dog_app

July 13, 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 [12]: 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

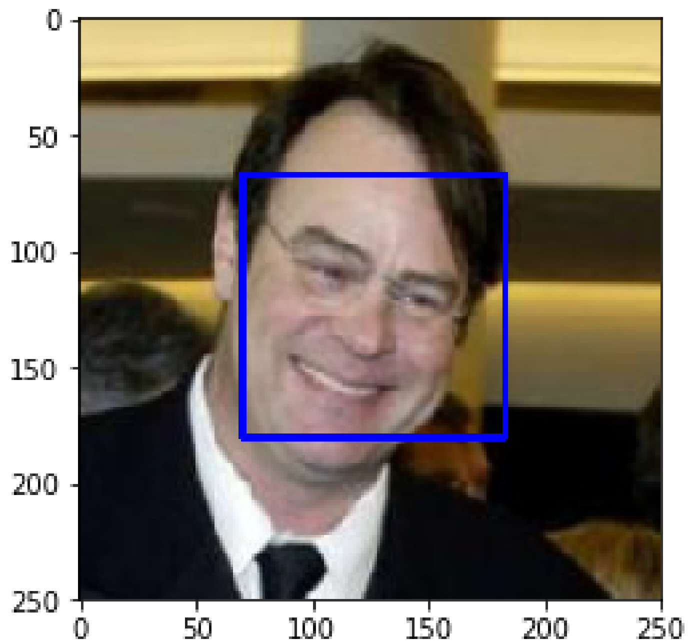
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [13]: 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 [14]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

Percentage of human faces detected by using the human_files: 98%

Percentage of human faces detected by using the dog_files: 17%

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

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

        #-#-# Do NOT modify the code above this line. #-#-#
        human_face_count = 0
        dog_face_count = 0
        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        for human_faces in tqdm(human_files_short):
            if face_detector(human_faces):
                human_face_count += 1

        for dog_faces in tqdm(dog_files_short):
            if face_detector(dog_faces):
                dog_face_count += 1

        print(f'Performance of face detector in human images: {float(human_face_count/100)*100}%
```

```
100%|| 100/100 [00:06<00:00, 14.85it/s]
100%|| 100/100 [01:13<00:00,  4.30it/s]
```

```
Performance of face detector in human images: 98.0%, and the one dog face: 17.0%
```

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 [15]: 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, 93204778.20it/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 [16]: 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
             '''
             VGG16.eval()
             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             transformation = transforms.Compose([
                 transforms.RandomResizedCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])
             ])


             image = Image.open(img_path)
             #image = Variable(image)


             image = transformation(image)
             image = torch.unsqueeze(image,0)

             if use_cuda:
                 image = image.cuda()

             ## Return the *index* of the predicted class for that image
             with torch.no_grad():
                 prediction = VGG16(image)
                 prediction  = torch.argmax(prediction).item()
```

```
        #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 [17]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             index = VGG16_predict(img_path)
             if 151 < index <= 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:

In the human files, 1% had a detected dog, meanwhile, in the dog_files, 99& had a detected dog

```
In [51]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         from tqdm import tqdm

         dog_in_human_files = 0
         dog_in_dog_files = 0

         for dog_faces in tqdm(human_files_short):
             if dog_detector(dog_faces):
                 dog_in_human_files += 1
         for dog_faces in tqdm(dog_files_short):
             if dog_detector(dog_faces):
                 dog_in_dog_files += 1

         print(f'Performance of dog detector in human images: {float(dog_in_human_files/100)*100
```

```
        # Reference: https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html
```

```
100%|| 100/100 [01:18<00:00,  1.31it/s]
100%|| 100/100 [01:17<00:00,  1.32it/s]

Performance of dog detector in human images: 3.0%, and the performance of the dog detector in th
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

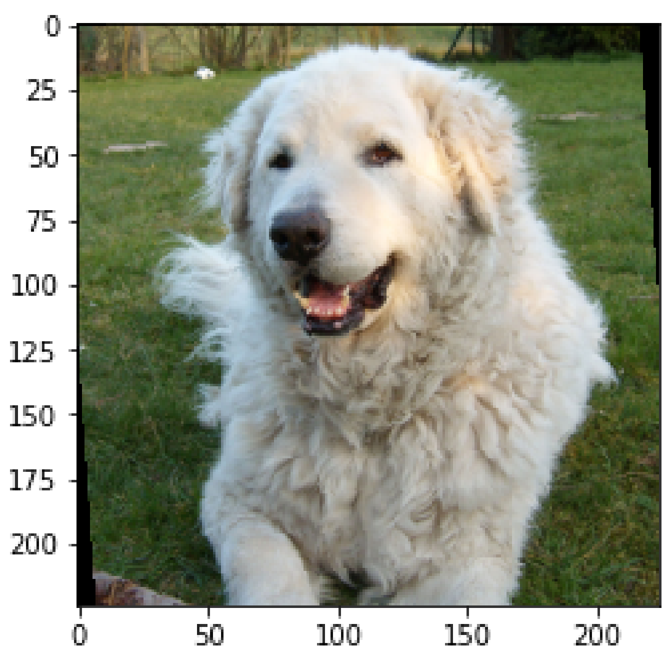
Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [36]: import os
         from torchvision import datasets, transforms
         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True


         # Function to display the image (Reference: video class)
         def imshow(img):
             img = img / 2 + 0.5 # unnormalize
             plt.imshow(np.transpose(img, (1, 2, 0)))

             # Definning the batch size
         batch_size = 64

         # Specifying transform for training and validation and test datasets
         train_transform = transforms.Compose([
             transforms.Resize(224),
             transforms.RandomHorizontalFlip(),
             #transforms.RandomGrayscale(p=0.2),
             transforms.RandomRotation(15),
             transforms.RandomVerticalFlip(p=0.2),
             transforms.CenterCrop(224),
             transforms.ToTensor(),
             transforms.Normalize([0.5,0.5,0.5], [0.5,0.5,0.5])
         ])
         test_transform = transforms.Compose([
             transforms.Resize(224),
             transforms.CenterCrop(224),
```

```
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])

# Loading and transforming the images from the original path folders

train_data = datasets.ImageFolder('/data/dog_images/train', transform = train_transform
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform = test_transform)
test_data = datasets.ImageFolder('/data/dog_images/test', transform = test_transform)



# Selecting the batches for training, validation and testing
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffle=T
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, shuffle=Fal


#Displaying a random image
data_iter = iter(train_loader)
images,labels = data_iter.next()
images = images.numpy()
imshow(images[2])
```



In [37]:
```
# Amount of labels of hour data
classes= train_data.class_to_idx
```

```
len(classes)
```

```
Out[37]: 133
```

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

The code resizes the images by croping them. Many images have different size, so I resize them so they all have the same size. For the input size I picked 224. By testing and the study of other models such as VGG16 I noticed this is a good value. If I pick a smaller value, the images start to get pixeled and lose quality. If I a picked a higher value, I would get a higher quality picture, but I am guessing these will require a higher computer power and more time of training.

I decided to do augmentation by randomly rotating and flipping the data. I did so the model could get more details of the image from different angles, expecting in this way getting a higher accuracy in the results. As explained in some parts of the course, this is not necessarily true and will depend on the images, so it is up to try and see how well it does.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [38]: 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__()
                 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
                 # convolution layer (224x224x3 image tensor)
                 self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
                 self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
                 self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
                 self.conv4 = nn.Conv2d(256,512, kernel_size=3, padding=1)
```

```
        self.conv5 = nn.Conv2d(512,512, kernel_size=3, padding=1)


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


        # linear layer = (512*7*7)
        self.fc1 = nn.Linear(512*7*7, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 133)

        # dropout layer
        self.dropout = nn.Dropout(0.4)




    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)))
        x = self.pool(F.relu(self.conv5(x)))



        # flatten image input
        x = x.view(-1, 512*7*7)

        # add dropout

        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        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()

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

```
model_scratch.cuda()
```

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

Answer:

These steps took a lot of try and error. At the end, I picked five 5 layers of convolution, with kernel = 3 and padding = 1, because it was a proven effective parameters along the course. The last layer (self.conv5 = nn.Conv2d(512,512, kernel_size=3, padding=1) proved effective to decrease the size of the matrix with the maxpooling during feedforward. I used relu as activation function, and dropout of 0.4 to avoid possible overfitting.

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 [39]: import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [40]: # Building loaders_scratch
         loaders_scratch ={
             'train': train_loader,
             'valid': valid_loader,
             'test': test_loader
         }
```

```
In [42]: 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
    #print(batch_idx)

    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()
    output = model(data)
    loss = criterion(output, target)

    # backward pass
    loss.backward()

    # optimization step
    optimizer.step()

    #train_loss += loss.item()*data.size(0)
    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
    with torch.no_grad():
        output = model(data)

    loss = criterion(output, target)

    # updating average validation loss
    #valid_loss += loss.item()*data.size(0)
    valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


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

```
            ))

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

    # return trained model
    return model


# # train the model
model_scratch = train(35, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
In [44]: # # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [45]: 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: 3.461277


Test Accuracy: 18% (157/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [1]: import torch.nn as nn
        import torch.nn.functional as F
        import os
        import torch
        from torchvision import transforms, datasets
        import torch.optim as optim
        import matplotlib.pyplot as plt
```

```
In [2]: ## TODO: Specify data loaders
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        def imshow(img):
            img = img / 2 + 0.456 # unnormalize
            plt.imshow(np.transpose(img, (1, 2, 0)))




        data_dir = '/data/dog_images/'
        train_dir = os.path.join(data_dir, 'train')
```

```
valid_dir = os.path.join(data_dir, 'valid')
test_dir = os.path.join(data_dir, 'test')

train_transform = transforms.Compose([
    transforms.Resize(224),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
])

batch_size = 64

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

#building loader for transfer learning
train_loader_tr = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                              num_workers=0, shuffle=True)
valid_loader_tr = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                             num_workers=0, shuffle=True)
test_loader_tr = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                            num_workers=0, shuffle=False)
```

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 [5]: import torchvision.models as models
        import torch.nn as nn
        use_cuda = torch.cuda.is_available()
        ## TODO: Specify model architecture

        model_transfer = models.vgg19(pretrained=True)
        print(model_transfer)
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:08<00:00, 63981126.61it/s]


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): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (17): ReLU(inplace)
    (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (19): Conv2d(256, 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): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (24): ReLU(inplace)
    (25): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (26): ReLU(inplace)
    (27): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (31): ReLU(inplace)
    (32): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (33): ReLU(inplace)
    (34): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (35): ReLU(inplace)
    (36): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

In [50]:
```
# Freezing the feature layers
for param in model_transfer.features.parameters():
    param.requires_grad = False
```

```
In [6]: n_inputs = model_transfer.classifier[6].in_features
        last_layer = nn.Linear(n_inputs, 133)

        model_transfer.classifier[6] = last_layer

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

```
In [7]: print(model_transfer.classifier[6].in_features)
        print(model_transfer.classifier[6].out_features)
```

```
4096
133
```

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

Answer:

After doing some research, I found out that VGG-19, which was trained in ImageNet obtaining a 92.7% accuracy, worked well for doog classification. . SImilar to VGG 16, this model have three more layers. Also, I could see that this model use a similar structure that the one I built before with similar kernel size of 3.

In my architecture, I downloaded the model and I printed the input and output festures for the last layer, which will be the one I will modify to train mine. Then I enter my own last 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 [52]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.0001)
```

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 [53]: loaders_transfer ={
             'train': train_loader_tr,
             'valid': valid_loader_tr,
             'test': test_loader_tr
         }
```

```
In [43]: # train the model
         model_transfer =  train(10, loaders_transfer, model_transfer, optimizer_transfer, crite

         # 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: 1.453642        Validation Loss: 0.681891
Validation loss decreased (inf --> 0.681891). Saving model ...
Epoch: 2         Training Loss: 0.535901        Validation Loss: 0.660836
Validation loss decreased (0.681891 --> 0.660836). Saving model ...
Epoch: 3         Training Loss: 0.317956        Validation Loss: 0.640526
Validation loss decreased (0.660836 --> 0.640526). Saving model ...
Epoch: 4         Training Loss: 0.220511        Validation Loss: 0.584681
Validation loss decreased (0.640526 --> 0.584681). Saving model ...
Epoch: 5         Training Loss: 0.167235        Validation Loss: 0.663722
Epoch: 6         Training Loss: 0.119917        Validation Loss: 0.656757
Epoch: 7         Training Loss: 0.119607        Validation Loss: 0.678566
Epoch: 8         Training Loss: 0.093414        Validation Loss: 0.774894
Epoch: 9         Training Loss: 0.081393        Validation Loss: 0.804013
Epoch: 10         Training Loss: 0.139961        Validation Loss: 0.849093
```

```
In [8]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

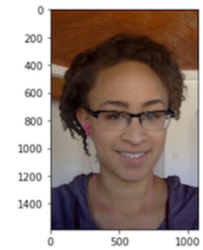
1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [44]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.714825


Test Accuracy: 81% (683/836)
```

```
In [26]: from PIL import Image
         class_names = [item[4:].replace("_", " ") for item in train_data.classes]

         def predict_breed_transfer(img_path):

             transformation = transforms.Compose([
                 transforms.Resize(224),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                      std=[0.229, 0.224, 0.225])
             ])

             image = Image.open(img_path)
             image = transformation(image)
             image = torch.unsqueeze(image, 0)

             if use_cuda:
```



Sample Human Output

```
        image = image.cuda()

    model_transfer.eval()
    with torch.no_grad():
        prediction = model_transfer(image)
    prediction = torch.argmax(prediction)
    #prediction = prediction.data.numpy().argmax()
    model_transfer.train()
    breed = class_names[prediction]

    return breed
```

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.

Step 5: Write your Algorithm

Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [10]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
```

```
## handle cases for a human face, dog, and neither
if face_detector(img_path):
    print('Hello human!')
    image = Image.open(img_path)
    plt.imshow(image)
    plt.show()
    print(f"You look like a.... \n {predict_breed_transfer(img_path)}!")
    print('\n\n\n')
elif dog_detector(img_path):
    print('What a nice dog! \n')
    image = Image.open(img_path)
    plt.imshow(image)
    plt.show()
    print(f"It must be a {predict_breed_transfer(img_path)}!")
    print('\n\n\n')
else:
    print("Sorry, this is neither a dog of a person")
    image = Image.open(img_path)
    plt.imshow(image)
    plt.show()
    print('\n\n\n')
```

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 was around what I could expect. Taking into account the hardship of identifying a dog breed, the model had an accuracy over 80%.

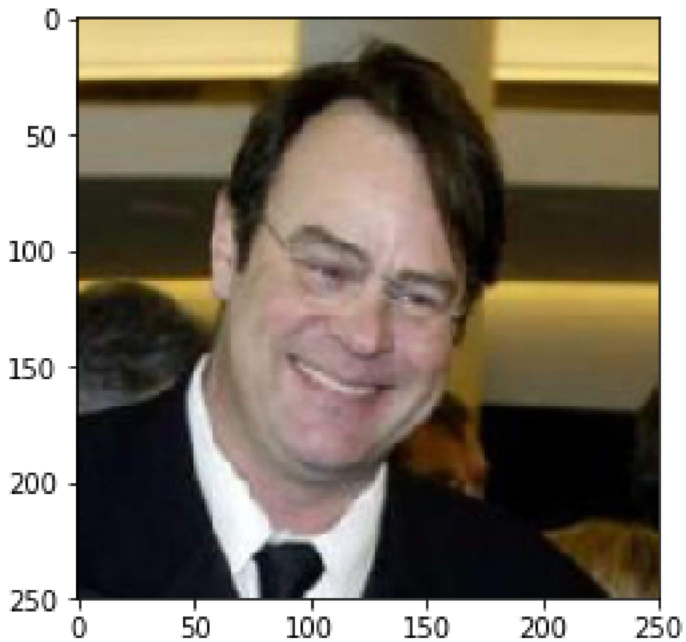
The first thing I would do for improvement is to use some augmentation in the dog and human face in order to improve the accuracy.

The second thing would be to increase the number of epoch when training in order to improve the accuracy.

THe third if that I would use transfer learning when identifying the dog face.

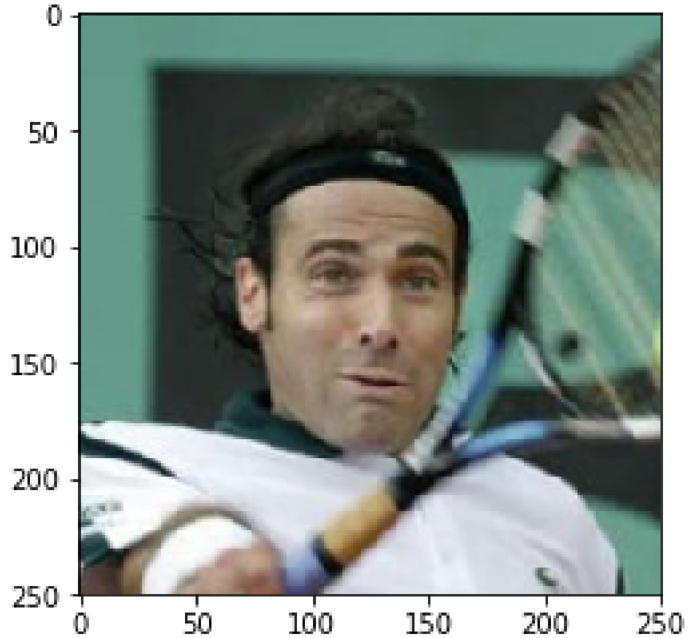
In [48]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

```
Hello human!
```



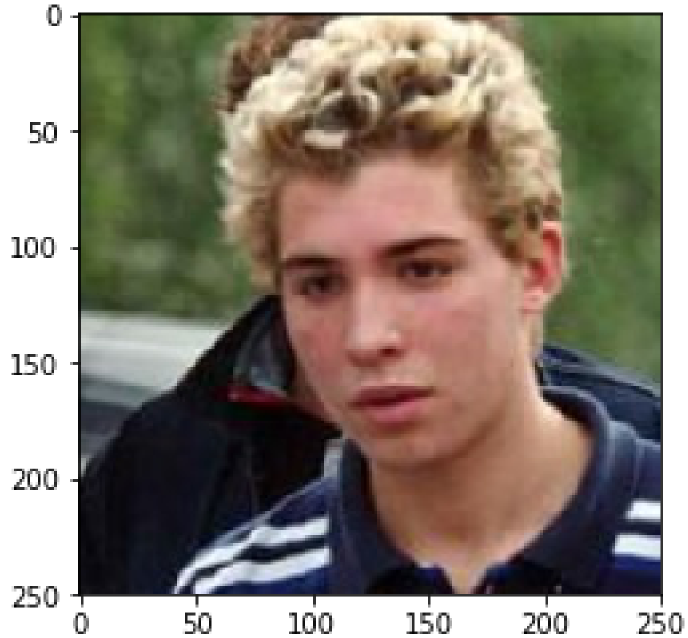
```
You look like a...
 Great pyrenees!
```

```
Hello human!
```



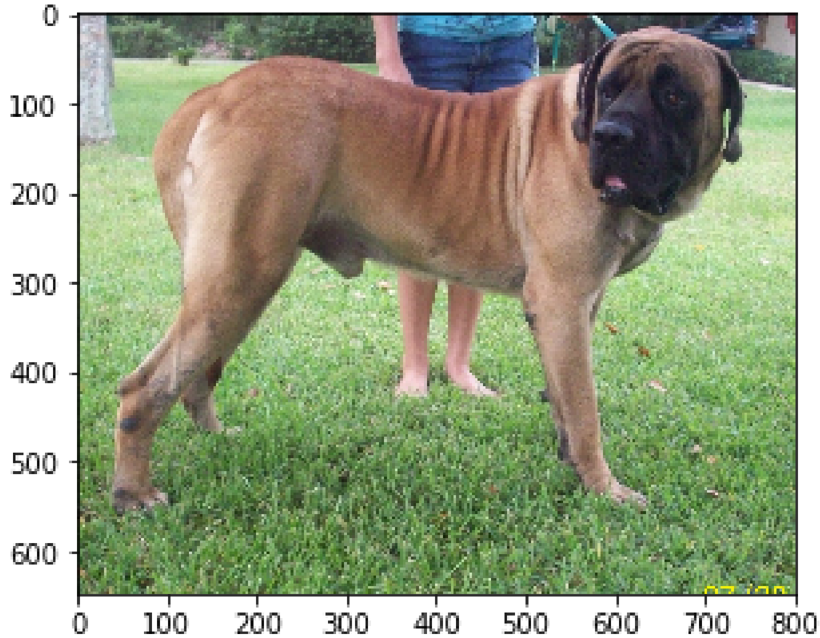

```
You look like a...
 Parson russell terrier!
```

```
Hello human!
```



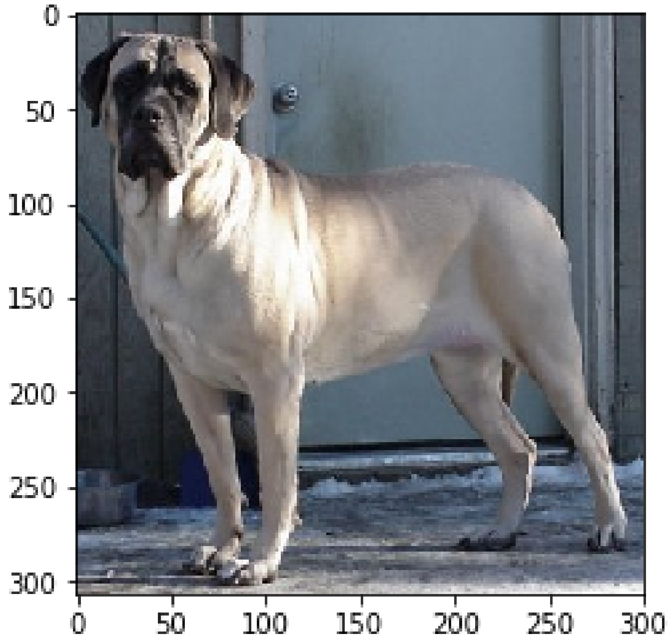

```
You look like a...
 Black russian terrier!
```

```
What a nice dog!
```



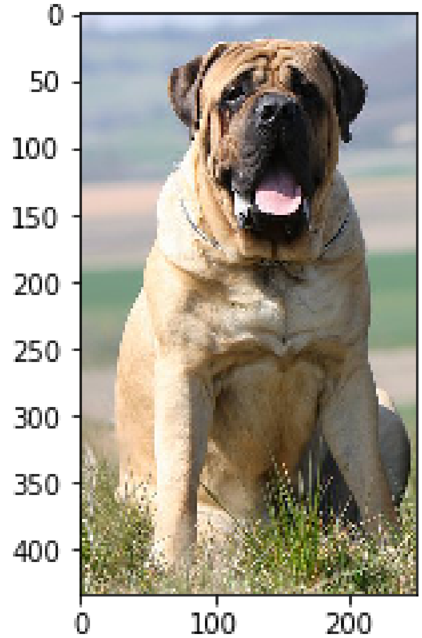

```
It must be a Norwich terrier!
```

```
What a nice dog!
```




```
It must be a Border collie!
```

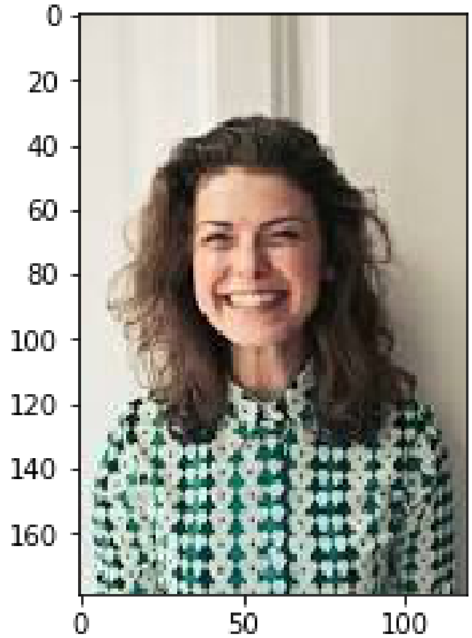
```
What a nice dog!
```




```
It must be a Cavalier king charles spaniel!
```

```
In [29]: ## TODO: Specify data loaders
         no_dog_files = ['./my_images/image1.jpg', './my_images/image2.jpg', './my_images/image3
         my_dog_files = ['./my_images/image_dog1.jpg', './my_images/image_dog2.jpg', './my_image

         # dir = './my_images'
         # for file in (dir):
         #     run_app(file)
         for file in np.hstack((no_dog_files, my_dog_files)):
             run_app(file)
```

```
Hello human!
```

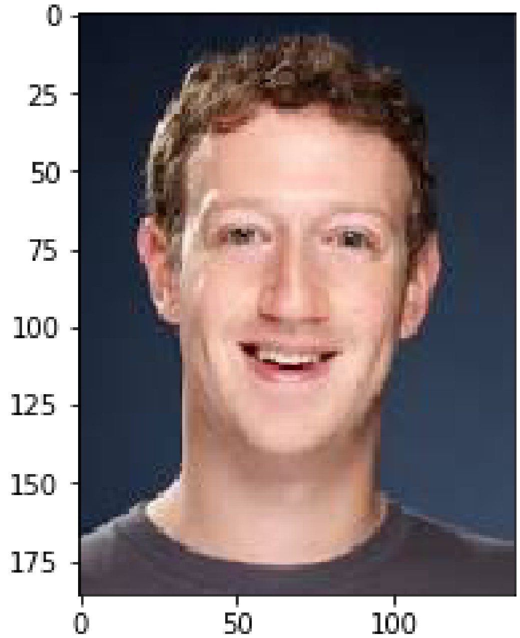



```
You look like a...
 American water spaniel!
```

```
Hello human!
```

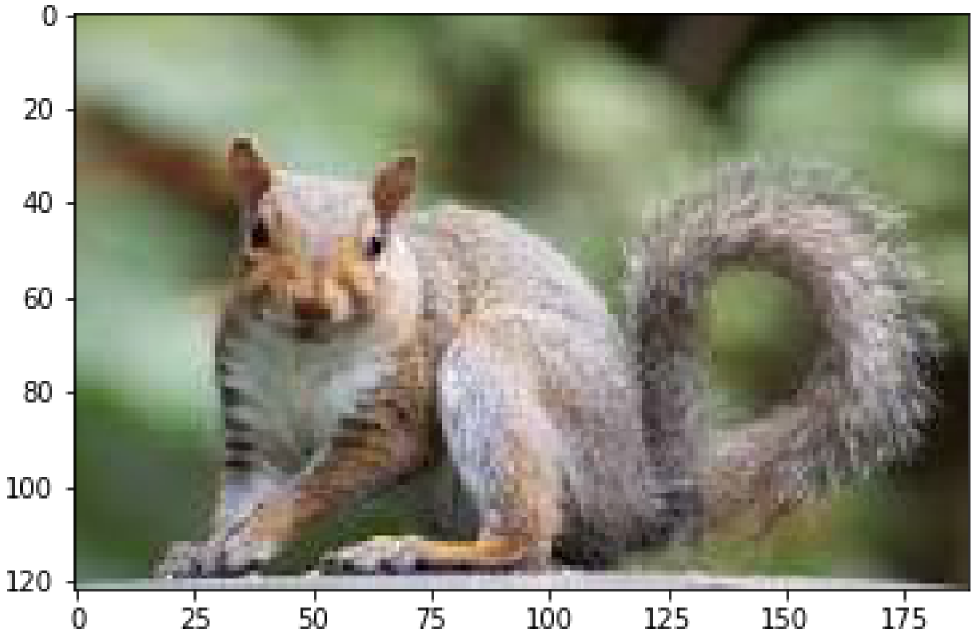

```
You look like a...
 Doberman pinscher!
```

```
Hello human!
```




```
You look like a...
 Chinese shar-pei!
```

```
Sorry, this is neither a dog of a person
```



What a nice dog!

```
It must be a Pomeranian!
```

```
What a nice dog!
```


```
It must be a Manchester terrier!
```

```
What a nice dog!
```


```
It must be a Beauceron!
```

In [ ]: