dog_app

September 10, 2019

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 [40]: 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

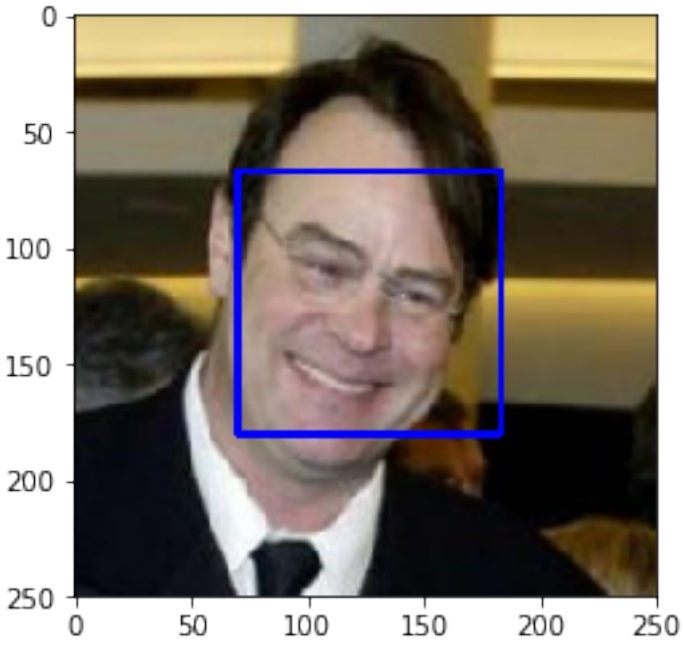
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [41]: 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 [42]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

- Percentage of first 100 images in human_files which detected a human face: 98%
- Percentage of first 100 images in dog_files which detected a human face: 17%

```
In [43]: 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.

         human_true = sum(1 for file in human_files_short if face_detector(file))
         dog_false = sum(1 for file in dog_files_short if face_detector(file))

         print("Percentage of human faces correctly detected as human face: " + str(human_true/1
         print("Percentage of dog faces incorrectly detected as human face: " + str(dog_false/10
```

```
Percentage of human faces correctly detected as human face: 0.98
Percentage of dog faces incorrectly detected as human face: 0.17
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make

use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
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 [44]: import torch
         import torchvision.models as models

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

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

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

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

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

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

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

```
In [45]: from PIL import Image
         import torchvision.transforms as transforms

         def VGG16_predict(img_path):
```

```
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = Image.open(img_path)
    transform_pipeline = transforms.Compose([transforms.Resize((224,224)),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                                  std=[0.229, 0.224, 0.225]
    img = transform_pipeline(img)
    img = img.unsqueeze(0)
    img = img.to('cuda')
    VGG16.eval()
    prediction = VGG16(img)
    prediction = prediction.to('cpu')
    prediction = prediction.data.numpy().argmax()
    return prediction # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [46]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             if VGG16_predict(img_path) >= 151 and VGG16_predict(img_path) <= 268:
                 return True
             else:
                 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: - Percentage of images in human_files_short which detected a dog: 0% - Percentage of images in dog_files_short which detected a dog: 100%

```
In [47]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short
         human_files_detected_dog = sum(1 for file in human_files_short if dog_detector(file))
         dog_files_detected_dog = sum(1 for file in dog_files_short if dog_detector(file))
         print("Percentage of images in human_files_short which detected dog: " +str(human_files
         print("Percentage of images in dog_files_short which detected dog: " +str(dog_files_det

Percentage of images in human_files_short which detected dog: 0.0
Percentage of images in dog_files_short which detected dog: 1.0
```

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`.

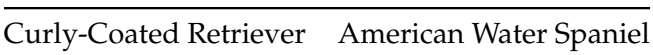
```
In [ ]: ### (Optional)
        ### TODO: Report the performance of another pre-trained network.
        ### Feel free to use as many code cells as needed.
```

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.

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 [9]: data_dir = '/data/dog_images'
        train_dir = data_dir + '/train'
        valid_dir = data_dir + '/valid'
        test_dir = data_dir + '/test'
```

```
In [10]: import os
         from torchvision import datasets, transforms

         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         train_transforms = transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.RandomRotation(30),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])

         valid_transforms = transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])
```

```
test_transforms = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

# TODO: Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform = train_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform = valid_transforms)
test_data = datasets.ImageFolder(test_dir, transform = test_transforms)

# TODO: Using the image datasets and the trainforms, define the dataloaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=32)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32)

loaders_scratch = {'train': trainloader, 'valid': validloader, 'test': testloader}
```

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: - My code resizes all images to 256x256 pixels and then center crops them to reduce them to 224x224 pixels, which is the size of the input tensor. This transformation is applied to all train, validation and test datasets - For training dataset in addition to resizing and cropping, I also used additional transformations like random horizontal flips and random rotation by 30 degrees. The aim was to augment the training dataset, so This hat it generalizes better and gives better accuracy on the testing dataset.

1.1.8 (IMPLEMENTATION) Model Architecture

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

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

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # convolutional layer
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 # convolutional layer
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # convolutional layer
```

```
                self.conv4 = nn.Conv2d(64, 128, 3, padding=1)
                # max pooling layer
                self.pool = nn.MaxPool2d(2, 2)
                # linear layer
                self.fc1 = nn.Linear(14 * 14 * 128, 1000)
                # linear layer
                self.fc2 = nn.Linear(1000, 500)
                # linear layer
                self.fc3 = nn.Linear(500, 133)
                # dropout layer (p=0.20)
                self.dropout = nn.Dropout(0.20)

            def forward(self, x):
                ## Define forward behavior
                x = self.pool(F.relu(self.conv1(x)))
                x = self.pool(F.relu(self.conv2(x)))
                x = self.pool(F.relu(self.conv3(x)))
                x = self.pool(F.relu(self.conv4(x)))
                # flatten image input
                x = x.view(-1, 128 * 14 * 14)
                # add dropout layer
                x = self.dropout(x)
                # add 1st hidden layer, with relu activation function
                x = F.relu(self.fc1(x))
                # add dropout layer
                x = self.dropout(x)
                # add 2nd hidden layer, with relu activation function
                x = F.relu(self.fc2(x))
                # add dropout layer
                x = self.dropout(x)
                x = self.fc3(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, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=25088, out_features=1000, bias=True)
  (fc2): Linear(in_features=1000, out_features=500, bias=True)
  (fc3): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.2)
)
```

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

Answer: - My final CNN architecture consists of 4 Convolutional layers and MaxPooling layers followed by 3 Fully connected layers. I have used 16 3x3 filters with padding of 1 for the first convolutional layer and gradually increasing the number of filters by a factor of 2. For Max Pooling layers I reduce the length and breadth by a factor of 2 where as increasing the depth. Finally we have 3 fully connected layers with 1000, 500 and 133 (number of classes) units. I have used RELU activation function for the layers and have used Dropout regulzarization with p=0.2 between the FC layers. - I initially used 3 Convoluational layers and 2 fully connected layers but I faced issues to meeting the 10% test accuracy rubric as specified by the project. So I decided to add one more convolutional layer and 1 more fully connected layer.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [12]:
```
import torch.optim as optim

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

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

In [13]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = 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 [14]:
```
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()
    ## 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
    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()
    # update training loss
    train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

######################
# validate the model #
######################
model.eval()
with torch.no_grad():
    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)
         # calculate the batch loss
        loss = criterion(output, target)
        # update average validation loss
        valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


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

    ## TODO: save the model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
```

```
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss
        # return trained model
        return model


# train the model
model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.850713         Validation Loss: 4.725215
Validation loss decreased (inf --> 4.725215).  Saving model ...
Epoch: 2         Training Loss: 4.632405         Validation Loss: 4.556520
Validation loss decreased (4.725215 --> 4.556520).  Saving model ...
Epoch: 3         Training Loss: 4.527247         Validation Loss: 4.472876
Validation loss decreased (4.556520 --> 4.472876).  Saving model ...
Epoch: 4         Training Loss: 4.370166         Validation Loss: 4.247534
Validation loss decreased (4.472876 --> 4.247534).  Saving model ...
Epoch: 5         Training Loss: 4.226288         Validation Loss: 4.160502
Validation loss decreased (4.247534 --> 4.160502).  Saving model ...
Epoch: 6         Training Loss: 4.122058         Validation Loss: 4.118450
Validation loss decreased (4.160502 --> 4.118450).  Saving model ...
Epoch: 7         Training Loss: 4.028961         Validation Loss: 4.059320
Validation loss decreased (4.118450 --> 4.059320).  Saving model ...
Epoch: 8         Training Loss: 3.911133         Validation Loss: 3.920666
Validation loss decreased (4.059320 --> 3.920666).  Saving model ...
Epoch: 9         Training Loss: 3.802415         Validation Loss: 3.943581
Epoch: 10         Training Loss: 3.706408         Validation Loss: 3.876677
Validation loss decreased (3.920666 --> 3.876677).  Saving model ...
Epoch: 11         Training Loss: 3.587921         Validation Loss: 3.834834
Validation loss decreased (3.876677 --> 3.834834).  Saving model ...
Epoch: 12         Training Loss: 3.484577         Validation Loss: 3.705796
Validation loss decreased (3.834834 --> 3.705796).  Saving model ...
Epoch: 13         Training Loss: 3.393525         Validation Loss: 3.729004
Epoch: 14         Training Loss: 3.292607         Validation Loss: 3.661549
Validation loss decreased (3.705796 --> 3.661549).  Saving model ...
Epoch: 15         Training Loss: 3.217931         Validation Loss: 3.598773
Validation loss decreased (3.661549 --> 3.598773).  Saving model ...
Epoch: 16         Training Loss: 3.143845         Validation Loss: 3.577302
Validation loss decreased (3.598773 --> 3.577302).  Saving model ...
Epoch: 17         Training Loss: 3.025565         Validation Loss: 3.698697
Epoch: 18         Training Loss: 2.963040         Validation Loss: 3.679426
Epoch: 19         Training Loss: 2.869311         Validation Loss: 3.659032
Epoch: 20         Training Loss: 2.798219         Validation Loss: 3.718408
Epoch: 21         Training Loss: 2.714042         Validation Loss: 3.684733
```

```
Epoch: 22         Training Loss: 2.656363         Validation Loss: 3.716555
Epoch: 23         Training Loss: 2.574896         Validation Loss: 3.536162
Validation loss decreased (3.577302 --> 3.536162).  Saving model ...
Epoch: 24         Training Loss: 2.506106         Validation Loss: 3.652049
Epoch: 25         Training Loss: 2.436637         Validation Loss: 3.546527
Epoch: 26         Training Loss: 2.349528         Validation Loss: 3.752162
Epoch: 27         Training Loss: 2.287557         Validation Loss: 3.650603
Epoch: 28         Training Loss: 2.244469         Validation Loss: 3.769622
Epoch: 29         Training Loss: 2.183292         Validation Loss: 3.777796
Epoch: 30         Training Loss: 2.119282         Validation Loss: 3.851401
```

1.1.11 (IMPLEMENTATION) Test the Model

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

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

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

    model.eval()
    with torch.no_grad():
        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().nump
            total += data.size(0)

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

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

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

```
Test Loss: 3.581942


Test Accuracy: 19% (167/836)
```

2 Accuracy of the CNN model built from scratch

- The CNN built from scratch achieves a test accuracy of 19% on the test dataset which exceeds the 10% test accuracy rubric specified fot the project

---

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.

2.0.1 (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]: data_dir = '/data/dog_images'
         train_dir = data_dir + '/train'
         valid_dir = data_dir + '/valid'
         test_dir = data_dir + '/test'

In [23]: import os
         import torch
         from torchvision import datasets, transforms

In [24]: ## TODO: Specify data loaders
         train_transforms = transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.RandomRotation(30),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])

         valid_transforms = transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize([0.485, 0.456, 0.406],
                                                                     [0.229, 0.224, 0.225])])
```

```
test_transforms = transforms.Compose([transforms.Resize(256),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

# TODO: Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform = train_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform = valid_transforms)
test_data = datasets.ImageFolder(test_dir, transform = test_transforms)

# TODO: Using the image datasets and the trainforms, define the dataloaders
trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=32)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32)

loaders_transfer = {'train': trainloader, 'valid': validloader, 'test': testloader}
```

2.0.2 (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 [25]:
```
import torchvision.models as models
import torch.nn as nn

use_cuda = torch.cuda.is_available()

## 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

n_inputs = model_transfer.classifier[6].in_features

# add last linear layer (n_inputs -> 5 flower classes)
# new layers automatically have requires_grad = True
last_layer = nn.Linear(n_inputs, 133)

model_transfer.classifier[6] = last_layer
print(model_transfer)

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

```
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)
  )
)
```

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: - For transfer learning exercise I started with a VGG16 baseline model. The VGG16 baseline model has 1000 output units where as in our problem we have 133 classes. So I modified the classifier section of the VGG16 to change the number of nodes in the output layer from 1000 to 133. I also froze the parameters for all layers except the classifier part of the network. - Although the network I chose was rather simplistic, I was able to achieve a test classification accuracy of 62% compared to 60% rubric.

2.0.3 (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 [26]: import torch.optim as optim
```

```
In [27]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.001)
```

2.0.4 (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 [28]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
```

```
In [29]: 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()
                     ## 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
                     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()
            # update training loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        with torch.no_grad():
            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)
                 # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


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

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

In [30]:
```
# train the model
model_transfer = train(20, loaders_transfer, model_transfer, optimizer_transfer, criter

# 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: 4.347772         Validation Loss: 2.299619
Validation loss decreased (inf --> 2.299619).  Saving model ...
Epoch: 2         Training Loss: 3.573876         Validation Loss: 1.684224
Validation loss decreased (2.299619 --> 1.684224).  Saving model ...
Epoch: 3         Training Loss: 3.205577         Validation Loss: 1.531882
```

```
Validation loss decreased (1.684224 --> 1.531882).  Saving model ...
Epoch: 4         Training Loss: 2.985179         Validation Loss: 1.501359
Validation loss decreased (1.531882 --> 1.501359).  Saving model ...
Epoch: 5         Training Loss: 2.941744         Validation Loss: 1.416442
Validation loss decreased (1.501359 --> 1.416442).  Saving model ...
Epoch: 6         Training Loss: 2.914958         Validation Loss: 1.456971
Epoch: 7         Training Loss: 2.794449         Validation Loss: 1.381727
Validation loss decreased (1.416442 --> 1.381727).  Saving model ...
Epoch: 8         Training Loss: 2.731120         Validation Loss: 1.342018
Validation loss decreased (1.381727 --> 1.342018).  Saving model ...
Epoch: 9         Training Loss: 2.691458         Validation Loss: 1.339883
Validation loss decreased (1.342018 --> 1.339883).  Saving model ...
Epoch: 10         Training Loss: 2.655596         Validation Loss: 1.351768
Epoch: 11         Training Loss: 2.738938         Validation Loss: 1.268548
Validation loss decreased (1.339883 --> 1.268548).  Saving model ...
Epoch: 12         Training Loss: 2.590256         Validation Loss: 1.402471
Epoch: 13         Training Loss: 2.657677         Validation Loss: 1.310161
Epoch: 14         Training Loss: 2.602963         Validation Loss: 1.342396
Epoch: 15         Training Loss: 2.629120         Validation Loss: 1.221816
Validation loss decreased (1.268548 --> 1.221816).  Saving model ...
Epoch: 16         Training Loss: 2.583704         Validation Loss: 1.268472
Epoch: 17         Training Loss: 2.531538         Validation Loss: 1.282012
Epoch: 18         Training Loss: 2.561271         Validation Loss: 1.274829
Epoch: 19         Training Loss: 2.605545         Validation Loss: 1.216648
Validation loss decreased (1.221816 --> 1.216648).  Saving model ...
Epoch: 20         Training Loss: 2.532186         Validation Loss: 1.308733
```

2.0.5 (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 [31]:
```python
def test(loaders, model, criterion, use_cuda):

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

    model.eval()
    with torch.no_grad():
        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().nump
            total += data.size(0)

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

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

In [32]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

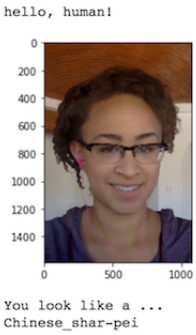
```
Test Loss: 1.275388


Test Accuracy: 62% (522/836)
```

2.0.6 (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 [35]:
```
from PIL import Image
import matplotlib.pyplot as plt
%matplotlib inline
```

In [36]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]

def preprocess_img(img_path):
    img = Image.open(img_path)
    transform_pipeline = transforms.Compose([transforms.Resize((224,224)),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                                  std=[0.229, 0.224, 0.225]
    img = transform_pipeline(img)
    img = img.unsqueeze(0)
    img = img.to('cuda')
    return img
```



Sample Human Output

```
def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = preprocess_img(img_path)
    model_transfer.eval()
    prediction = model_transfer(img)
    prediction = prediction.to('cpu')
    prediction = prediction.data.numpy().argmax()
    return class_names[prediction]
```

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!

2.0.7 (IMPLEMENTATION) Write your Algorithm

In [37]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

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

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if dog_detector(img_path):
        print("\nHello, dog! Your predicted breed is: " + predict_breed_transfer(img_pa
```

```
        print("\n")
        show_image(img_path)
    elif face_detector(img_path):
        print("\nHello, human! You look like a " + predict_breed_transfer(img_path))
        print("\n")
        show_image(img_path)
    else:
        print("\nyou are neither human nor dog!\n")
        show_image(img_path)
```

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?

2.0.8 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

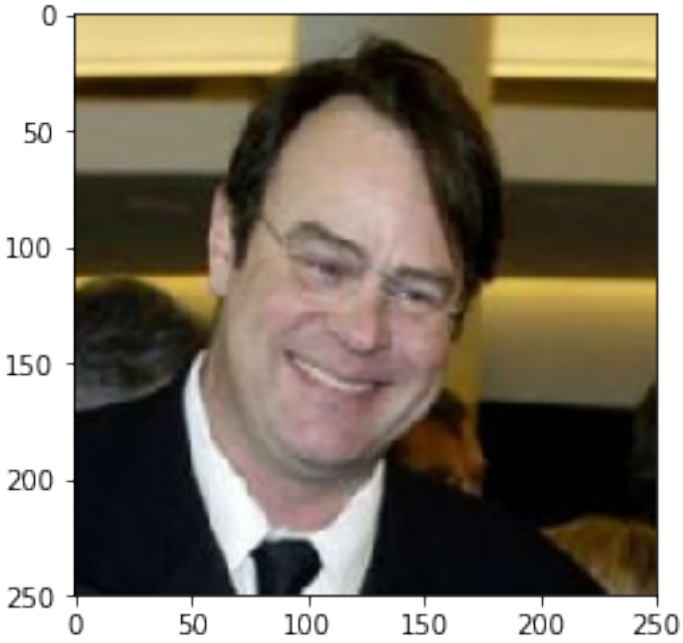
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

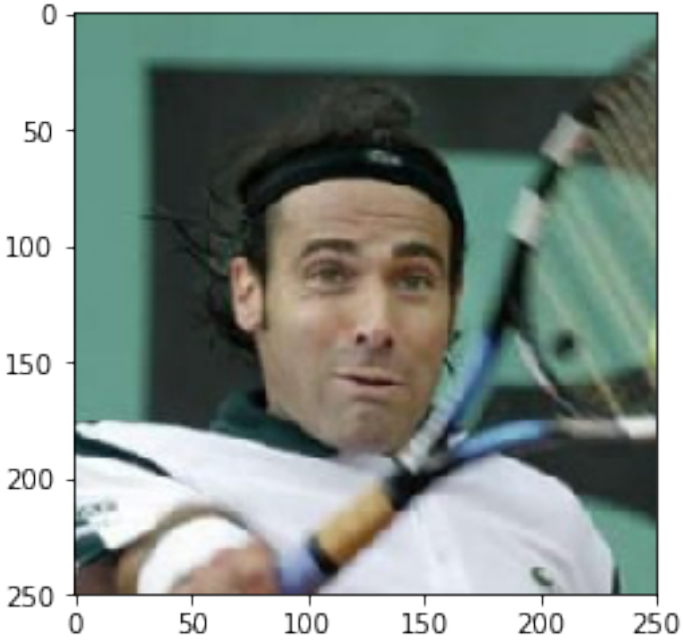
Answer: My algorithm performs pretty well n the limited set of 6 images. It's able to correctly identify humans and also makes a correct prediction for the dog breed. However since the overall accuracy of the model is ~62% for the dog breed classifier, I can think of some possible improvements to increase accuracy of the model: 1. Try adding a couple of FC layers to the classifier section of the model. Might improve accuracy 2. Try using a deeper network architecture like ResNet instead of VGG16 2. Play around with different learning rate and optimization criteria like SGD, Adam, RMSprop etc to improve accuracy

In [48]:
```
## 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)
```

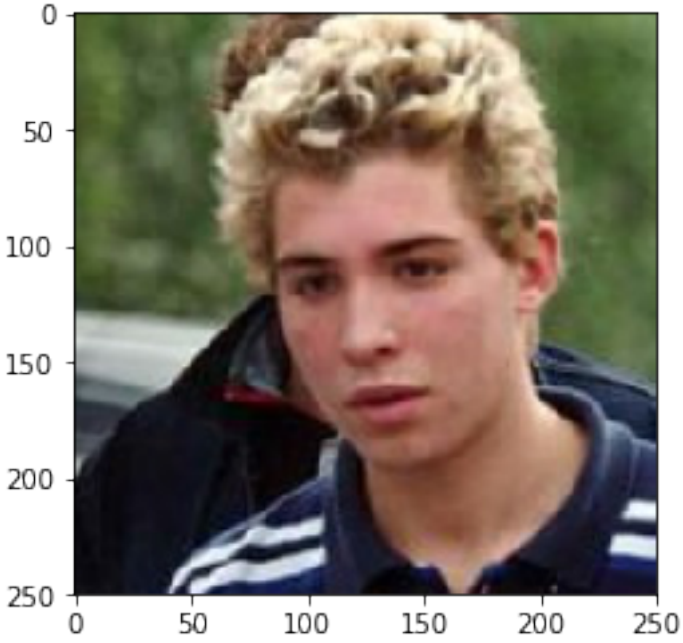
```
Hello, human! You look like a American staffordshire terrier
```




```
Hello, human! You look like a Greyhound
```

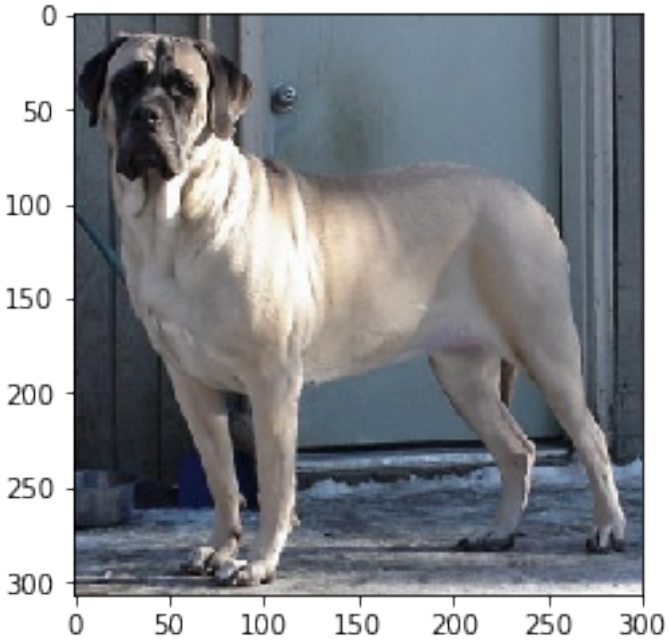


```
Hello, human! You look like a Cane corso
```

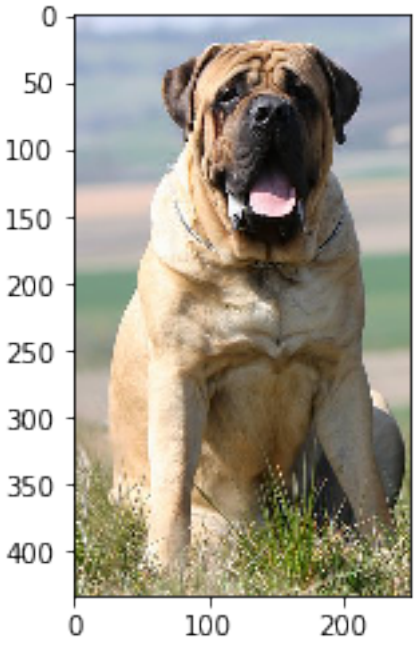



```
Hello, dog! Your predicted breed is: Bullmastiff
```


```
Hello, dog! Your predicted breed is: Bullmastiff
```



```
Hello, dog! Your predicted breed is: Bullmastiff
```



In [ ]: