## TODO: Complete the function.

            return DOG_LOWER < VGG16_predict(img_path) < DOG_UPPER
```

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: 0% for dogs in human picture 100% dogs in dogs pictures

```
In [9]: human_counter,dog_counter = 0,0
        for human_image,dog_image in zip(human_files_short,dog_files_short):
            human_counter+= 1 if dog_detector(human_image) else 0
            dog_counter+= 1 if dog_detector(dog_image) else 0

        print('dog percentage in human {:3.2f}%'.format((human_counter/len(human_files_short))*1
        print('dog percentage in dog {:3.2f}%'.format((dog_counter/len(dog_files_short))*100))
```

```
dog percentage in human 0.00%
dog percentage in dog 100.00%
```

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

```
In [ ]: ### (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 [10]: import os
         from glob import glob
         from torchvision import datasets, models, transforms
         import matplotlib.pyplot as plt
         import torch

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         # define dataloader parameters
         batch_size = 20
         num_workers=0

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

         # convert data to a normalized torch.FloatTensor
         data_transform = transforms.Compose([
             transforms.RandomResizedCrop(224),
             transforms.ToTensor(),
             transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])


         # set the pathes to the data sub directories
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         test_dir = os.path.join(data_dir, 'test/')
         valid_dir = os.path.join(data_dir,'valid/')



         train_data = datasets.ImageFolder(train_dir, transform=data_transform)
         test_data = datasets.ImageFolder(test_dir, transform=data_transform)
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)

         # prepare data loaders
         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                    num_workers=num_workers, shuffle=True)
         test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
```

```
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)

loaders_scratch = {}
loaders_scratch['train'] = train_loader
loaders_scratch['test'] = valid_loader
loaders_scratch['valid'] = test_loader
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer: I chose to resize image to 224 because VGG16 works well with 224 image size. I didnt augment dataset. I wanted to see how the network performs without it first

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

         use_cuda = torch.cuda.is_available()

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 # convolutional layer (sees 224x224x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # convolutional layer (sees 112x112x16 tensor)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 # convolutional layer (sees 56x56x32 tensor)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 # linear layer (64 * 28 * 28 -> 50176)
                 self.fc1 = nn.Linear(64 * 28 * 28, 1024)
                 self.fc2 = nn.Linear(1024, 512)
                 self.fc3 = nn.Linear(512, 133)
                 self.dropout = nn.Dropout(0.25)



             def forward(self, x):
                 # add sequence of convolutional and max pooling layers
                 x = self.pool(F.relu(self.conv1(x)))
```

```
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        # flatten image input
        x = x.view(-1, 64 * 28 * 28)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add 3rd hidden layer, with relu activation function
        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: VGG16 seems to work resonable well so I didn't modify it furture by adding/removing any layers. if It didnt perform well I would modify layers to increase the accuracy of the network

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
         import numpy as np

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [13]: 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_scratch.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     output = model_scratch(data)
                     optimizer.zero_grad()
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss = train_loss + ( (1/ (batch_idx + 1) ) * (loss.data - train_loss

                 ######################
                 # validate the model #
                 ######################
                 model_scratch.eval()
                 for batch_idx, (data, target) in enumerate(loaders['valid']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## update the average validation loss
                     valid_output = model_scratch(data)
                     loss = criterion(valid_output, target)
                     valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


                 # print training/validation statistics
```

```
                print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
                    epoch,
                    train_loss,
                    valid_loss
                    ))

                ## TODO: save the model if validation loss has decreased
                if valid_loss <= valid_loss_min:
                    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_scratch


        # train the model
        model_scratch = train(100, 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: 4.889875        Validation Loss: 4.887188
Validation loss decreased (inf --> 4.887188).  Saving model ...
Epoch: 2        Training Loss: 4.886193        Validation Loss: 4.882294
Validation loss decreased (4.887188 --> 4.882294).  Saving model ...
Epoch: 3        Training Loss: 4.881273        Validation Loss: 4.873181
Validation loss decreased (4.882294 --> 4.873181).  Saving model ...
Epoch: 4        Training Loss: 4.873718        Validation Loss: 4.861144
Validation loss decreased (4.873181 --> 4.861144).  Saving model ...
Epoch: 5        Training Loss: 4.866223        Validation Loss: 4.851496
Validation loss decreased (4.861144 --> 4.851496).  Saving model ...
Epoch: 6        Training Loss: 4.854913        Validation Loss: 4.830202
Validation loss decreased (4.851496 --> 4.830202).  Saving model ...
Epoch: 7        Training Loss: 4.812630        Validation Loss: 4.766119
Validation loss decreased (4.830202 --> 4.766119).  Saving model ...
Epoch: 8        Training Loss: 4.752745        Validation Loss: 4.718157
Validation loss decreased (4.766119 --> 4.718157).  Saving model ...
Epoch: 9        Training Loss: 4.689977        Validation Loss: 4.642375
Validation loss decreased (4.718157 --> 4.642375).  Saving model ...
Epoch: 10        Training Loss: 4.632619        Validation Loss: 4.583909
Validation loss decreased (4.642375 --> 4.583909).  Saving model ...
Epoch: 11        Training Loss: 4.590381        Validation Loss: 4.581051
Validation loss decreased (4.583909 --> 4.581051).  Saving model ...
Epoch: 12        Training Loss: 4.567843        Validation Loss: 4.503722
```

```
Validation loss decreased (4.581051 --> 4.503722).  Saving model ...
Epoch: 13 	Training Loss: 4.528212 	Validation Loss: 4.527297
Epoch: 14 	Training Loss: 4.500091 	Validation Loss: 4.504893
Epoch: 15 	Training Loss: 4.475224 	Validation Loss: 4.480901
Validation loss decreased (4.503722 --> 4.480901).  Saving model ...
Epoch: 16 	Training Loss: 4.450442 	Validation Loss: 4.473807
Validation loss decreased (4.480901 --> 4.473807).  Saving model ...
Epoch: 17 	Training Loss: 4.425238 	Validation Loss: 4.446131
Validation loss decreased (4.473807 --> 4.446131).  Saving model ...
Epoch: 18 	Training Loss: 4.405654 	Validation Loss: 4.401535
Validation loss decreased (4.446131 --> 4.401535).  Saving model ...
Epoch: 19 	Training Loss: 4.386006 	Validation Loss: 4.421918
Epoch: 20 	Training Loss: 4.358000 	Validation Loss: 4.373992
Validation loss decreased (4.401535 --> 4.373992).  Saving model ...
Epoch: 21 	Training Loss: 4.338986 	Validation Loss: 4.360038
Validation loss decreased (4.373992 --> 4.360038).  Saving model ...
Epoch: 22 	Training Loss: 4.315027 	Validation Loss: 4.307079
Validation loss decreased (4.360038 --> 4.307079).  Saving model ...
Epoch: 23 	Training Loss: 4.277020 	Validation Loss: 4.306912
Validation loss decreased (4.307079 --> 4.306912).  Saving model ...
Epoch: 24 	Training Loss: 4.279337 	Validation Loss: 4.320511
Epoch: 25 	Training Loss: 4.255278 	Validation Loss: 4.346368
Epoch: 26 	Training Loss: 4.214726 	Validation Loss: 4.318421
Epoch: 27 	Training Loss: 4.192971 	Validation Loss: 4.271269
Validation loss decreased (4.306912 --> 4.271269).  Saving model ...
Epoch: 28 	Training Loss: 4.182899 	Validation Loss: 4.222484
Validation loss decreased (4.271269 --> 4.222484).  Saving model ...
Epoch: 29 	Training Loss: 4.170583 	Validation Loss: 4.324975
Epoch: 30 	Training Loss: 4.145036 	Validation Loss: 4.244522
Epoch: 31 	Training Loss: 4.124596 	Validation Loss: 4.238822
Epoch: 32 	Training Loss: 4.093071 	Validation Loss: 4.254711
Epoch: 33 	Training Loss: 4.093788 	Validation Loss: 4.189815
Validation loss decreased (4.222484 --> 4.189815).  Saving model ...
Epoch: 34 	Training Loss: 4.059215 	Validation Loss: 4.278981
Epoch: 35 	Training Loss: 4.042684 	Validation Loss: 4.183824
Validation loss decreased (4.189815 --> 4.183824).  Saving model ...
Epoch: 36 	Training Loss: 4.025827 	Validation Loss: 4.205773
Epoch: 37 	Training Loss: 4.006161 	Validation Loss: 4.223042
Epoch: 38 	Training Loss: 3.981204 	Validation Loss: 4.139154
Validation loss decreased (4.183824 --> 4.139154).  Saving model ...
Epoch: 39 	Training Loss: 3.937263 	Validation Loss: 4.177459
Epoch: 40 	Training Loss: 3.916361 	Validation Loss: 4.156765
Epoch: 41 	Training Loss: 3.901338 	Validation Loss: 4.160783
Epoch: 42 	Training Loss: 3.895740 	Validation Loss: 4.131145
Validation loss decreased (4.139154 --> 4.131145).  Saving model ...
Epoch: 43 	Training Loss: 3.867314 	Validation Loss: 4.100809
Validation loss decreased (4.131145 --> 4.100809).  Saving model ...
Epoch: 44 	Training Loss: 3.833627 	Validation Loss: 4.090057
```

```
Validation loss decreased (4.100809 --> 4.090057).  Saving model ...
Epoch: 45        Training Loss: 3.812904        Validation Loss: 4.121654
Epoch: 46        Training Loss: 3.794885        Validation Loss: 4.078866
Validation loss decreased (4.090057 --> 4.078866).  Saving model ...
Epoch: 47        Training Loss: 3.758514        Validation Loss: 4.108519
Epoch: 48        Training Loss: 3.745343        Validation Loss: 4.082185
Epoch: 49        Training Loss: 3.733916        Validation Loss: 4.127425
Epoch: 50        Training Loss: 3.701480        Validation Loss: 4.109523
Epoch: 51        Training Loss: 3.696167        Validation Loss: 4.043377
Validation loss decreased (4.078866 --> 4.043377).  Saving model ...
Epoch: 52        Training Loss: 3.656136        Validation Loss: 4.158065
Epoch: 53        Training Loss: 3.612526        Validation Loss: 4.083551
Epoch: 54        Training Loss: 3.626987        Validation Loss: 4.094752
Epoch: 55        Training Loss: 3.588694        Validation Loss: 4.102215
Epoch: 56        Training Loss: 3.560844        Validation Loss: 3.997990
Validation loss decreased (4.043377 --> 3.997990).  Saving model ...
Epoch: 57        Training Loss: 3.555960        Validation Loss: 4.043593
Epoch: 58        Training Loss: 3.527645        Validation Loss: 4.019653
Epoch: 59        Training Loss: 3.513016        Validation Loss: 3.999606
Epoch: 60        Training Loss: 3.467328        Validation Loss: 4.104486
Epoch: 61        Training Loss: 3.448616        Validation Loss: 4.067416
Epoch: 62        Training Loss: 3.447357        Validation Loss: 3.931874
Validation loss decreased (3.997990 --> 3.931874).  Saving model ...
Epoch: 63        Training Loss: 3.413180        Validation Loss: 4.066048
Epoch: 64        Training Loss: 3.399317        Validation Loss: 3.987575
Epoch: 65        Training Loss: 3.364875        Validation Loss: 3.986837
Epoch: 66        Training Loss: 3.373736        Validation Loss: 3.974311
Epoch: 67        Training Loss: 3.310589        Validation Loss: 3.991368
Epoch: 68        Training Loss: 3.299910        Validation Loss: 4.016927
Epoch: 69        Training Loss: 3.299911        Validation Loss: 3.940525
Epoch: 70        Training Loss: 3.256166        Validation Loss: 4.037337
Epoch: 71        Training Loss: 3.245584        Validation Loss: 4.104031
Epoch: 72        Training Loss: 3.214890        Validation Loss: 4.026923
Epoch: 73        Training Loss: 3.157124        Validation Loss: 4.039547
Epoch: 74        Training Loss: 3.164147        Validation Loss: 4.084382
Epoch: 75        Training Loss: 3.144150        Validation Loss: 4.060569
Epoch: 76        Training Loss: 3.155766        Validation Loss: 3.972074
Epoch: 77        Training Loss: 3.075011        Validation Loss: 4.039514
Epoch: 78        Training Loss: 3.077507        Validation Loss: 4.096889
Epoch: 79        Training Loss: 3.033070        Validation Loss: 3.994619
Epoch: 80        Training Loss: 3.052697        Validation Loss: 4.040046
Epoch: 81        Training Loss: 3.006931        Validation Loss: 4.046448
Epoch: 82        Training Loss: 3.007349        Validation Loss: 4.136899
Epoch: 83        Training Loss: 2.966326        Validation Loss: 4.114817
Epoch: 84        Training Loss: 2.950780        Validation Loss: 4.069660
Epoch: 85        Training Loss: 2.907016        Validation Loss: 4.103395
Epoch: 86        Training Loss: 2.917519        Validation Loss: 4.087334
Epoch: 87        Training Loss: 2.876752        Validation Loss: 4.039225
```

```
Epoch: 88        Training Loss: 2.873645        Validation Loss: 3.983921
Epoch: 89        Training Loss: 2.856347        Validation Loss: 4.018569
Epoch: 90        Training Loss: 2.848808        Validation Loss: 4.034148
Epoch: 91        Training Loss: 2.823375        Validation Loss: 4.095042
Epoch: 92        Training Loss: 2.746257        Validation Loss: 4.122802
Epoch: 93        Training Loss: 2.759711        Validation Loss: 4.058539
Epoch: 94        Training Loss: 2.751487        Validation Loss: 4.049664
Epoch: 95        Training Loss: 2.723165        Validation Loss: 4.084633
Epoch: 96        Training Loss: 2.686018        Validation Loss: 4.176244
Epoch: 97        Training Loss: 2.693659        Validation Loss: 4.016965
Epoch: 98        Training Loss: 2.673019        Validation Loss: 4.031929
Epoch: 99        Training Loss: 2.625636        Validation Loss: 4.049385
Epoch: 100         Training Loss: 2.632522         Validation Loss: 4.088660
```

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

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

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

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

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

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

Test Loss: 4.001387


Test Accuracy: 11% (92/835)
```

## 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 [16]: import os
         from glob import glob
         from torchvision import datasets, models, transforms
         import matplotlib.pyplot as plt
         import torch

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         ## TODO: Specify data loaders
         # define dataloader parameters
         batch_size = 20
         num_workers=0
         # set the pathes to the data sub directories
         data_dir = '/data/dog_images/'
         train_dir = os.path.join(data_dir, 'train/')
         test_dir = os.path.join(data_dir, 'test/')
         valid_dir = os.path.join(data_dir,'valid/')

         # VGG-16 Takes 224x224 images as input, so we resize all of them
         data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                              transforms.ToTensor()])

         train_data = datasets.ImageFolder(train_dir, transform=data_transform)
         test_data = datasets.ImageFolder(test_dir, transform=data_transform)
```

```
valid_data = datasets.ImageFolder(valid_dir, transform=data_transform)

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)

valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                          num_workers=num_workers, shuffle=True)

data_loaders = {}
data_loaders['train'] = train_loader
data_loaders['test'] = valid_loader
data_loaders['valid'] = test_loader
```

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

         device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

         model_transfer = models.vgg16(pretrained=True)

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

         n_inputs = model_transfer.classifier[6].in_features
         #number of dogs in train_data is 133
         last_layer = nn.Linear(n_inputs, len(train_data.classes))
         model_transfer.classifier[6] = last_layer

         model_transfer.to(device)
         print(model_transfer.classifier[6].out_features)
```

```
133
```

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 took the VGG16 architecture and added a fully connected linear layer at the end with 133 ouput nodes to predict the 133 dog breed the train_data object contains. VGG16 is a pretty good model for this task and I dont feel I should modify it too much to include more layers, but it's possible that I won't achieve the best results during testing

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 [18]: import torch.optim as optim
         import numpy as np
         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.005)
```

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 [19]: # train the model


         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_transfer.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     output = model_transfer(data)
                     optimizer.zero_grad()
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss = train_loss + ( (1/ (batch_idx + 1) ) * (loss.data - train_loss

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

```
            model_transfer.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                valid_output = model_transfer(data)
                loss = criterion(valid_output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


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

            ## TODO: save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                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_transfer
```

```
model_transfer = train(30, data_loaders, model_transfer, optimizer_transfer, criterion_

# 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: 3.495628         Validation Loss: 2.261879
Validation loss decreased (inf --> 2.261879).  Saving model ...
Epoch: 2         Training Loss: 2.139978         Validation Loss: 1.827351
Validation loss decreased (2.261879 --> 1.827351).  Saving model ...
Epoch: 3         Training Loss: 1.829739         Validation Loss: 1.594619
Validation loss decreased (1.827351 --> 1.594619).  Saving model ...
Epoch: 4         Training Loss: 1.659845         Validation Loss: 1.541265
Validation loss decreased (1.594619 --> 1.541265).  Saving model ...
Epoch: 5         Training Loss: 1.576695         Validation Loss: 1.444355
Validation loss decreased (1.541265 --> 1.444355).  Saving model ...
Epoch: 6         Training Loss: 1.472359         Validation Loss: 1.369518
Validation loss decreased (1.444355 --> 1.369518).  Saving model ...
```

```
Epoch: 7        Training Loss: 1.448186        Validation Loss: 1.380496
Epoch: 8        Training Loss: 1.356554        Validation Loss: 1.377313
Epoch: 9        Training Loss: 1.339813        Validation Loss: 1.440123
Epoch: 10        Training Loss: 1.315165        Validation Loss: 1.441095
Epoch: 11        Training Loss: 1.244133        Validation Loss: 1.386069
Epoch: 12        Training Loss: 1.192402        Validation Loss: 1.445642
Epoch: 13        Training Loss: 1.199571        Validation Loss: 1.380319
Epoch: 14        Training Loss: 1.168877        Validation Loss: 1.329039
Validation loss decreased (1.369518 --> 1.329039).  Saving model ...
Epoch: 15        Training Loss: 1.165229        Validation Loss: 1.377387
Epoch: 16        Training Loss: 1.103899        Validation Loss: 1.396441
Epoch: 17        Training Loss: 1.105143        Validation Loss: 1.394721
Epoch: 18        Training Loss: 1.089787        Validation Loss: 1.361182
Epoch: 19        Training Loss: 1.100637        Validation Loss: 1.337179
Epoch: 20        Training Loss: 1.060784        Validation Loss: 1.405429
Epoch: 21        Training Loss: 1.060981        Validation Loss: 1.436695
Epoch: 22        Training Loss: 1.025963        Validation Loss: 1.201844
Validation loss decreased (1.329039 --> 1.201844).  Saving model ...
Epoch: 23        Training Loss: 1.009449        Validation Loss: 1.245653
Epoch: 24        Training Loss: 1.022469        Validation Loss: 1.291696
Epoch: 25        Training Loss: 1.005555        Validation Loss: 1.288684
Epoch: 26        Training Loss: 0.963095        Validation Loss: 1.295873
Epoch: 27        Training Loss: 0.968997        Validation Loss: 1.401946
Epoch: 28        Training Loss: 0.939150        Validation Loss: 1.274284
Epoch: 29        Training Loss: 0.963448        Validation Loss: 1.255587
Epoch: 30        Training Loss: 0.966502        Validation Loss: 1.269447
```

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%.

1.1.17 I interrupted the training in the middle,but still was able to get above 60%

```
In [20]: test(data_loaders, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.172740


Test Accuracy: 66% (557/835)
```

1.1.18 (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 [21]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         # 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)
         print(len(class_names))

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

             preprocess = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 normalize])

             image = PIL.Image.open(str(img_path))
             image = preprocess(image)
             image = image.to(device)
             image.unsqueeze_(0)
             output = model_transfer(image)
             probabilities = torch.exp(output)
             top_probability, top_class = probabilities.topk(1, dim=1)
             # return dog breed that is predicted by the model
             return class_names[top_class.item()]
```

```
['Affenpinscher', 'Afghan hound', 'Airedale terrier', 'Akita', 'Alaskan malamute', 'American esk
133
```

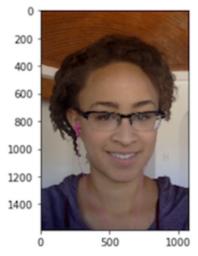
## Step 5: Write your Algorithm

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!

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

1.1.19 (IMPLEMENTATION) Write your Algorithm

```
In [22]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         from IPython.core.display import Image, display
         import PIL.Image
         def run_app(img_path):
             display(PIL.Image.open(img_path))
             if dog_detector(img_path) == 1:
                 print("This is believed to be a dog. Its predicted breed is: ")
                 print(predict_breed_transfer(img_path))
             elif face_detector(img_path) == 1:
                 print("This is believed to be a human. Its predicted breed is: ")
                 print(predict_breed_transfer(img_path))
             else:
                 print("Could not identify a human or dog in the chosen image. Try again.")
```

## 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?

1.1.20 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: the outcome was worse than what I expected. the model predict with more errors than I initially anticipated. to improve the result I would suggest the following steps :

1. dataset augmentation - transform dog images by flipping , rotating and scaling

2. train CNN network with more dog images. there are many dogs that are not included in the class_name object. to do so we will have to download more dog images and probably modify the ouput layer to contain more output nodes

3. More epochs could be used (as long as we are ensuring ourselves that we are not overfitting)

4. I could try to use a different optimizers to see if I can achieve better results

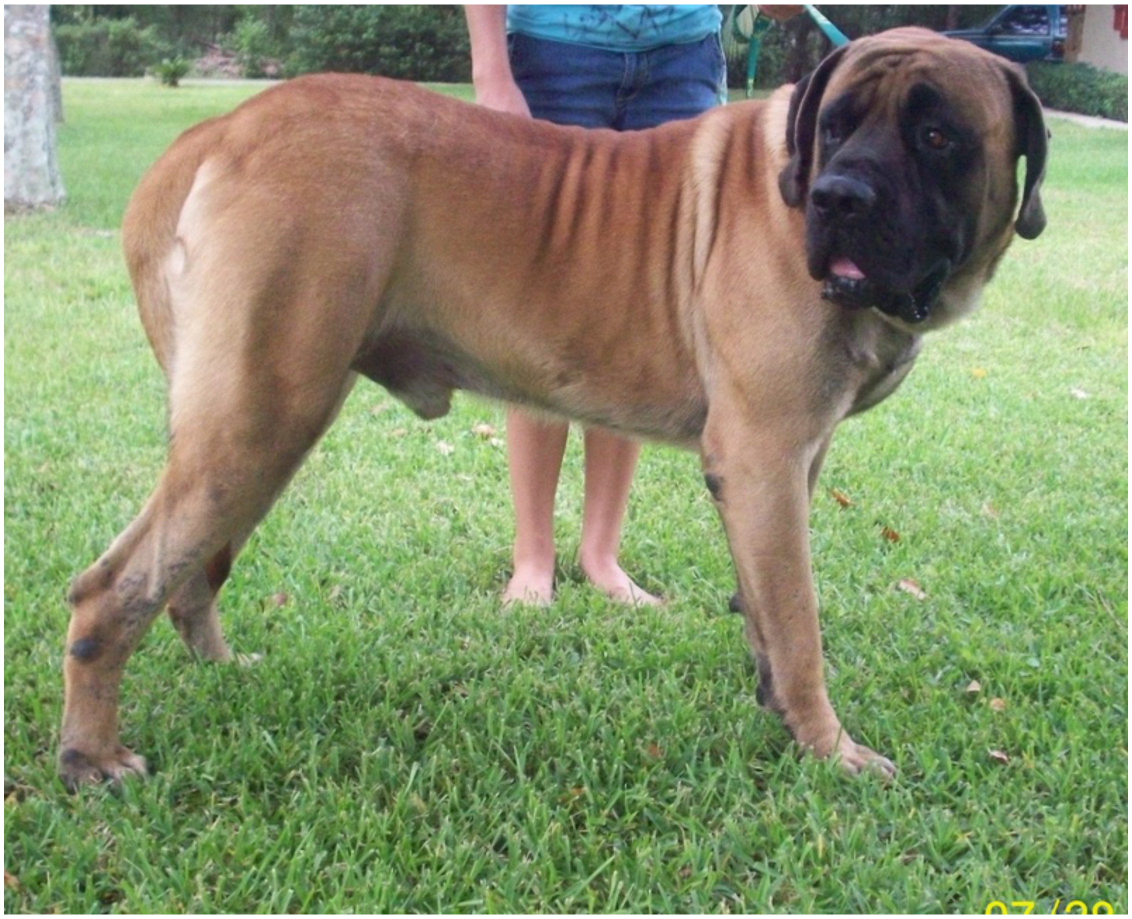
5.

```
In [23]: ## 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)
```

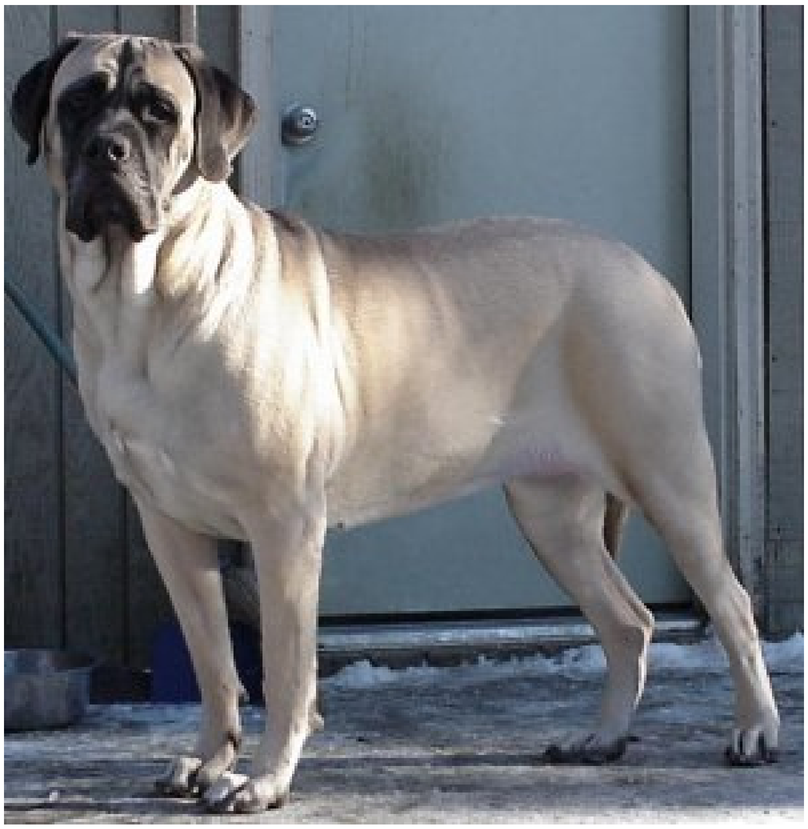
```
This is believed to be a human. Its predicted breed is:
Welsh springer spaniel
```

```
This is believed to be a human. Its predicted breed is:
Australian shepherd
```

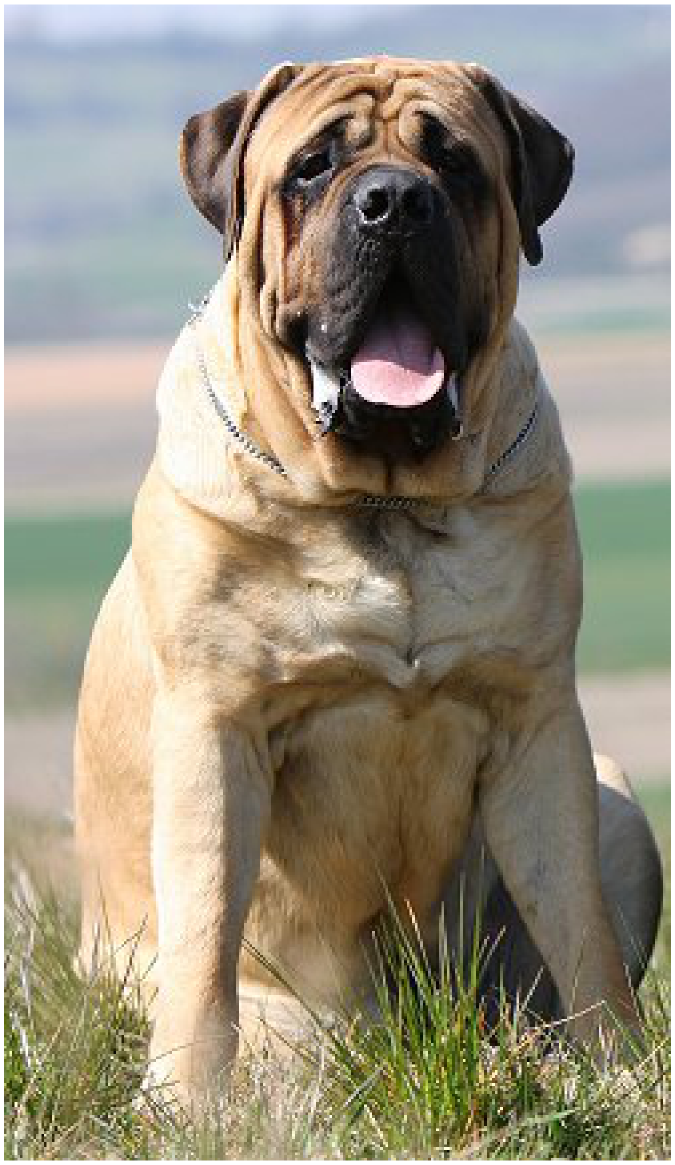
```
This is believed to be a human. Its predicted breed is:
Australian shepherd
```



```
This is believed to be a dog. Its predicted breed is:
Mastiff
```



```
This is believed to be a dog. Its predicted breed is:
Bullmastiff
```



```
This is believed to be a dog. Its predicted breed is:
Bloodhound
```

In [ ]: