dog_app

April 22, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

In [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

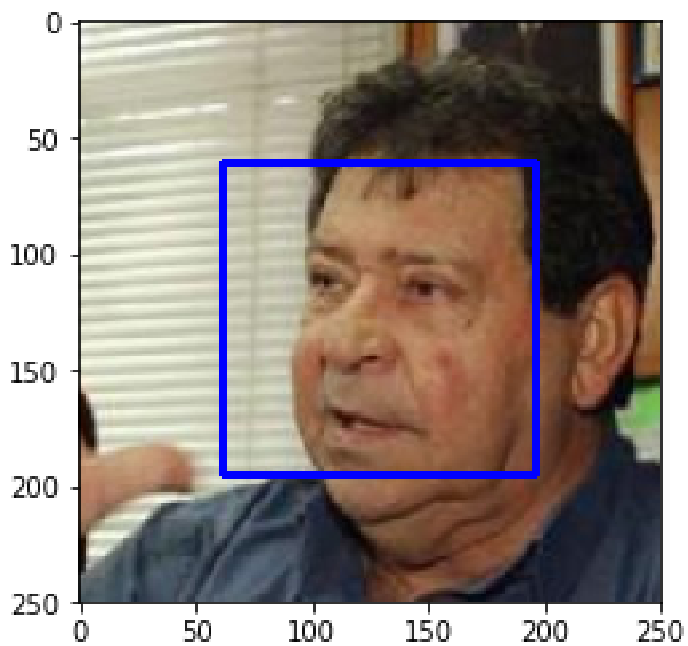
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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
from random import shuffle


%matplotlib inline

#shuffle human files
shuffle(human_files)

# 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)
```

```python
# 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:

To test the performance of the `face_detector` function, we got theses results.

- 100 % of the first 100 images in `human_files` have a detected human face? :
- 17 % of the first 100 images in `dog_files` have a detected human face? :

We see that our algorithm is not `perfect`and falls short of the goal(100% of human images with a detected face and 0% of dog images with a detected face).

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

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

        percentage_of_human_faces = 0
        percentage_of_dog_faces = 0

        for path in tqdm(human_files_short):
            if face_detector(path):
                percentage_of_human_faces += 1

        for path in tqdm(dog_files_short):
```

```
            if face_detector(path):
                percentage_of_dog_faces += 1

        print(f'Percentage of Human faces detected {percentage_of_human_faces:0.2f}')
        print(f'Percentage of Dog faces detected {percentage_of_dog_faces:0.2f}')
```

```
100%|| 100/100 [00:02<00:00, 35.50it/s]
100%|| 100/100 [00:29<00:00,  3.37it/s]

Percentage of Human faces detected 98.00
Percentage of Dog faces detected 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 [7]: ### (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 [5]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 95873608.13it/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 [6]: from PIL import Image
        import torchvision.transforms as transforms
        from torch.autograd import Variable

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

            Args:
                img_path: path to an image

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

            # Open image from path using PIL
            img = Image.open(img_path)

            #  Normalization using the documentation
            normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])

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

            img_tensor = preprocessing(img).float()
            batch_tensor = torch.unsqueeze(img_tensor, 0)
```

```
    # The input to the network needs to be an autograd Variable

    batch_tensor = Variable(batch_tensor)
    if use_cuda:
        batch_tensor = Variable(batch_tensor.cuda())

    VGG16.eval()

    output = VGG16(batch_tensor)
    output = output.cpu()

    index = output.data.numpy().argmax()
    return index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [7]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):

            vgg16_index = VGG16_predict(img_path)

            if (vgg16_index >= 151) & (vgg16_index <= 268):
                return True
            return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: - 1% of the images in `human_files_short` have a detected dog
- 100% of the images in `dog_files_short` have a detected dog

```
In [8]: ### Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        percentage_of_human_faces = 0
        percentage_of_dog_faces = 0

        for path in tqdm(human_files_short) :
            if dog_detector(path):
```

```
            percentage_of_human_faces += 1

    for path in tqdm(dog_files_short) :
        if dog_detector(path):
            percentage_of_dog_faces += 1

    print(f'Percentage human faces detected: {percentage_of_human_faces} %')
    print(f'Percentage dog faces detected: {percentage_of_dog_faces:} %')
```

```
100%|| 100/100 [00:03<00:00, 29.50it/s]
100%|| 100/100 [00:04<00:00, 25.51it/s]

Percentage human faces detected: 1 %
Percentage dog faces detected: 100 %
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [9]:
```
import os
from torchvision import datasets

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

batch_size = 20
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
num_workers = 0

# Transform training and testing and validation
# Transform for training loader
transform_train = transforms.Compose([
    transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Normalize(mean=mean, std=std)
    ])
# Transform for testing and validation loaders
transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
```

```
            transforms.ToTensor(),
            transforms.Normalize(mean=mean, std=std)
        ])

    trainingset = datasets.ImageFolder('/data/dog_images/train', transform=transform_train)
    trainingloader = torch.utils.data.DataLoader(trainingset, batch_size=batch_size, shuffle

    testset = datasets.ImageFolder('/data/dog_images/test', transform=transform)
    testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True, n

    validationset = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
    validationloader = torch.utils.data.DataLoader(validationset, batch_size=batch_size, shu
```

In [10]:
```
assert(len(trainingset.classes) == 133)
image_datasets = {
    'train': trainingset,
    'valid':validationset,
    'test':testset
}
classes = trainingset.classes
```

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: Here is my procedure for preprocessing the data. - I decided to resize all images in my dataset (training, test, validation). I used the transforms to resize the training data randomly to 224. Then I used random flipping transformation before rotating images. Finaly I transform to tensors and applied normalization with parameters as recommended in Api documentation. - I did the same operation for testing data and validation using the same transform function parameters. I augmented the training dataset through translations and rotation.

In [15]:
```
def imshow(img):
        img = np.transpose(img, (1, 2, 0))
        img = img * std + mean  # unnormalize
        plt.imshow(np.clip(img, 0, 1))  # convert from Tensor image


images, labels = next(iter(trainingloader))
images = images.numpy()
# convert images to numpy for display
# plot the images in the batch, along with the corresponding labels

fig = plt.figure(figsize=(20, 4))
# display 20 images
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    imshow(images[idx])
    ax.set_title((classes[labels[idx]].split('.')[1])[:8] + '...')
```

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

         #number of classes
         n = len(classes)
         # 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)
                 # convolutional layer (sees 28x28x64 tensor)
                 self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                 # convolutional layer (sees 14x14x128 tensor)
                 self.conv5 = nn.Conv2d(128, 256, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)

                 # linear layer (256 * 7 * 7 -> 512)
                 self.fc1 = nn.Linear(256 * 7 * 7, 512)
                 # linear layer (512 -> 133)
                 self.fc2 = nn.Linear(512, n)
                 # dropout layer (p=0.4)
                 self.dropout = nn.Dropout(0.4)

                 # Batch normalization
                 self.bn1 = nn.BatchNorm2d(224, 3)
                 self.bn2 = nn.BatchNorm2d(16)
```

```
        self.bn3 = nn.BatchNorm2d(32)
        self.bn4 = nn.BatchNorm2d(64)
        self.bn5 = nn.BatchNorm2d(128)
        self.bn6 = nn.BatchNorm2d(256)

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.bn2(x)
        x = self.pool(F.relu(self.conv2(x)))
        x = self.bn3(x)
        x = self.pool(F.relu(self.conv3(x)))
        x = self.bn4(x)
        x = self.pool(F.relu(self.conv4(x)))
        x = self.bn5(x)
        x = self.pool(F.relu(self.conv5(x)))
        x = self.bn6(x)
        #print(x.shape)
        # flatten image input
        x = x.view(-1, 256 * 7 * 7)
        # 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 = self.fc2(x)
        return x

#-#-# You do 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 = model_scratch.cuda()
```

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

Answer:

- I created a CNN composed of five convolution layers and two fully connected layers.
- Each of theses conv. layers is follows with a max polling layer.
- Then I activated layers with relu except the last layer.
- I used batch normalized after max pooling and two dropout before and after the first connected layers.

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

         ### Select loss function
         criterion_scratch = nn.CrossEntropyLoss()

         ### Select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001)
```

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 [26]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         epochs = 20

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

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

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

                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()

                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)

                     # calculate the batch loss
```

```
            loss = criterion(output, target)

            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            valid_loss = valid_loss + (1 / (batch_idx + 1)) * (loss.data - valid_loss)

        # calculate average losses
        train_loss = train_loss / len(loaders['train'])
        valid_loss = valid_loss / len(loaders['valid'])

        # 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 from {:.6f} ---> {:.6f} Saving the model .
            valid_loss_min,
            valid_loss
            ))
        #torch.save(model.state_dict(), 'model_scratch.pt')
        valid_loss_min = valid_loss
        torch.save(model.state_dict(), save_path)

    # return trained model
    return model
```

```python
loaders_scratch = {
    'train': trainingloader,
    'valid': validationloader,
    'test': testloader
}

# train the model
model_scratch = train(epochs, 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: 0.013102         Validation Loss: 0.101832
Validation Loss decreased from inf ---> 0.101832 Saving the model ...
Epoch: 2         Training Loss: 0.013086         Validation Loss: 0.101129
Validation Loss decreased from 0.101832 ---> 0.101129 Saving the model ...
Epoch: 3         Training Loss: 0.013006         Validation Loss: 0.100234
Validation Loss decreased from 0.101129 ---> 0.100234 Saving the model ...
Epoch: 4         Training Loss: 0.012943         Validation Loss: 0.099922
Validation Loss decreased from 0.100234 ---> 0.099922 Saving the model ...
Epoch: 5         Training Loss: 0.012873         Validation Loss: 0.099359
Validation Loss decreased from 0.099922 ---> 0.099359 Saving the model ...
Epoch: 6         Training Loss: 0.012788         Validation Loss: 0.098324
Validation Loss decreased from 0.099359 ---> 0.098324 Saving the model ...
Epoch: 7         Training Loss: 0.012773         Validation Loss: 0.097740
Validation Loss decreased from 0.098324 ---> 0.097740 Saving the model ...
Epoch: 8         Training Loss: 0.012668         Validation Loss: 0.097196
Validation Loss decreased from 0.097740 ---> 0.097196 Saving the model ...
Epoch: 9         Training Loss: 0.012585         Validation Loss: 0.096853
Validation Loss decreased from 0.097196 ---> 0.096853 Saving the model ...
Epoch: 10         Training Loss: 0.012525         Validation Loss: 0.095933
Validation Loss decreased from 0.096853 ---> 0.095933 Saving the model ...
Epoch: 11         Training Loss: 0.012531         Validation Loss: 0.095505
Validation Loss decreased from 0.095933 ---> 0.095505 Saving the model ...
Epoch: 12         Training Loss: 0.012454         Validation Loss: 0.095110
Validation Loss decreased from 0.095505 ---> 0.095110 Saving the model ...
Epoch: 13         Training Loss: 0.012395         Validation Loss: 0.094641
Validation Loss decreased from 0.095110 ---> 0.094641 Saving the model ...
Epoch: 14         Training Loss: 0.012339         Validation Loss: 0.093910
Validation Loss decreased from 0.094641 ---> 0.093910 Saving the model ...
Epoch: 15         Training Loss: 0.012286         Validation Loss: 0.093189
Validation Loss decreased from 0.093910 ---> 0.093189 Saving the model ...
Epoch: 16         Training Loss: 0.012210         Validation Loss: 0.093122
Validation Loss decreased from 0.093189 ---> 0.093122 Saving the model ...
Epoch: 17         Training Loss: 0.012198         Validation Loss: 0.092687
Validation Loss decreased from 0.093122 ---> 0.092687 Saving the model ...
```

```
Epoch: 18         Training Loss: 0.012147         Validation Loss: 0.091951
Validation Loss decreased from 0.092687 ---> 0.091951 Saving the model ...
Epoch: 19         Training Loss: 0.012062         Validation Loss: 0.091384
Validation Loss decreased from 0.091951 ---> 0.091384 Saving the model ...
Epoch: 20         Training Loss: 0.011975         Validation Loss: 0.091615
```

1.1.11 (IMPLEMENTATION) Test the Model

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

In [14]:
```
def test(loaders, model, criterion, use_cuda):

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

    model.eval()
    for batch_idx, (data, target) in enumerate(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: 3.797801


Test Accuracy: 14% (121/836)
```

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](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [15]:
```
## TODO: Specify data loaders
## TODO: Specify data loaders
batch_size = 20
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]
num_workers = 0

# Transform training and testing and validation
# Transform for training loader
transform_train = transforms.Compose([
    transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(10),
        transforms.ToTensor(),
        transforms.Normalize(mean=mean, std=std)
    ])
# Transform for testing and validation loaders
transform = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=mean, std=std)
    ])

trainingset = datasets.ImageFolder('/data/dog_images/train', transform=transform_train)
trainingloader = torch.utils.data.DataLoader(trainingset, batch_size=batch_size, shuffl

testset = datasets.ImageFolder('/data/dog_images/test', transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=True,

validationset = datasets.ImageFolder('/data/dog_images/valid', transform=transform)
validationloader = torch.utils.data.DataLoader(validationset, batch_size=batch_size, sh
```

In [16]:
```
loaders_transfer = {
    'train': trainingloader,
```

```
        'valid': validationloader,
        'test': testloader
    }
```

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


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

         # Freeze training for all "features" layers
         for param in model_transfer.features.parameters():
             param.requires_grad = False


         in_size = model_transfer.classifier[6].in_features
         out_size = n

         # FC Linear Layer
         fc1 = nn.Linear(in_size, out_size)
         # Replace model_transfer last fc
         model_transfer.classifier[6] = fc1

         #print(model_transfer)
         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: Transfer learning is a popular method in computer vision because it allows us to build accurate models in a `timesaving way`. For this current problem freezing the convolutional base is a good idea. The main idea is to keep the convolutional base in its original form and then use its outputs to feed the classifier(predict one of dog breed classes). In the original classifier we have classes from 0 to 999.

- This approch is suitable for the current problem because dataset is small, and/or pre-trained model solves a problem very similar to the one we want to solve. Here are steps to final CNN architecture: 1 - Freeze training for all "features" layers 2 - Replace last fully connected layer with new FC Linear Layer

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [19]: n_epochs = 10

         # train the model
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

         # 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: 0.012912         Validation Loss: 0.070377
Validation Loss decreased from inf ---> 0.070377 Saving the model ...
Epoch: 2         Training Loss: 0.008410         Validation Loss: 0.032263
Validation Loss decreased from 0.070377 ---> 0.032263 Saving the model ...
Epoch: 3         Training Loss: 0.005900         Validation Loss: 0.020426
Validation Loss decreased from 0.032263 ---> 0.020426 Saving the model ...
Epoch: 4         Training Loss: 0.004807         Validation Loss: 0.016045
Validation Loss decreased from 0.020426 ---> 0.016045 Saving the model ...
Epoch: 5         Training Loss: 0.004484         Validation Loss: 0.013896
Validation Loss decreased from 0.016045 ---> 0.013896 Saving the model ...
Epoch: 6         Training Loss: 0.004071         Validation Loss: 0.012972
Validation Loss decreased from 0.013896 ---> 0.012972 Saving the model ...
Epoch: 7         Training Loss: 0.003780         Validation Loss: 0.012122
Validation Loss decreased from 0.012972 ---> 0.012122 Saving the model ...
Epoch: 8         Training Loss: 0.003761         Validation Loss: 0.011297
Validation Loss decreased from 0.012122 ---> 0.011297 Saving the model ...
Epoch: 9         Training Loss: 0.003530         Validation Loss: 0.010781
Validation Loss decreased from 0.011297 ---> 0.010781 Saving the model ...
Epoch: 10         Training Loss: 0.003444         Validation Loss: 0.010442
Validation Loss decreased from 0.010781 ---> 0.010442 Saving the model ...
```

1.1.16 (IMPLEMENTATION) Test the Model

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

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

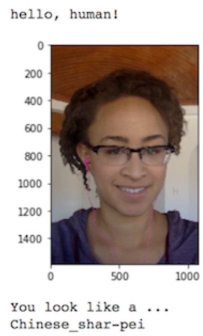
```
Test Loss: 0.491847


Test Accuracy: 84% (710/836)
```

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 [24]:
```
### 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 image_datasets['train'].classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    # Open image from path using PIL
    img = Image.open(img_path)

    #  Normalization using the documentation
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                     std=[0.229, 0.224, 0.225])

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

    img_tensor = preprocessing(img).float()
    batch_tensor = torch.unsqueeze(img_tensor, 0)

    # The input to the network needs to be an autograd Variable

    batch_tensor = Variable(batch_tensor)
    if use_cuda:
        batch_tensor = Variable(batch_tensor.cuda())

    model_transfer.eval()

    output = model_transfer(batch_tensor)
    output = output.cpu()

    index = output.data.numpy().argmax()

    return index, class_names[index], image_datasets['train'].classes[index]
```

Step 5: Write your Algorithm



Sample Human Output

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 [25]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def show_thumbnail(img_path):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    detect_human  =  face_detector(img_path)
    detect_dog = dog_detector(img_path)

    index, dog, breed = predict_breed_transfer(img_path)
    if detect_dog:
        print('A dog detected')
        # display the image, along with bounding box
        show_thumbnail(img_path)
        print('A dog breed is a {}'.format(breed[4:].replace("_", " ")))
    elif detect_human:
            print('Hello, Human!')
            show_thumbnail(img_path)
            print('You look like a {}'.format(breed[4:].replace("_", " ")))
```

```
        else:
            # Plot image of neither dog or human detected
            print('Sorry nothing detected in this image!')
            show_thumbnail(img_path)
            print('Please choose another image with dog or human')
```

Step 6: Test Your Algorithm
In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement) The output is worse than what I expected.

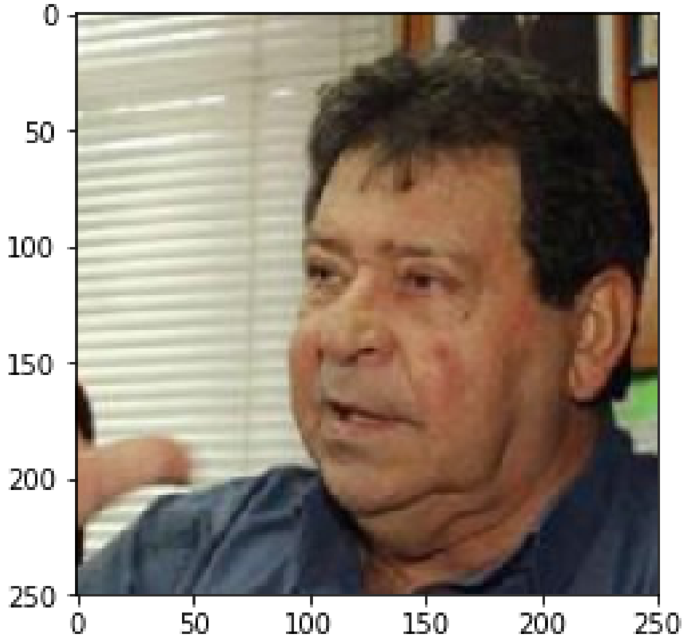
Find the best learning rate during training, I trained with (0.001) withoud trying others possible values

Augment the number of training epochs, I used 10 epochs for training for 79% accuracy

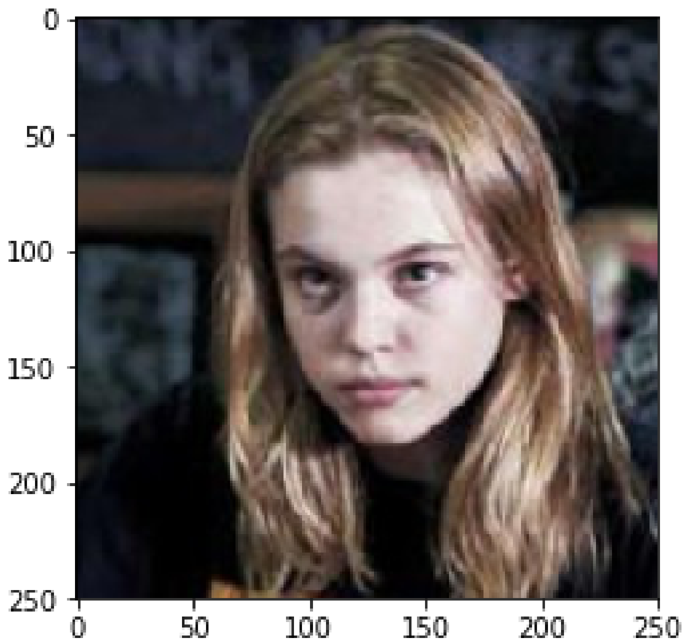
Augmend data, add more transformations in images.

```
In [26]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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

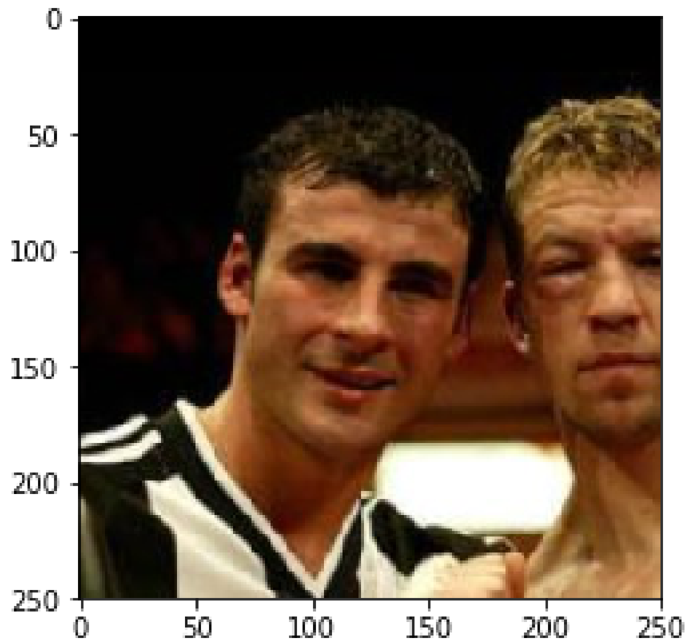
```
Hello, Human!
```



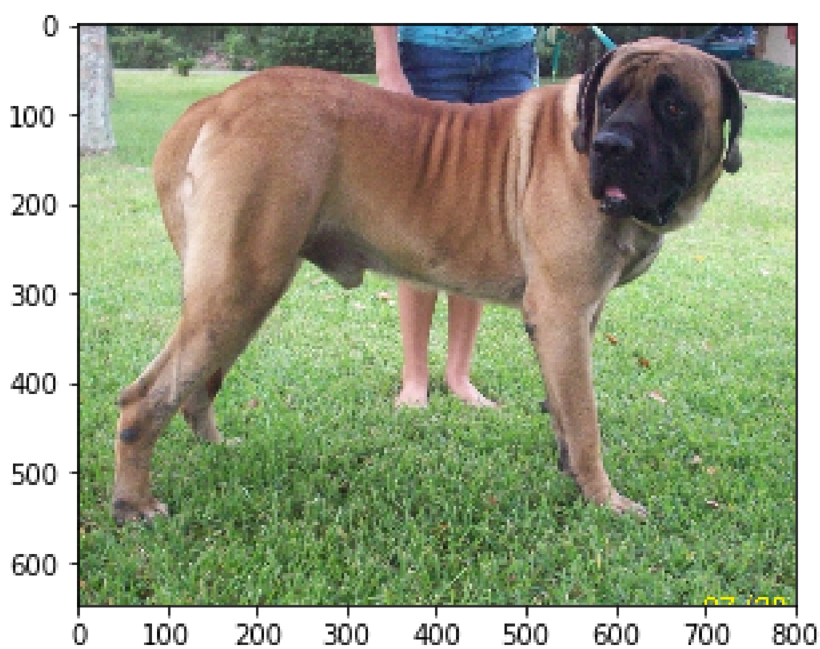

```
You look like a Silky terrier
Hello, Human!
```



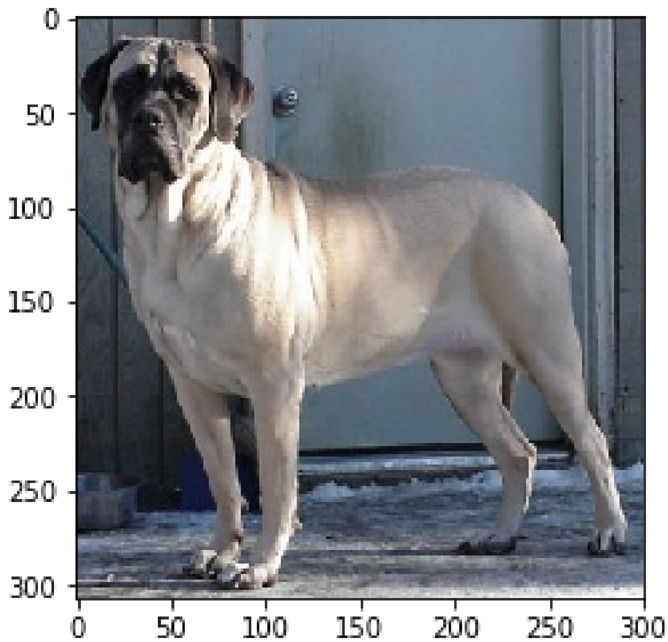
You look like a Cavalier king charles spaniel
Hello, Human!



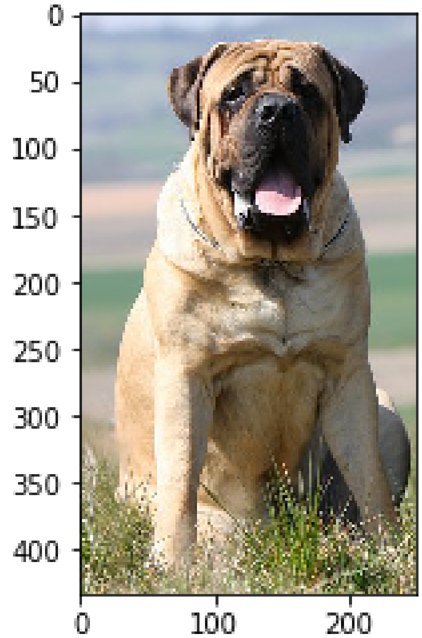
You look like a Beagle
A dog detected



```
A dog breed is a Bullmastiff
A dog detected
```




```
A dog breed is a Bullmastiff
A dog detected
```




```
A dog breed is a Bullmastiff
```

In [ ]: