dog_app

September 20, 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 [4]: 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

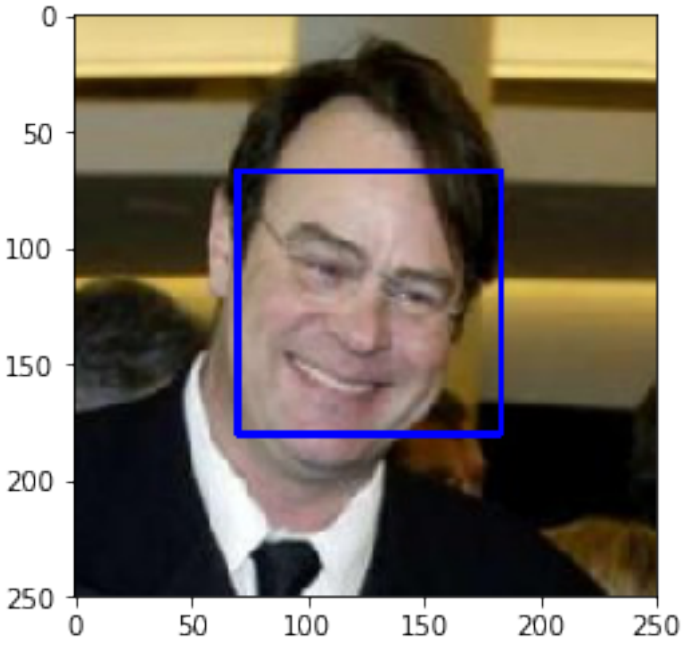
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [5]: 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 [6]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [7]: 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.
        detected_count_human = 0
        detected_count_dog = 0
        for file in human_files_short:
            if face_detector(file):
                detected_count_human += 1

        for file in dog_files_short:
            if face_detector(file):
                detected_count_dog += 1

        print("{}% faces detected in human_files".format(detected_count_human))
        print("{}% faces detected in dog_files".format(detected_count_dog))
```

```
98% faces detected in human_files
17% faces detected in dog_files
```

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 [8]:
```
### (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 [9]:
```
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 [00:13<00:00, 40569298.47it/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 [10]: device = 'cuda' if use_cuda else 'cpu'

In [11]: from PIL import Image

         # From https://knowledge.udacity.com/questions/32899
         from PIL import ImageFile
         import torchvision.transforms as transforms

         ImageFile.LOAD_TRUNCATED_IMAGES = True

         def im_convert(tensor):
             image = tensor.to("cpu").clone().detach()
             image = image.numpy().squeeze()
             image = image.transpose(1,2,0)
             image = image * np.array((0.229, 0.224, 0.225)) + np.array((0.485, 0.456, 0.406))
             image = image.clip(0, 1)

             return image

In [16]: 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

             image = Image.open(img_path).convert('RGB')
             in_transform = transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))])
             image = in_transform(image)[:3,:,:].unsqueeze(0)
             image = image.to(device)
             output = VGG16(image)
             accuracy, class_index = output.topk(1)

             return class_index.item() # predicted class index
         # VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jpg')
```

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)
             return 151 <= index <= 268 # true/false

         dog_detector('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jpg')
```

```
Out[17]: True
```

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 [18]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         detected_count_human = 0
         detected_count_dog = 0
         # human_files_short, dog_files_short = human_files_short.to(device), dog_files_short.to
         for file in human_files_short:
             if dog_detector(file):
                 detected_count_human += 1

         for file in dog_files_short:
             if dog_detector(file):
                 detected_count_dog += 1

         print("{}% dog detected in human_files".format(detected_count_human))
         print("{}% dog detected in dog_files".format(detected_count_dog))
```

```
1% dog detected in human_files
100% dog detected in dog_files
```

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 [19]: ### (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 [20]:
```
def img_show(img):
    img = img.transpose((1, 2, 0))
    img = img*np.array([0.229, 0.224, 0.225]) + np.array([0.485, 0.456, 0.406])
    img = img.clip(0,1)
    plt.imshow(img)
    plt.show()
```

In [21]:
```
import os
from torchvision import datasets
from torch.utils.data.sampler import SubsetRandomSampler

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_transform = transforms.Compose([transforms.Resize((256, 256)),
                                     transforms.CenterCrop(224),
                                     transforms.RandomRotation(10),
                                     transforms.RandomHorizontalFlip(),
                                     transforms.ToTensor(),
                                     transforms.Normalize((0.485, 0.456, 0.406),
                                             (0.229, 0.224, 0.225))])

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

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

batch_size = 20
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))
np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, sampler=t
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, sampler=v
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)

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

```
# images, label = next(iter(train_loader))
# img = images.numpy()[0]
# img_show(img)
# print(label[0].item())
```

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

Answer:

I used transforms.Resize for resizing images. Following VGG, I set image size as 224. I decided to use RandomRotation, RandomHorizontalFlip and RandomVerticalFlip to reduce the risk of overfitting.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [29]: 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, stride=1)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1, stride=1)
                 self.pool = nn.MaxPool2d(2, 2)

                 # (WF+2P)/S+1
                 self.fc1 = nn.Linear(56*56*32, 160)# Number of breed
                 self.fc2 = nn.Linear(160, 133)
                 self.dropout = nn.Dropout(0.3)
                 self.batch_norm = nn.BatchNorm1d(160)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.pool(F.relu(self.conv1(x)))
                 x = self.pool(F.relu(self.conv2(x)))
                 # (WF+2P)/S+1
                 x = self.dropout(F.relu(self.batch_norm(self.fc1(x.view(-1, 56*56*32)))))
                 x = self.fc2(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()

model_scratch
```

```
Out[29]: 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))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=100352, out_features=160, bias=True)
  (fc2): Linear(in_features=160, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
  (batch_norm): BatchNorm1d(160, eps=1e-05, momentum=0.1, affine=True, track_running_st
)
```

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

Answer:

First, I created 3 convolutional layer and 3 FC layer.
Although I tried to change some parameters, it didn't decrease loss at all.
Then I found this post and got a clue. https://knowledge.udacity.com/questions/106988

I reduced number of layers of my network. Tried with some learning rate and I decided to set it 0.001. Reduced kernel layer and Increased dropout rate to avoid overfitting. But I couldn't reduce validation loss enough. Then, I added batch normarization. Finally I works.

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 [30]: 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)
```

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 [31]: import sys
```

```
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
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            if batch_idx%50 == 0:
                print("")
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            sys.stdout.write("\rBatch: {} loss: {}".format(batch_idx, train_loss))
            sys.stdout.flush()
            loss.backward()
            optimizer.step()

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


        # print training/validation statistics
        print('Epoch: {} \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:
                    torch.save(model.state_dict(), save_path)
                    print("valid_loss decreased {} -> {}".format(valid_loss_min, valid_loss))
                    print("saving the model...")
                    valid_loss_min = valid_loss

            # return trained model
            return model
```

In [32]:
```
# train the model
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Batch: 49 loss: 4.8893408775329595
Batch: 99 loss: 4.8348898887634285
Batch: 149 loss: 4.7902979850769045
Batch: 199 loss: 4.7442464828491215
Batch: 249 loss: 4.6974830627441415
Batch: 267 loss: 4.6799492835998535Epoch: 1         Training Loss: 4.679949         Validation L
valid_loss decreased inf -> 4.4672393798828125
saving the model...

Batch: 49 loss: 4.4415831565856935
Batch: 99 loss: 4.3755211830139165
Batch: 149 loss: 4.3555259704589845
Batch: 199 loss: 4.3452882766723635
Batch: 249 loss: 4.3285336494445835
Batch: 267 loss: 4.325262069702148Epoch: 2         Training Loss: 4.325262         Validation Lo
valid_loss decreased 4.4672393798828125 -> 4.280754566192627
saving the model...

Batch: 49 loss: 4.1558461189270025
Batch: 99 loss: 4.1343960762023935
Batch: 149 loss: 4.1368718147277835
Batch: 199 loss: 4.1230082511901855
Batch: 249 loss: 4.1129803657531745
Batch: 267 loss: 4.1169409751892095Epoch: 3         Training Loss: 4.116941         Validation L
valid_loss decreased 4.280754566192627 -> 4.138545036315918
saving the model...

Batch: 49 loss: 3.9425165653228767
```

```
Batch: 99 loss: 3.9385404586791996
Batch: 149 loss: 3.9494285583496094
Batch: 199 loss: 3.9451212882995605
Batch: 249 loss: 3.9449181556701666
Batch: 267 loss: 3.9443848133087163Epoch: 4         Training Loss: 3.944385         Validation L
valid_loss decreased 4.138545036315918 -> 4.045343399047852
saving the model...

Batch: 49 loss: 3.8152234554290777
Batch: 99 loss: 3.8239822387695312
Batch: 149 loss: 3.8093538284301766
Batch: 199 loss: 3.7915136814117435
Batch: 249 loss: 3.7925832271575928
Batch: 267 loss: 3.7926018238067627Epoch: 5         Training Loss: 3.792602         Validation L
valid_loss decreased 4.045343399047852 -> 4.024439811706543
saving the model...

Batch: 49 loss: 3.6204066276550293
Batch: 99 loss: 3.6291017532348633
Batch: 149 loss: 3.6264777183532715
Batch: 199 loss: 3.6210064888000496
Batch: 249 loss: 3.6227190494537354
Batch: 267 loss: 3.6135492324829177Epoch: 6         Training Loss: 3.613549         Validation L
valid_loss decreased 4.024439811706543 -> 4.011301517486572
saving the model...

Batch: 49 loss: 3.4475920200347955
Batch: 99 loss: 3.4497647285461426
Batch: 149 loss: 3.4649484157562256
Batch: 199 loss: 3.4857945442199707
Batch: 249 loss: 3.4705731868743896
Batch: 267 loss: 3.4693818092346196Epoch: 7         Training Loss: 3.469382         Validation L
valid_loss decreased 4.011301517486572 -> 3.932079553604126
saving the model...

Batch: 49 loss: 3.2070145606994636
Batch: 99 loss: 3.2588496208190925
Batch: 149 loss: 3.2709867954254156
Batch: 199 loss: 3.2548325061798096
Batch: 249 loss: 3.2532031536102295
Batch: 267 loss: 3.2679445743560793Epoch: 8         Training Loss: 3.267945         Validation L
valid_loss decreased 3.932079553604126 -> 3.9217069149017334
saving the model...

Batch: 49 loss: 3.0782248973846436
Batch: 99 loss: 3.0708193778991714
Batch: 149 loss: 3.0737519264221194
Batch: 199 loss: 3.0822215080261237
```

```
Batch: 249 loss: 3.1051440238952637
Batch: 267 loss: 3.1177721023559575Epoch: 9         Training Loss: 3.117772         Validation L
valid_loss decreased 3.9217069149017334 -> 3.8542261123657227
saving the model...

Batch: 49 loss: 2.8352687358856227
Batch: 99 loss: 2.8748586177825928
Batch: 149 loss: 2.8599298000335693
Batch: 199 loss: 2.8790783882141113
Batch: 249 loss: 2.8723974227905273
Batch: 267 loss: 2.8760387897491455Epoch: 10         Training Loss: 2.876039         Validation

Batch: 49 loss: 2.5584728717803955
Batch: 99 loss: 2.5802075862884523
Batch: 149 loss: 2.6166148185729987
Batch: 199 loss: 2.6298406124114997
Batch: 249 loss: 2.6457254886627197
Batch: 267 loss: 2.6590147018432617Epoch: 11         Training Loss: 2.659015         Validation

Batch: 49 loss: 2.3924343585968018
Batch: 99 loss: 2.4164760112762457
Batch: 149 loss: 2.4451932907104493
Batch: 199 loss: 2.4763989448547363
Batch: 249 loss: 2.4831180572509766
Batch: 267 loss: 2.4918975830078125Epoch: 12         Training Loss: 2.491898         Validation

Batch: 49 loss: 2.2392685413360596
Batch: 99 loss: 2.2056486606597946
Batch: 149 loss: 2.2418098449707037
Batch: 199 loss: 2.2616817951202393
Batch: 249 loss: 2.2863645553588867
Batch: 267 loss: 2.2929003238677986Epoch: 13         Training Loss: 2.292900         Validation

Batch: 49 loss: 2.0629060268402145
Batch: 99 loss: 2.0260505676269538
Batch: 149 loss: 2.0340623855590825
Batch: 199 loss: 2.0470225811004643
Batch: 249 loss: 2.0791366100311288
Batch: 267 loss: 2.0880508422851562Epoch: 14         Training Loss: 2.088051         Validation

Batch: 49 loss: 1.8144284486770638
Batch: 99 loss: 1.8680399656295776
Batch: 149 loss: 1.8867409229278564
Batch: 199 loss: 1.8761118650436401
Batch: 249 loss: 1.8898450136184692
Batch: 267 loss: 1.9133514165878296Epoch: 15         Training Loss: 1.913351         Validation

Batch: 49 loss: 1.7540313005447388
```

```
Batch: 99 loss: 1.7440081834793097
Batch: 149 loss: 1.7493730783462524
Batch: 199 loss: 1.7554696798324585
Batch: 249 loss: 1.7605990171432495
Batch: 267 loss: 1.7585849761962897Epoch: 16         Training Loss: 1.758585         Validation

Batch: 49 loss: 1.4920227527618408
Batch: 99 loss: 1.5584733486175537
Batch: 149 loss: 1.5553493499755864
Batch: 199 loss: 1.5513290166854858
Batch: 249 loss: 1.5708340406417847
Batch: 267 loss: 1.5772060155868536Epoch: 17         Training Loss: 1.577206         Validation

Batch: 49 loss: 1.5073082447052002
Batch: 99 loss: 1.4974793195724487
Batch: 149 loss: 1.4906544685363776
Batch: 199 loss: 1.4999467134475708
Batch: 249 loss: 1.4973150491714478
Batch: 267 loss: 1.5064233541488647


        ---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-32-7dafa179598a> in <module>()
          1 # train the model
          2 model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
    ----> 3                       criterion_scratch, use_cuda, 'model_scratch.pt')


        <ipython-input-31-90233d062825> in train(n_epochs, loaders, model, optimizer, criterion,
         37         ######################
         38         model.eval()
    ---> 39         for batch_idx, (data, target) in enumerate(loaders['valid']):
         40             # move to GPU
         41             if use_cuda:


        /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
        262         if self.num_workers == 0:  # same-process loading
        263             indices = next(self.sample_iter)  # may raise StopIteration
    --> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
        265             if self.pin_memory:
        266                 batch = pin_memory_batch(batch)


        /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
```

```
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
     99         """
    100         path, target = self.samples[index]
--> 101         sample = self.loader(path)
    102         if self.transform is not None:
    103             sample = self.transform(sample)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    145         return accimage_loader(path)
    146     else:
--> 147         return pil_loader(path)
    148
    149


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    128     with open(path, 'rb') as f:
    129         img = Image.open(f)
--> 130         return img.convert('RGB')
    131
    132


    /opt/conda/lib/python3.6/site-packages/PIL/Image.py in convert(self, mode, matrix, dithe
    890         """
    891
--> 892         self.load()
    893
    894         if not mode and self.mode == "P":


    /opt/conda/lib/python3.6/site-packages/PIL/ImageFile.py in load(self)
    233
    234                             b = b + s
--> 235                             n, err_code = decoder.decode(b)
    236                             if n < 0:
    237                                 break


    KeyboardInterrupt:
```

In [33]:
```
# 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 [34]:
```
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: 3.856180


Test Accuracy: 12% (102/836)
```

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 [38]: ## TODO: Specify data loaders
         #loaders_transfer = {'train': train_loader, 'valid': valid_loader, 'test': test_loader}
         loaders_transfer = loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [45]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg11(pretrained=True)
         # print(model_transfer)
         for param in model_transfer.features.parameters():
             param.requires_grad = False
         model_transfer.classifier = nn.Sequential(nn.Linear(25088, 200),
                                                   nn.BatchNorm1d(200),
                                                   nn.ReLU(),
                                                   nn.Dropout(0.5),
                                                   nn.Linear(200, 133))
         # print(model_transfer)
         # model_transfer = models.resnet18()
         # for param in model_transfer.parameters():
         #     param.requires_grad = False
         # model_transfer.fc = nn.Sequential(nn.Linear(512, 133))

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

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

Answer:

I chose VGG11 as a CNN architecture as I thought it is light enough to detect dog's breed. I created a classifier that consists of 2 FC layer with batch normalization and dropout.

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 [46]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.classifier.parameters(), lr=0.005)
         # optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.03)
```

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 [47]: # train the model
         n_epochs = 10
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,


Batch: 49 loss: 4.7207508087158275
Batch: 99 loss: 4.2542777061462425
Batch: 149 loss: 3.8573336601257324
Batch: 199 loss: 3.5442888736724854
Batch: 249 loss: 3.3090047836303718
Batch: 267 loss: 3.2427451610565186Epoch: 1         Training Loss: 3.242745         Validation L
valid_loss decreased inf -> 1.4943279027938843
saving the model...

Batch: 49 loss: 1.7748415470123296
Batch: 99 loss: 1.7255934476852417
Batch: 149 loss: 1.6986690759658813
Batch: 199 loss: 1.6773021221160889
Batch: 249 loss: 1.6547222137451172
Batch: 267 loss: 1.6518464088439941Epoch: 2         Training Loss: 1.651846         Validation L
valid_loss decreased 1.4943279027938843 -> 1.077866554260254
saving the model...

Batch: 49 loss: 1.2054772377014166
Batch: 99 loss: 1.2087248563766486
Batch: 149 loss: 1.2009099721908574
Batch: 199 loss: 1.1922475099563599
Batch: 249 loss: 1.2294436693191528
Batch: 267 loss: 1.2387566566467285Epoch: 3         Training Loss: 1.238757         Validation L
valid_loss decreased 1.077866554260254 -> 0.9669297933578491
saving the model...

Batch: 49 loss: 1.0290896892547607
Batch: 99 loss: 1.0314964056015015
Batch: 149 loss: 1.0443055629730225
```

```
Batch: 199 loss: 1.0494900941848755
Batch: 249 loss: 1.0450109243392944
Batch: 267 loss: 1.0503487586975098Epoch: 4         Training Loss: 1.050349         Validation L
valid_loss decreased 0.9669297933578491 -> 0.9337984323501587
saving the model...

Batch: 49 loss: 0.8161781430244446
Batch: 99 loss: 0.8360822796821594
Batch: 149 loss: 0.8608890175819397
Batch: 199 loss: 0.8591322898864746
Batch: 249 loss: 0.8554588556289673
Batch: 267 loss: 0.8574138879776001Epoch: 5         Training Loss: 0.857414         Validation L
valid_loss decreased 0.9337984323501587 -> 0.8515341281890869
saving the model...

Batch: 49 loss: 0.7522447109222412
Batch: 99 loss: 0.7083674669265747
Batch: 149 loss: 0.7436771392822266
Batch: 199 loss: 0.7569662332534792
Batch: 249 loss: 0.7655176520347595
Batch: 267 loss: 0.7706692218780518Epoch: 6         Training Loss: 0.770669         Validation L

Batch: 49 loss: 0.6297307014465332
Batch: 99 loss: 0.6340252161026001
Batch: 149 loss: 0.6476740837097168
Batch: 199 loss: 0.6615803837776184
Batch: 249 loss: 0.6758080124855042
Batch: 267 loss: 0.6770513653755188Epoch: 7         Training Loss: 0.677051         Validation L

Batch: 49 loss: 0.61475628614425667
Batch: 99 loss: 0.5867264866828918
Batch: 149 loss: 0.5880090594291687
Batch: 199 loss: 0.5918093919754028
Batch: 249 loss: 0.6013051867485046
Batch: 267 loss: 0.6125923395156863Epoch: 8         Training Loss: 0.612592         Validation L

Batch: 49 loss: 0.54696905612945567
Batch: 53 loss: 0.5449297428131104


        ---------------------------------------------------------------------------

        KeyboardInterrupt                         Traceback (most recent call last)

        <ipython-input-47-8f7e286a8b43> in <module>()
          1 # train the model
          2 n_epochs = 10
    ----> 3 model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfe
```

```
    <ipython-input-31-90233d062825> in train(n_epochs, loaders, model, optimizer, criterion,
     15         ####################
     16         model.train()
---> 17         for batch_idx, (data, target) in enumerate(loaders['train']):
     18             # move to GPU
     19             if use_cuda:


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
     99         """
    100         path, target = self.samples[index]
--> 101         sample = self.loader(path)
    102         if self.transform is not None:
    103             sample = self.transform(sample)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    145         return accimage_loader(path)
    146     else:
--> 147         return pil_loader(path)
    148
    149


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    128     with open(path, 'rb') as f:
    129         img = Image.open(f)
--> 130         return img.convert('RGB')
    131
    132
```

```
/opt/conda/lib/python3.6/site-packages/PIL/Image.py in convert(self, mode, matrix, dithe
890         """
891
--> 892         self.load()
893
894         if not mode and self.mode == "P":


/opt/conda/lib/python3.6/site-packages/PIL/ImageFile.py in load(self)
233
234                             b = b + s
--> 235                             n, err_code = decoder.decode(b)
236                             if n < 0:
237                                 break


KeyboardInterrupt:
```

```
In [51]: # load the model that got the best validation accuracy (uncomment the line below)
         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%.

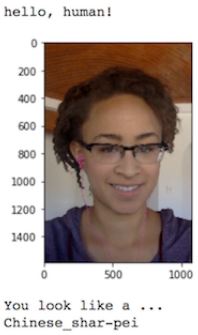
```
In [52]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.884196


Test Accuracy: 71% (600/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [53]: ### 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]
         # print(class_names)
         def predict_breed_transfer(img_path):
```



Sample Human Output

```
# load the image and return the predicted breed
image = Image.open(img_path).convert('RGB')
in_transform = transforms.Compose([transforms.Resize((256, 256)),
                    transforms.CenterCrop(224),
                    transforms.ToTensor(),
                    transforms.Normalize((0.485, 0.456, 0.406),
                                         (0.229, 0.224, 0.225))])
image = in_transform(image)[:3,:,:].unsqueeze(0)
image = image.to(device)
output = model_transfer(image)
_, class_index = output.topk(1)
class_index.cpu()
return class_names[class_index]
```

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 [55]:
```
### 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
    image = Image.open(img_path).convert('RGB')
    im_list = np.asarray(image)
```

```
    plt.imshow(im_list)
    title = ""

    if dog_detector(img_path):
        title = "This is a dog!"
    elif face_detector(img_path):
        title = "This is a human!"
    else:
        title = "This is neither a dog nor a human!"

    dogname = predict_breed_transfer(img_path)
    plt.title(title + "\nYou look like a {}".format(dogname))
    plt.show()
```

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.

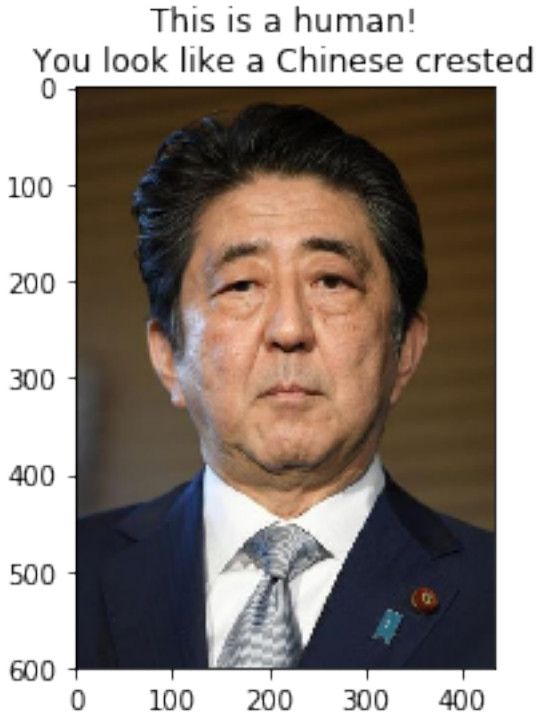
Answer: (Three possible points for improvement)

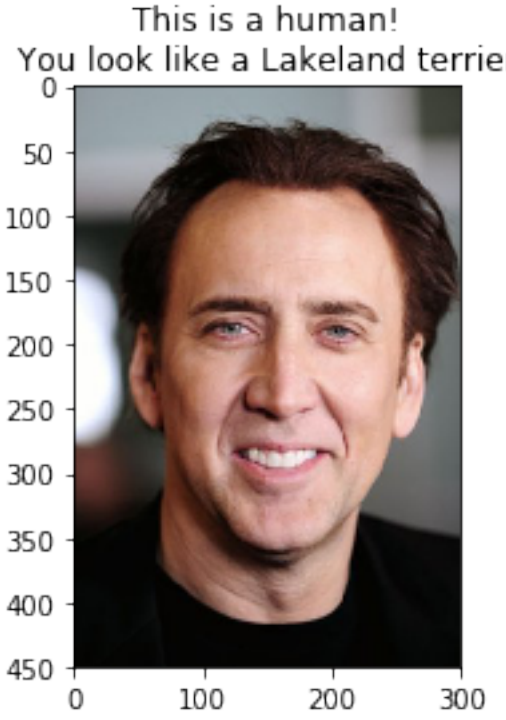
The accuracy is within my expectation.

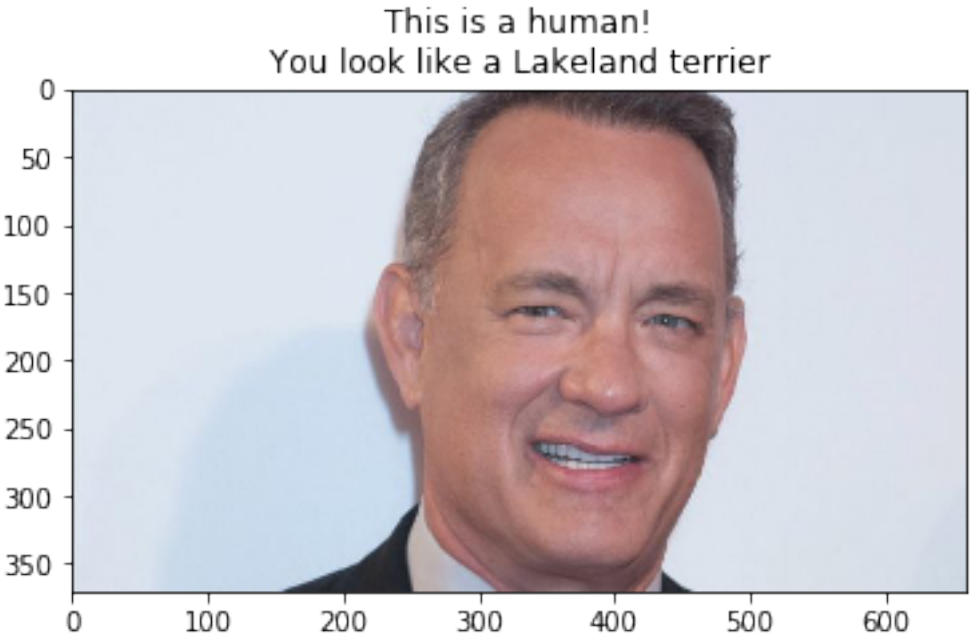
To get better result, I might 1. Feed more various images to the model 2. try ensamble learning to oversome overfitting https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/ 3. try the state of the art algorithm like GAN

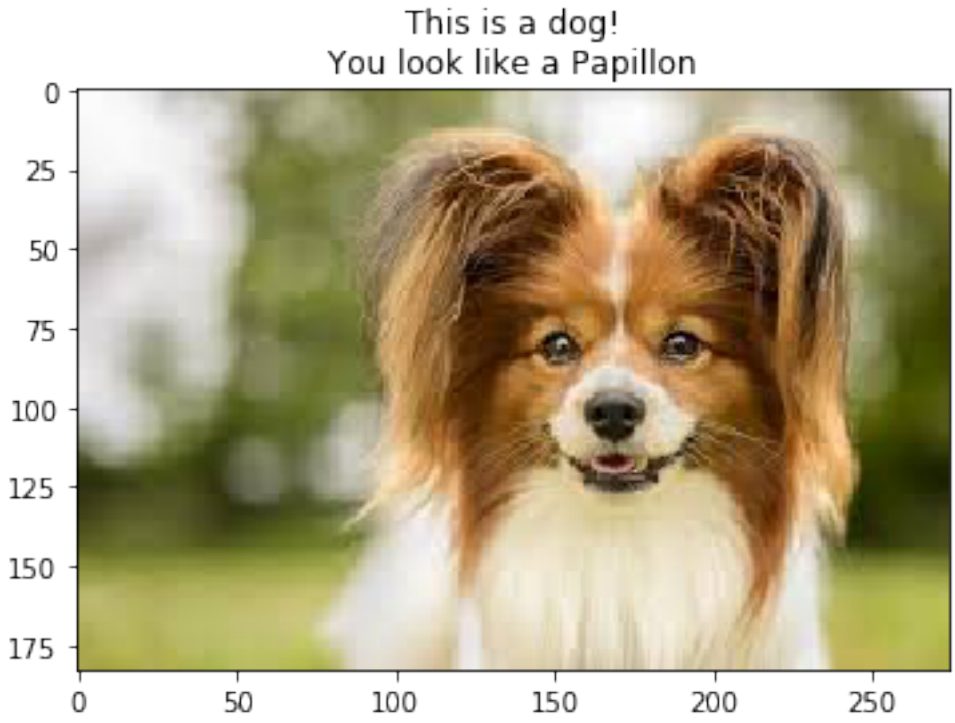
In [59]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.


file_path_list = [
    "./images/abe.jpg",
    "./images/nic.jpg",
    "./images/tom.jpg",
    "./images/dog_1.jpg",
    "./images/dog_2.jpg",
    "./images/dog_3.jpg"
]
for file in file_path_list:
    run_app(file)
```

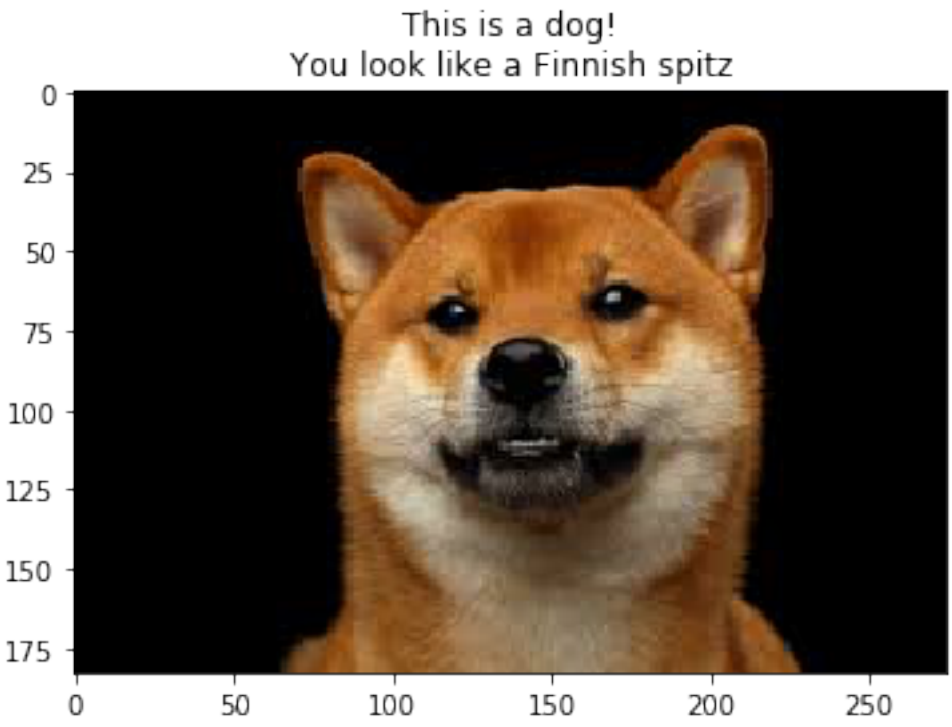
```
## suggested code, below
# for file in np.hstack((human_files[:3], dog_files[:3])):
#     run_app(file)
```

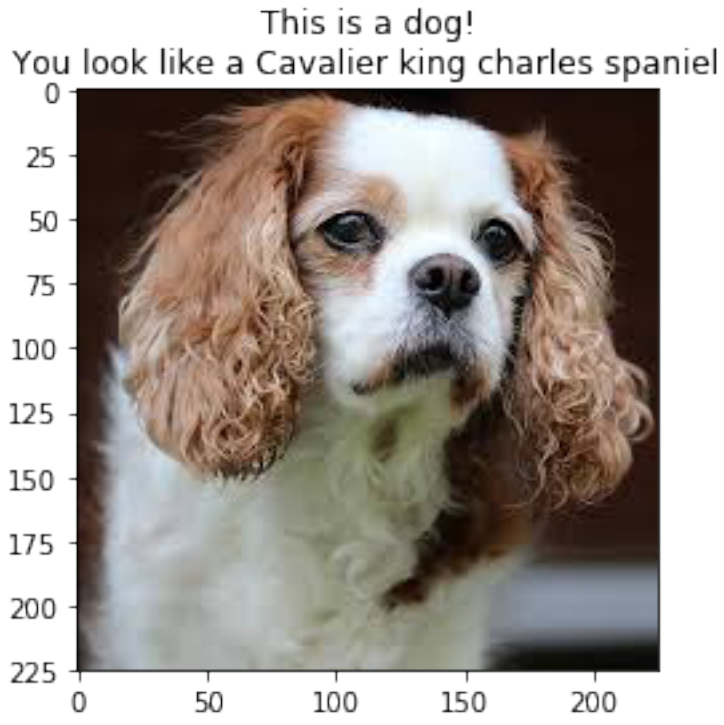





This is a human!
You look like a Lakeland terrier
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600


This is a dog!
You look like a Papillon
0
25
50
75
100
125
150
175
0
50
100
150
200
250


This is a dog!
You look like a Finnish spitz
0
25
50
75
100
125
150
175
0
50
100
150
200
250


This is a dog!
You look like a Cavalier king charles spaniel
0
25
50
75
100
125
150
175
200
225
0
50
100
150
200

In [ ]: