dog_app

November 4, 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 [2]: import numpy as np
        from glob import glob
        from PIL import Image

        # 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

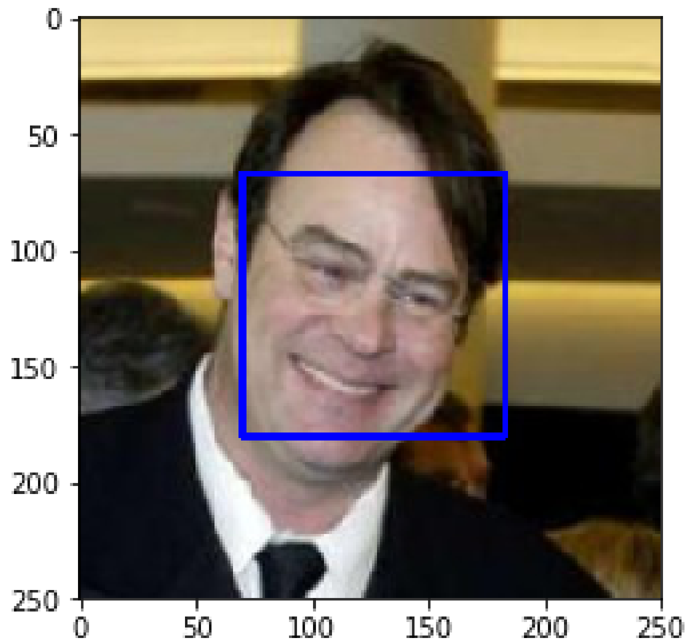
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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 [4]: # 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: - 98% of human faces were detected as human faces, while 17% of dog_photos were incorrectly recognized as having human faces.

```
In [5]: human_files_short = human_files[:100]
        dog_files_short = dog_files[:100]
```

```
In [6]: 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.

        Dog = 0
        Human = 0

        for human, dog in zip(human_files_short, dog_files_short):
            if face_detector(dog):
                Dog += 1
            if face_detector(human):
                Human += 1

        print(str(Human)+ "% Accuracy detecting Human Faces")
        print(str(Dog) + "% False Positive for Human Faces in Dog Photos")
```

```
98% Accuracy detecting Human Faces
17% False Positive for Human Faces in Dog Photos
```

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 [ ]: ### (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 [7]: import torch
        import torchvision.models as models

        # define VGG16 model
        use_cuda = torch.cuda.is_available()

        VGG16 = models.vgg16(pretrained=True)
        #if use_cuda:
        #    VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:18<00:00, 30643718.78it/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 [8]: import os
        import numpy as np
        import torch
        import torchvision
```

```
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F
```

In [9]:
```
def load_image(img_path):

    image = Image.open(img_path).convert('RGB')

    in_transform = transforms.Compose([
        transforms.Resize(256),
        transforms.RandomCrop(224),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406),
                             (0.229, 0.224, 0.255))])

    image = in_transform(image)[:3,:,:].unsqueeze(0)
    return image
```

In [10]:
```
torch.max(VGG16(load_image('images/Labrador_retriever_06457.jpg')), 1)
```

Out[10]:
```
(tensor([ 17.9820]), tensor([ 208]))
```

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 [11]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    content = load_image(img_path)
    pred = torch.max(VGG16(content), 1)[1]
    if pred > 150 and pred < 269:
        return True
    else:
        return False

dog_detector('images/Labrador_retriever_06457.jpg')
```

Out[11]:
```
True
```

In [12]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
```

```
        human_pred = np.array(list(map(dog_detector, tqdm(human_files_short, desc='Dog Detectio
        dog_pred = np.array(list(map(dog_detector, tqdm(dog_files_short, desc='Dog Detection In

        human_score = np.mean(human_pred)
        dog_score = np.mean(dog_pred)
```

```
Dog Detection In Human Files: 100%|| 100/100 [02:10<00:00,  1.29s/it]
Dog Detection In Dog Files  : 100%|| 100/100 [02:12<00:00,  1.30s/it]
```

```
In [15]: print(str(human_score*100)+"% Dogs Detected in Human Photos")
         print(str(dog_score*100)+"% Dogs Detected in Dog Photos")
```

```
0.0% Dogs Detected in Human Photos
100.0% Dogs Detected in Dog Photos
```

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? - 100% of human images were detected by the dog detector (presumably since the algo can only detect dogs, not human faces). Therefore, I show 0% accuracy for human face detection. **- What percentage of the images in `dog_files_short` have a detected dog?** - 100% of dogs were detected by the dog detector for the same reasons outlined above

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 [17]:
```
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes

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/')


#transform
data_transform = {'train': (transforms.Compose([
                            transforms.Resize(256),
                            transforms.RandomCrop(224),
                            transforms.ToTensor(),
                            transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229

                  'valid':(transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0
```

```
                    'test': (transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0


#set train/test/validation loads
train_data = datasets.ImageFolder(train_dir, transform=data_transform['train'])
validation_data = datasets.ImageFolder(valid_dir, transform=data_transform['valid'])
test_data = datasets.ImageFolder(test_dir, transform=data_transform['test'])

print('Number of training Images: ' + str(len(train_data)))
print('Number of test Images: ' + str(len(test_data)))
print('Number of Validation Images: ' + str(len(test_data)))
batch_size = 32
num_workers = 0

#define data loads
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worke
validation_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size,
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers


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

```
Number of training Images: 6680
Number of test Images: 836
Number of Validation Images: 836
```

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?**

```
- The images are first resized to 256 and then cropped to 224 in a random fasion in order to max
```

- **Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?**
  - I decided to augment data via normalization, random crops, and shuffling. This small number of augmentations was chosen as a result of the relativley steep training time after some trial and error.

Answer:

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [18]:
```
# 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
        self.conv1 = nn.Conv2d(3, 64, 3, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(64, 128, 3, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(128, 128, 3, padding = 1)
        self.conv4 = nn.Conv2d(128, 256, 3, padding = 1)
        self.conv5 = nn.Conv2d(256, 512, 3, padding = 1)

        self.pool = nn.MaxPool2d(2,2)

        self.fc1 = nn.Linear(7*7*512, 512)
        self.fc2 = nn.Linear(512, 133)

        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.pool(x)
        x = F.relu(self.conv4(x))
        x = self.pool(x)
        x = F.relu(self.conv5(x))
        x = x.view(-1, 512 * 7 * 7)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x


#-#-# You so NOT have to modify the code below this line. #-#-#

# instantiate the CNN
model_scratch = Net()
print(model_scratch)

# move tensors to GPU if CUDA is available
```

```
        if use_cuda:
            model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv5): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=25088, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer:

At first I defined my network based on VGG16 architecture - this however proved to be too large to train efficientlly, so from there I decreased the number of layers and added strides of 2 to the initial 2 layers to lead to further image compresssion. I decided to increase stride only on the first 2 layers since upper layers tend to "recognize" broader features. I chose my optimizer as Adam (amsgrad varient) since Adam typically starts with high learning rates and then decreases learning rates as the optimizer approaches the minima (this can speed up training and increase performance). AMSGRAD was chosen since I've read that it generally outperforms other optimizers with images/smaller data sets. Training epochs and batch size were then chosen based on experimentation.

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 [19]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = torch.optim.Adam(model_scratch.parameters(), lr=0.001, amsgrad=True
```

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 [20]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
def train(n_epochs, loaders, model, optimizer, criterion, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.inf

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):

            if use_cuda:
                data, target = data.cuda(), target.cuda()

            optimizer.zero_grad()

            output = model(data)

            loss = criterion(output, target)

            loss.backward()

            optimizer.step()

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

            if batch_idx % 100 == 0:
                print('Epoch: %d, Batch: %d, Loss: %.6f' % (epoch, batch_idx+1, train_l



            ## 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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
```

```
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss

            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1/(batch_idx + 1)) * (loss.data - valid_loss))


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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)
            print('Validation Loss Decreased ({:.6f} --> {:.6f}).  Model Saved...'.form
                valid_loss_min,
                valid_loss))

            valid_loss_min = valid_loss
    # return trained model
    return model
```

In [58]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, 'model_scratch.pt')
```

```
Epoch: 1, Batch: 1, Loss: 4.881490
Epoch: 1, Batch: 101, Loss: 4.891015
Epoch: 1, Batch: 201, Loss: 4.874722
Epoch Total: 1,         Training Loss: 4.871705         Validation Loss: 4.770620
Validation Loss Decreased (inf --> 4.770620).  Model Saved...
Epoch: 2, Batch: 1, Loss: 4.582980
Epoch: 2, Batch: 101, Loss: 4.723765
Epoch: 2, Batch: 201, Loss: 4.688519
Epoch Total: 2,         Training Loss: 4.684111         Validation Loss: 4.618574
Validation Loss Decreased (4.770620 --> 4.618574).  Model Saved...
Epoch: 3, Batch: 1, Loss: 4.645508
Epoch: 3, Batch: 101, Loss: 4.542374
Epoch: 3, Batch: 201, Loss: 4.523311
Epoch Total: 3,         Training Loss: 4.521351         Validation Loss: 4.445395
Validation Loss Decreased (4.618574 --> 4.445395).  Model Saved...
Epoch: 4, Batch: 1, Loss: 4.328767
Epoch: 4, Batch: 101, Loss: 4.347949
Epoch: 4, Batch: 201, Loss: 4.342152
Epoch Total: 4,         Training Loss: 4.340547         Validation Loss: 4.262448
```

```
Validation Loss Decreased (4.445395 --> 4.262448).  Model Saved...
Epoch: 5, Batch: 1, Loss: 3.844523
Epoch: 5, Batch: 101, Loss: 4.176630
Epoch: 5, Batch: 201, Loss: 4.185952
Epoch Total: 5,         Training Loss: 4.184273         Validation Loss: 4.116028
Validation Loss Decreased (4.262448 --> 4.116028).  Model Saved...
Epoch: 6, Batch: 1, Loss: 3.976613
Epoch: 6, Batch: 101, Loss: 4.026888
Epoch: 6, Batch: 201, Loss: 4.044595
Epoch Total: 6,         Training Loss: 4.044587         Validation Loss: 4.011809
Validation Loss Decreased (4.116028 --> 4.011809).  Model Saved...
Epoch: 7, Batch: 1, Loss: 3.952982
Epoch: 7, Batch: 101, Loss: 3.892328
Epoch: 7, Batch: 201, Loss: 3.892389
Epoch Total: 7,         Training Loss: 3.894991         Validation Loss: 3.962697
Validation Loss Decreased (4.011809 --> 3.962697).  Model Saved...
Epoch: 8, Batch: 1, Loss: 3.635346
Epoch: 8, Batch: 101, Loss: 3.730313
Epoch: 8, Batch: 201, Loss: 3.766437
Epoch Total: 8,         Training Loss: 3.762837         Validation Loss: 3.880400
Validation Loss Decreased (3.962697 --> 3.880400).  Model Saved...
Epoch: 9, Batch: 1, Loss: 3.537534
Epoch: 9, Batch: 101, Loss: 3.596453
Epoch: 9, Batch: 201, Loss: 3.625682
Epoch Total: 9,         Training Loss: 3.628198         Validation Loss: 3.763822
Validation Loss Decreased (3.880400 --> 3.763822).  Model Saved...
Epoch: 10, Batch: 1, Loss: 3.691286
Epoch: 10, Batch: 101, Loss: 3.487224
Epoch: 10, Batch: 201, Loss: 3.507011
Epoch Total: 10,         Training Loss: 3.511554         Validation Loss: 3.651515
Validation Loss Decreased (3.763822 --> 3.651515).  Model Saved...
Epoch: 11, Batch: 1, Loss: 3.054910
Epoch: 11, Batch: 101, Loss: 3.322381
Epoch: 11, Batch: 201, Loss: 3.386947
Epoch Total: 11,         Training Loss: 3.393826         Validation Loss: 3.685270
Epoch: 12, Batch: 1, Loss: 3.204538
Epoch: 12, Batch: 101, Loss: 3.290200
Epoch: 12, Batch: 201, Loss: 3.279422
Epoch Total: 12,         Training Loss: 3.285961         Validation Loss: 3.645249
Validation Loss Decreased (3.651515 --> 3.645249).  Model Saved...
Epoch: 13, Batch: 1, Loss: 3.136940
Epoch: 13, Batch: 101, Loss: 3.164048
Epoch: 13, Batch: 201, Loss: 3.168872
Epoch Total: 13,         Training Loss: 3.172271         Validation Loss: 3.630377
Validation Loss Decreased (3.645249 --> 3.630377).  Model Saved...
Epoch: 14, Batch: 1, Loss: 2.567816
Epoch: 14, Batch: 101, Loss: 3.020328
Epoch: 14, Batch: 201, Loss: 3.041727
```

```
Epoch Total: 14,         Training Loss: 3.046611         Validation Loss: 3.599572
Validation Loss Decreased (3.630377 --> 3.599572).  Model Saved...
Epoch: 15, Batch: 1, Loss: 2.968119
Epoch: 15, Batch: 101, Loss: 2.888113
Epoch: 15, Batch: 201, Loss: 2.927652
Epoch Total: 15,         Training Loss: 2.929636         Validation Loss: 3.555558
Validation Loss Decreased (3.599572 --> 3.555558).  Model Saved...
Epoch: 16, Batch: 1, Loss: 2.971072
Epoch: 16, Batch: 101, Loss: 2.790254
Epoch: 16, Batch: 201, Loss: 2.825051
Epoch Total: 16,         Training Loss: 2.834043         Validation Loss: 3.537908
Validation Loss Decreased (3.555558 --> 3.537908).  Model Saved...
Epoch: 17, Batch: 1, Loss: 3.413647
Epoch: 17, Batch: 101, Loss: 2.647681
Epoch: 17, Batch: 201, Loss: 2.701332
Epoch Total: 17,         Training Loss: 2.697863         Validation Loss: 3.601479
Epoch: 18, Batch: 1, Loss: 2.714215
Epoch: 18, Batch: 101, Loss: 2.569989
Epoch: 18, Batch: 201, Loss: 2.614389
Epoch Total: 18,         Training Loss: 2.618150         Validation Loss: 3.483469
Validation Loss Decreased (3.537908 --> 3.483469).  Model Saved...
Epoch: 19, Batch: 1, Loss: 2.501984
Epoch: 19, Batch: 101, Loss: 2.483154
Epoch: 19, Batch: 201, Loss: 2.528270
Epoch Total: 19,         Training Loss: 2.525058         Validation Loss: 3.533379
Epoch: 20, Batch: 1, Loss: 2.389571
Epoch: 20, Batch: 101, Loss: 2.394791
Epoch: 20, Batch: 201, Loss: 2.446010
Epoch Total: 20,         Training Loss: 2.452153         Validation Loss: 3.682560
```

```
In [21]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

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 [35]: 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: 3.422123


Test Accuracy: 21% (176/836)
```

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 [22]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

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 [26]:
```python
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.vgg16(pretrained=True)

print(model_transfer)
```

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

```
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```
In [27]: for param in model_transfer.features.parameters():
             param.require_grad = False
```

```
In [28]: import torch.nn as nn

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

         print(model_transfer.classifier[6].in_features)
         print(model_transfer.classifier[6].out_features)
```

```
4096
133
```

```
In [29]: model_transfer_FC = model_transfer.classifier[6].parameters()

         for param in model_transfer_FC:
             param.requires_grad = True
```

```
In [30]: use_cuda = torch.cuda.is_available()
         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:

Step 1 - first I printed out the structure of the VGG16 pretrained model. This architecture was suitable since it worked decently well already with the convolutional model and already was trained for the breeds in question.

Step 2 - all weights were locked on their trained values to prevent those weights from being overwritten.

Step 3 - the last linear layer was replaced with a softmax with 133 categories rather then the original 1000.

Step 4 - this new layer was then flagged for retraining in order to recognize the 133 dog breeds.

Step 5 - cross entropy loss was chosen as the loss metric and the amsgrad variant of Adam was chosen for gradient decscent. cross entropy is standard for loss, while amsgrad version of adam was shown to outperform adam and other vairants for smaller datasets such as the data used to train for dog breeds.

Step 6 - training of the final layer was conducted for 5 epochs - less training was required for this instance relative to the manually defined model since it already trained conv oand other fc layers with far more data than what we have at our disposal. So I took advantage of that fact and trained fewer epochs while only keeping new weights that led to decreases in validation loss.

Step 7 - accuaracy was tested and revealed to be 85%

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 [31]: criterion_transfer = nn.CrossEntropyLoss()

         ### TODO: select optimizer
         optimizer_transfer = torch.optim.Adam(model_transfer.classifier[6].parameters(), lr=0.0
```

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 [32]: train(5, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, 'mod
```

```
Epoch: 1, Batch: 1, Loss: 5.135298
Epoch: 1, Batch: 101, Loss: 1.744092
Epoch: 1, Batch: 201, Loss: 1.259885
Epoch Total: 1,         Training Loss: 1.238765         Validation Loss: 0.482042
Validation Loss Decreased (inf --> 0.482042).  Model Saved...
Epoch: 2, Batch: 1, Loss: 0.302673
Epoch: 2, Batch: 101, Loss: 0.551186
Epoch: 2, Batch: 201, Loss: 0.535305
Epoch Total: 2,         Training Loss: 0.536285         Validation Loss: 0.420898
Validation Loss Decreased (0.482042 --> 0.420898).  Model Saved...
Epoch: 3, Batch: 1, Loss: 0.215790
Epoch: 3, Batch: 101, Loss: 0.441955
Epoch: 3, Batch: 201, Loss: 0.453088
Epoch Total: 3,         Training Loss: 0.457497         Validation Loss: 0.423357
Epoch: 4, Batch: 1, Loss: 0.083532
Epoch: 4, Batch: 101, Loss: 0.379233
Epoch: 4, Batch: 201, Loss: 0.414239
Epoch Total: 4,         Training Loss: 0.418266         Validation Loss: 0.406183
Validation Loss Decreased (0.420898 --> 0.406183).  Model Saved...
Epoch: 5, Batch: 1, Loss: 0.237342
Epoch: 5, Batch: 101, Loss: 0.348140
Epoch: 5, Batch: 201, Loss: 0.381872
Epoch Total: 5,         Training Loss: 0.385501         Validation Loss: 0.420060
```

```
Out[32]: VGG(
           (features): Sequential(
             (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (1): ReLU(inplace)
             (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
             (3): ReLU(inplace)
             (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
        (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (6): ReLU(inplace)
        (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (8): ReLU(inplace)
        (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (13): ReLU(inplace)
        (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (15): ReLU(inplace)
        (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (18): ReLU(inplace)
        (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (20): ReLU(inplace)
        (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (22): ReLU(inplace)
        (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
        (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (25): ReLU(inplace)
        (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (27): ReLU(inplace)
        (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (29): ReLU(inplace)
        (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      )
      (classifier): Sequential(
        (0): Linear(in_features=25088, out_features=4096, bias=True)
        (1): ReLU(inplace)
        (2): Dropout(p=0.5)
        (3): Linear(in_features=4096, out_features=4096, bias=True)
        (4): ReLU(inplace)
        (5): Dropout(p=0.5)
        (6): Linear(in_features=4096, out_features=133, bias=True)
      )
    )
```

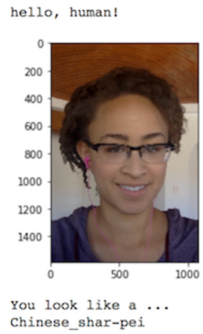
In [33]:
```
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

In [36]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.481520
```



Sample Human Output

```
Test Accuracy: 85% (712/836)
```

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

         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset
         dog_test_files = np.array(glob("/data/dog_images/test/*/*"))
         def predict_breed_transfer(img_path, model):
             model = model.cpu()
             model.eval()
             return class_names[torch.max(model(load_image(img_path)), 1)[1]]
```

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!

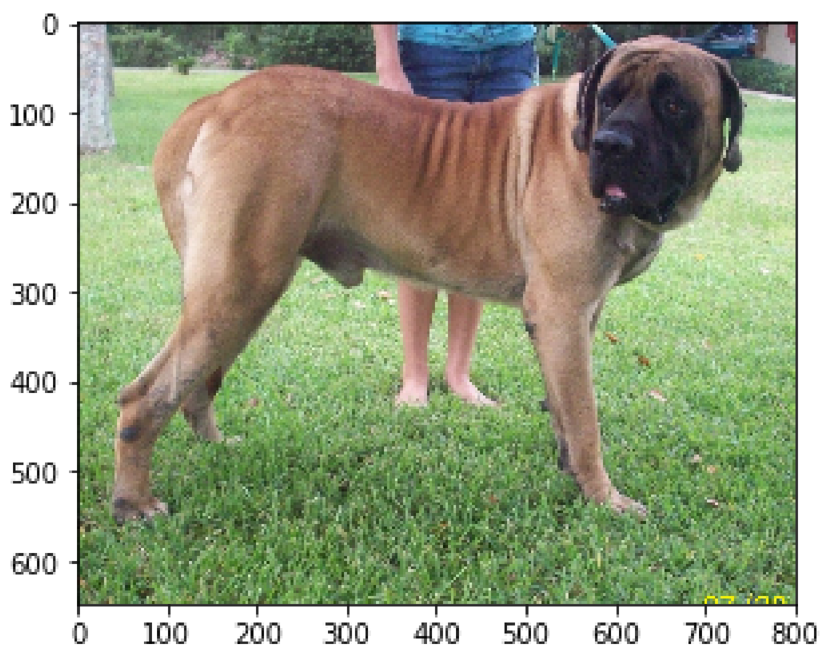
1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [38]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
```

```
def plot_image(img_path):
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    return plt.show()

def run_app(img_path, model):
    ## handle cases for a human face, dog, and neither
    if dog_detector(img_path):
        print("Dog Detected")
        plot_image(img_path)
        print("I predict that this dog is a " + predict_breed_transfer(img_path, model)
        print("---------------------------------------------------------------------------
    elif face_detector(img_path):
        print("Human Detected")
        plot_image(img_path)
        print("You look like a " + predict_breed_transfer(img_path, model))
        print("---------------------------------------------------------------------------
    else:
        print("Neither Human or Dog Detected!")
        print("---------------------------------------------------------------------------
```

In [39]: run_app(dog_files[0], model_transfer)

```
Dog Detected
```



```
I predict that this dog is a Bullmastiff
-----------------------------------------------------------------------
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

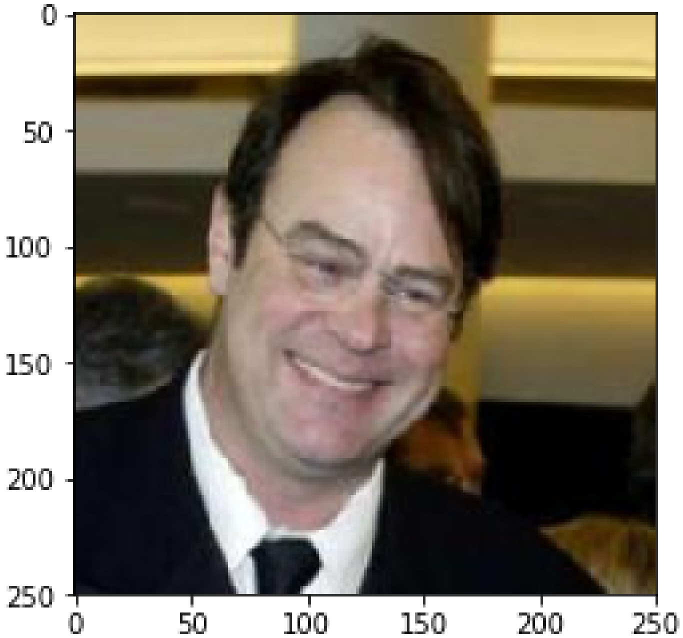
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

The output was better than I expected given that the manually created convolutional network was so difficult to train and get a high accuracy with. A few possible points of improvement focus around 1) hyperparameter tuning, 2) number of epochs trained, and the method used for gradient descent. I imagine that training beyond 5 epochs will only serve to further improve the reliability of the modified VGG16 model.

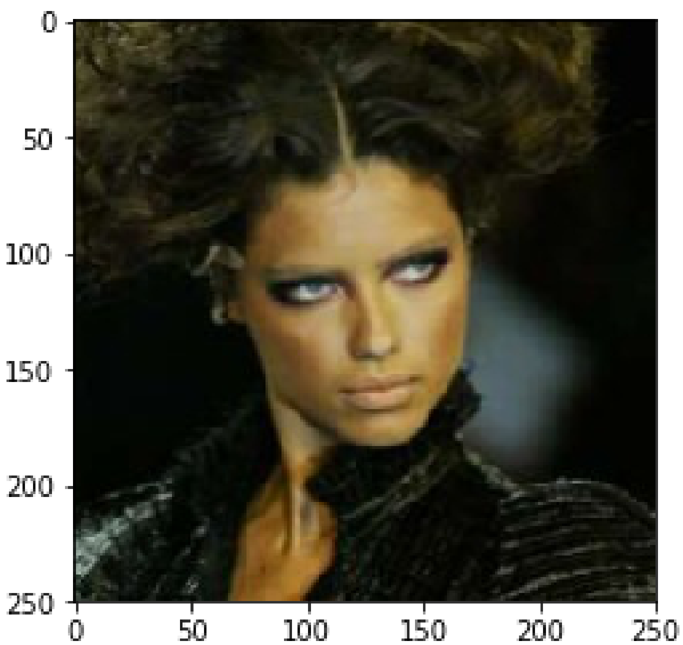
```
In [43]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         for file in np.hstack((human_files[0:100:20], dog_test_files[99:200:20])):
             run_app(file, model_transfer)
```

```
Human Detected
```

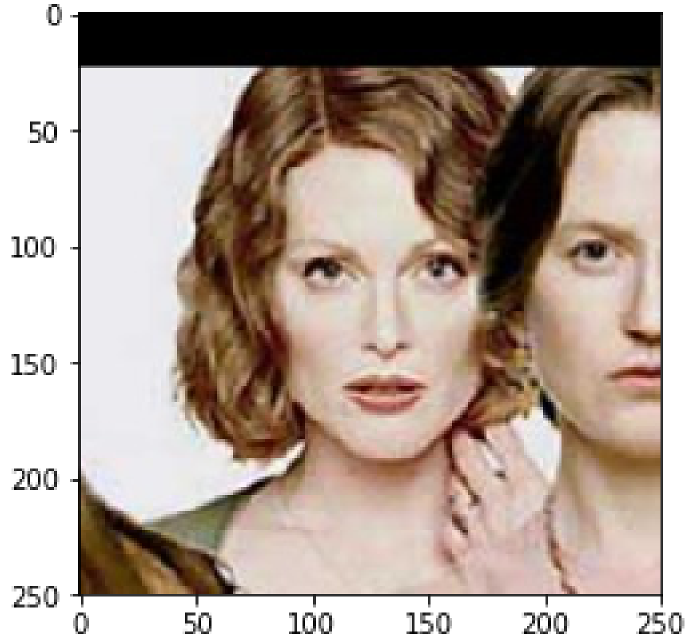


You look like a Chihuahua

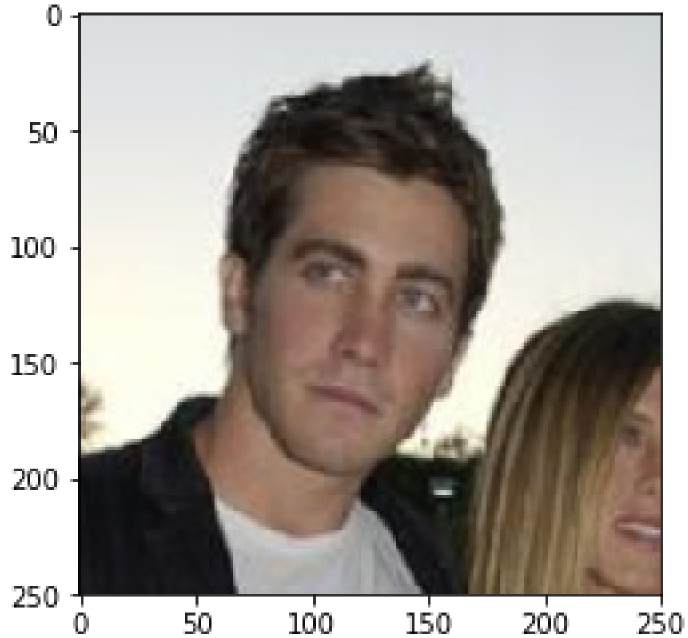
Human Detected



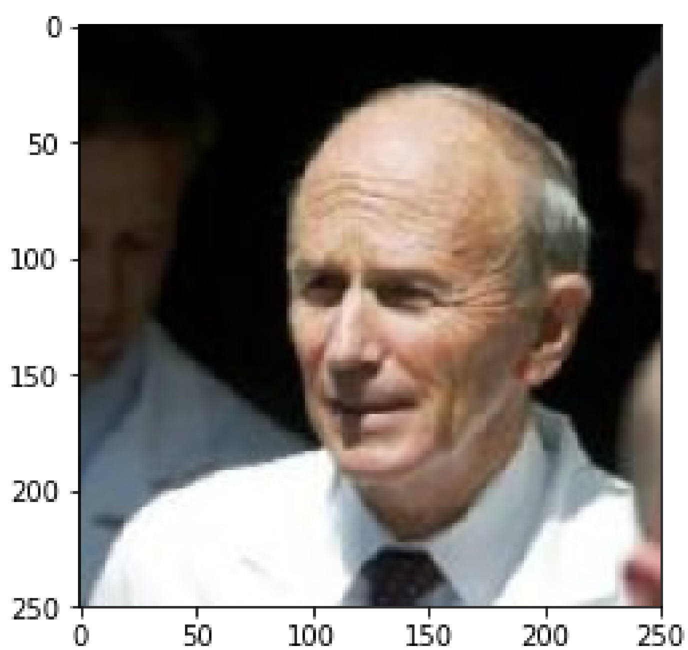
```
You look like a Afghan hound
------------------------------------------------------------------------
Human Detected
```




```
You look like a Afghan hound
------------------------------------------------------------------------
Human Detected
```

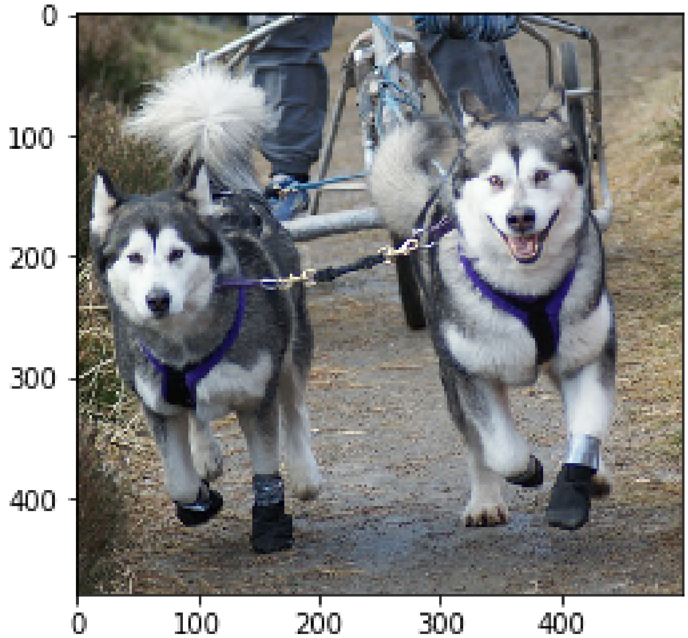



```
You look like a Silky terrier
------------------------------------------------------------------------
Human Detected
```

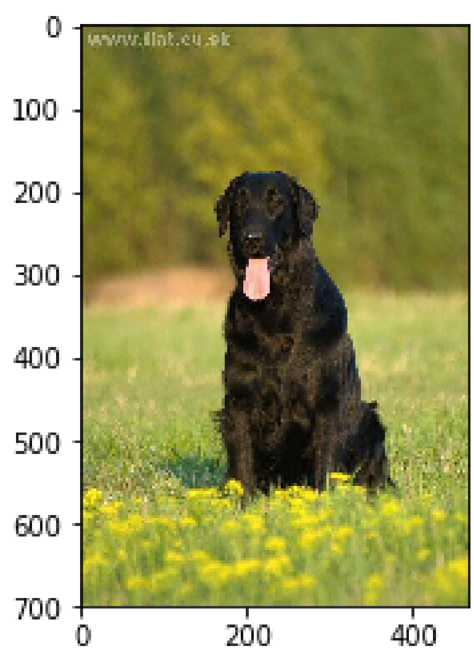


```
You look like a Welsh springer spaniel
--------------------------------------------------------------------------
Dog Detected
```


```
I predict that this dog is a Bluetick coonhound
--------------------------------------------------------------------------
Dog Detected
```



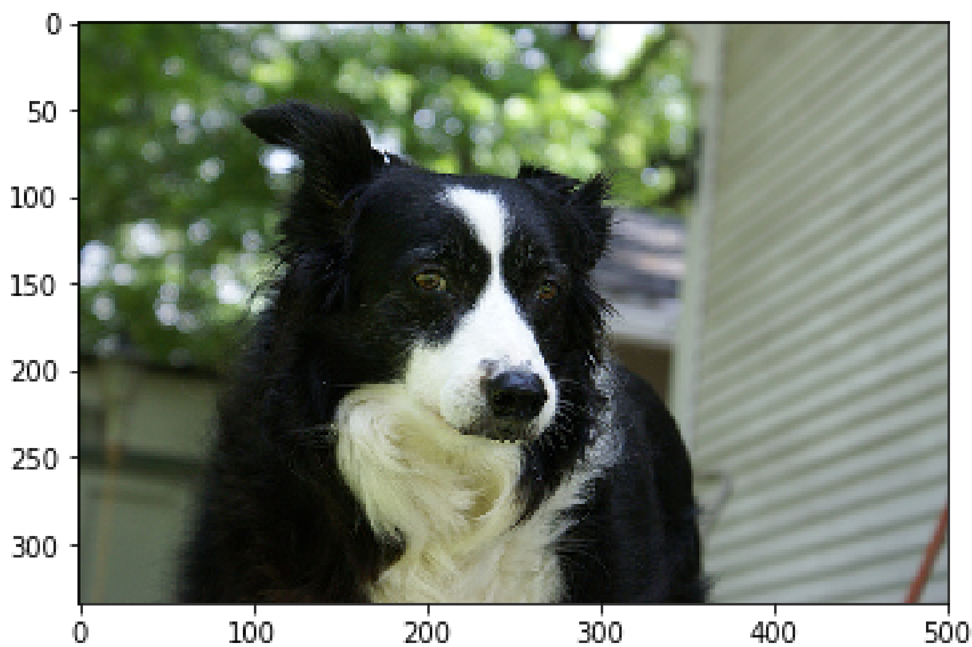

```
I predict that this dog is a Alaskan malamute
-----------------------------------------------------------------------
Dog Detected
```



```
I predict that this dog is a Flat-coated retriever
------------------------------------------------------------------------
Dog Detected
```


```
I predict that this dog is a Chihuahua
------------------------------------------------------------------------
Dog Detected
```


```
I predict that this dog is a Black and tan coonhound
------------------------------------------------------------------------
Dog Detected
```



```
I predict that this dog is a Border collie
--------------------------------------------------------------------------
```

In [ ]: