dog_app

May 2, 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 [3]: 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

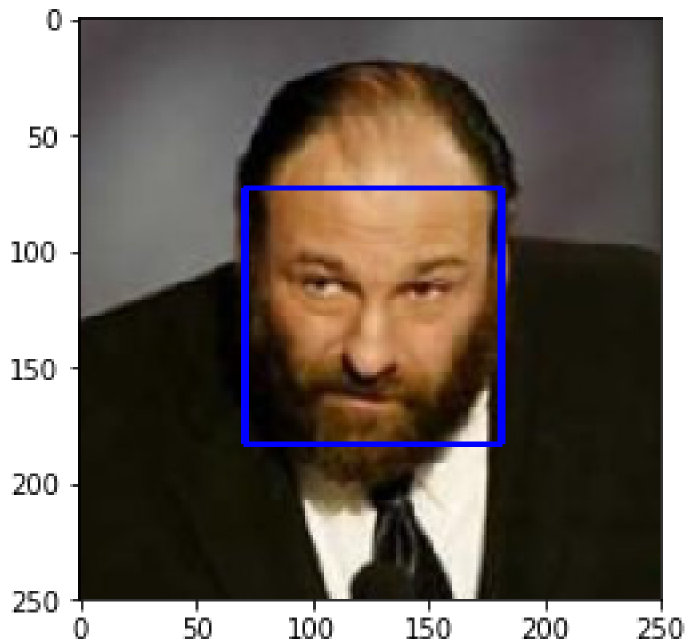
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [4]: 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[475])
        # 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 [5]: # 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 [6]: from tqdm import tqdm

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

        #-#-# Do NOT modify the code above this line. #-#-#

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.


        faces_detected_humans = 0
        faces_detected_dogs = 0

        for ii in range(100):
            if face_detector(human_files_short[ii]):
                faces_detected_humans +=1
            if face_detector(dog_files_short[ii]):
                faces_detected_dogs +=1
        print(f"Detected faces in dog_files: {faces_detected_dogs}%")
        print(f"Detected faces in human_files: {faces_detected_humans}%")
```

```
Detected faces in dog_files: 17%
Detected faces in human_files: 98%
```

In [ ]:

In [ ]:

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

In [ ]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

In [7]:
```
import torch
import torchvision.models as models

# define VGG16 model
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 [00:06<00:00, 87460996.25it/s]
```

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

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

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

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

In [8]:
```
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
    '''

    img_pil = Image.open(img_path)

    data_transform = transforms.Compose([transforms.Resize(255),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406],
                                                              [0.229, 0.224, 0.225])])

    #     Image.Image.show(img_pil)
    img_tensor = data_transform(img_pil)
    img_tensor.unsqueeze_(0)

    if use_cuda:
        img_tensor = img_tensor.cuda()

    output = VGG16(img_tensor)
    _, preds_tensor = torch.max(output, 1)
    preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso


    return preds.item() # predicted class index

# VGG16_predict('data/dogs/train/001.Affenpinscher/Affenpinscher_00001.jpg')
```

In [9]: VGG16_predict(dog_files_short[90])

Out[9]: 236

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 [10]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    if VGG16_predict(img_path) in range(151, 268):
        return True # true/false
    else:
        return False
```

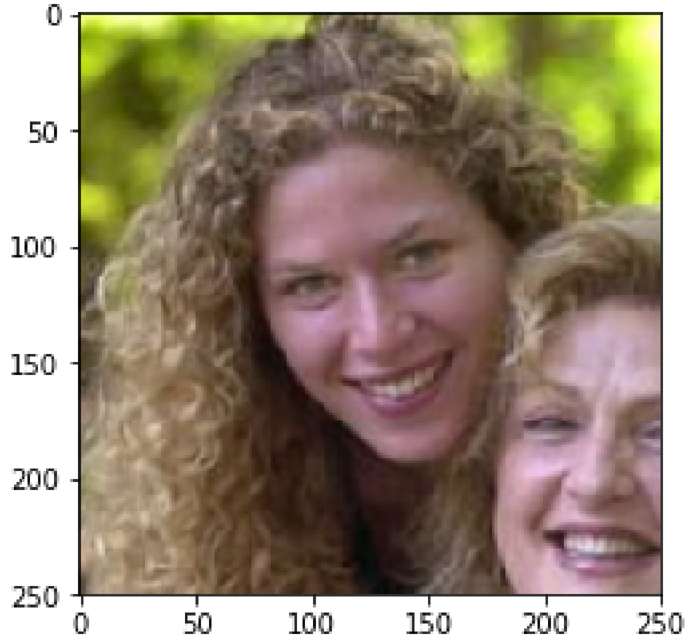
1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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 [11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
detected_dogs_in_humans = 0
detected_dogs_in_dogs = 0

for ii in range(100):
    if dog_detector(human_files_short[ii]):
        detected_dogs_in_humans += 1
        print(f"This human ({ii}) looks like a dog")
        human_dog_image = Image.open(human_files_short[ii])
        plt.imshow(human_dog_image)
        plt.show()
    if dog_detector(dog_files_short[ii]):
        detected_dogs_in_dogs +=1

print (f"Percentage of the images in human_files_short that have a detected dog: {detec
print (f"Percentage of the images in dog_files_short that have a detected dog: {detecte
```

```
This human (88) looks like a dog
```




```
Percentage of the images in human_files_short that have a detected dog: 1%
Percentage of the images in dog_files_short that have a detected dog: 100%
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [ ]: ### (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 [12]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
from PIL import ImageFile



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

ImageFile.LOAD_TRUNCATED_IMAGES = True

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

# Declare the transforms for train, valid and test sets.
```

```
# Convert to Tensor
# Normalize images because the values of images should be loaded between [0 - 1]
transforms = {

    # RandomHorizontalFlip() & RandomRotation() to augement data in train transformatic
    'train' : transforms.Compose([transforms.Resize(256),
                                  transforms.RandomResizedCrop(224),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.RandomRotation(10),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])]),

    'valid' : transforms.Compose([transforms.Resize(256),
                                  transforms.CenterCrop(224),
                                  transforms.ToTensor(),
                                  transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])]),

    'test' : transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                      std=[0.229, 0.224, 0.225])])
}

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20

# Create image datasets (train, valid, test)
image_datasets = {x: datasets.ImageFolder(os.path.join('/data/dog_images', x), transfor
                  for x in ['train', 'valid', 'test']}

# Create data loaders (train, valid, test)
data_loaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size
                                               shuffle=True, num_workers=num_workers)
                for x in ['train', 'valid', 'test']}
```

In [13]:
```
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid', 'test']}
num_classes = len(image_datasets['train'].classes)

print('Number of records of training dataset: {}'.format(dataset_sizes['train']))
print('Number of records of validation dataset: {}'.format(dataset_sizes['valid']))
print('Number of records of test dataset: {}'.format(dataset_sizes['test']))
print('Number of dog classes: {}'.format(num_classes))
```

```
Number of records of training dataset: 6680
```

```
Number of records of validation dataset: 835
Number of records of test dataset: 836
Number of dog classes: 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 resizing to 256px then cropping them to 224x224 px. Resizing increases the processing time.
- Yes, I augumented the dataset through horizontal flips and rotations.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [14]: 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

                 # convolutional layer, e.g: nn.Conv2d(in_channels, out_channels, kernel_size, s
                 2

                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

                 # max pooling layer, to down-sample an input representation
                 # nn.MaxPool2d(kernel_size, stride)
                 self.pool = nn.MaxPool2d(2, 2)

                 # fully-connected
                 self.fc1 = nn.Linear(7*7*128, 512)
                 self.fc2 = nn.Linear(512, num_classes)

                 # drop-out
                 self.dropout = nn.Dropout(0.25)

             def forward(self, x):
```

```
        ## Define forward behavior
        # shape 224
        x = F.relu(self.conv1(x)) # shape 112
        x = self.pool(x) # shape 56
        x = F.relu(self.conv2(x)) # shape 28
        x = self.pool(x) # size 14
        x = F.relu(self.conv3(x)) # shape 14
        x = self.pool(x) # shape 7

        # flatten
        # -1 means inferring the size from other dimensions.
        x = x.view(x.size(0), -1) # or x = x.view(-1, 7*7*128)

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

        x = self.dropout(x)
        x = self.fc2(x)
        return x

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

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

# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

Answer:

I used three convolutional layers in the model. Added maxpooling to downsample by a factor of 2 after each layer. Then, 25% dropout added before each fully connected layer to avoid 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 [15]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

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

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     # clear the gradients of all optimized variables, initialize weights to zer
                     optimizer.zero_grad()
                     # forward pass
                     output = model(data)
                     # calculate batch loss
                     loss = criterion(output, target)
                     # backward pass
                     loss.backward()
                     # parameter update
```

```
            optimizer.step()
            # update training loss
            train_loss += loss.item() * data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass
            output = model(data)
            # batch loss
            loss = criterion(output, target)
            # update validation loss
            valid_loss += loss.item() * data.size(0)

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

        # 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...'.
                 format(valid_loss_min, valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(15, data_loaders, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.486494        Validation Loss: 4.326636
Validation loss decreased (inf --> 4.326636).    Saving model...
Epoch: 2         Training Loss: 4.462603        Validation Loss: 4.309030
Validation loss decreased (4.326636 --> 4.309030).    Saving model...
Epoch: 3         Training Loss: 4.433697        Validation Loss: 4.284525
Validation loss decreased (4.309030 --> 4.284525).    Saving model...
Epoch: 4         Training Loss: 4.398346        Validation Loss: 4.287289
Epoch: 5         Training Loss: 4.390127        Validation Loss: 4.183188
Validation loss decreased (4.284525 --> 4.183188).    Saving model...
Epoch: 6         Training Loss: 4.355791        Validation Loss: 4.165607
Validation loss decreased (4.183188 --> 4.165607).    Saving model...
Epoch: 7         Training Loss: 4.320226        Validation Loss: 4.162504
Validation loss decreased (4.165607 --> 4.162504).    Saving model...
Epoch: 8         Training Loss: 4.303936        Validation Loss: 4.125225
Validation loss decreased (4.162504 --> 4.125225).    Saving model...
Epoch: 9         Training Loss: 4.269098        Validation Loss: 4.098028
Validation loss decreased (4.125225 --> 4.098028).    Saving model...
Epoch: 10         Training Loss: 4.256041        Validation Loss: 4.094284
Validation loss decreased (4.098028 --> 4.094284).    Saving model...
Epoch: 11         Training Loss: 4.248190        Validation Loss: 4.106730
Epoch: 12         Training Loss: 4.208746        Validation Loss: 4.044650
Validation loss decreased (4.094284 --> 4.044650).    Saving model...
Epoch: 13         Training Loss: 4.190118        Validation Loss: 4.054092
Epoch: 14         Training Loss: 4.170426        Validation Loss: 3.985852
Validation loss decreased (4.044650 --> 3.985852).    Saving model...
Epoch: 15         Training Loss: 4.149206        Validation Loss: 3.948996
Validation loss decreased (3.985852 --> 3.948996).    Saving model...
```

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 [19]: 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(data_loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.946120


Test Accuracy: 11% (100/836)
```

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 [62]: ## TODO: Specify data loaders
         transfer_loaders = data_loaders.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 [63]: import torchvision.models as models
         import torch.nn as nn
```

```
## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained=True)

# Freeze parameters so we don't backprop through them
for param in model_transfer.parameters():
    param.requires_grad = False

# Replace the last fully connected layer with a linear layer with 133 out features
model_transfer.fc = nn.Linear(2048, 133)


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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:03<00:00, 26344619.69it/s]
```

In [64]:
```
model_transfer
```

Out[64]:
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
```

```
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (relu): ReLU(inplace)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
      (relu): ReLU(inplace)
    )
```

```
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
    (fc): Linear(in_features=2048, out_features=133, bias=True)
  )
```

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

Answer:

I used ResNet50 because it was trained on ImageNet for classification of a wide range of objects making it an efficient way to tackle computer vision problems.

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 [65]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [68]: # train the model
         model_transfer =  train(10, transfer_loaders, model_transfer, optimizer_transfer, crite

Epoch: 1          Training Loss: 2.809478         Validation Loss: 1.011981
Validation loss decreased (inf --> 1.011981).    Saving model...
Epoch: 2          Training Loss: 1.546248         Validation Loss: 0.843417
Validation loss decreased (1.011981 --> 0.843417).    Saving model...
Epoch: 3          Training Loss: 1.340143         Validation Loss: 0.596983
Validation loss decreased (0.843417 --> 0.596983).    Saving model...
Epoch: 4          Training Loss: 1.267726         Validation Loss: 0.635504
Epoch: 5          Training Loss: 1.236588         Validation Loss: 0.632336
Epoch: 6          Training Loss: 1.196741         Validation Loss: 0.517217
Validation loss decreased (0.596983 --> 0.517217).    Saving model...
Epoch: 7          Training Loss: 1.165586         Validation Loss: 0.544164
Epoch: 8          Training Loss: 1.136808         Validation Loss: 0.569028
Epoch: 9          Training Loss: 1.140126         Validation Loss: 0.606187
Epoch: 10          Training Loss: 1.130488         Validation Loss: 0.653638
```

```
In [71]: # 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 [73]: test(transfer_loaders, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.522034


Test Accuracy: 84% (707/836)
```

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.

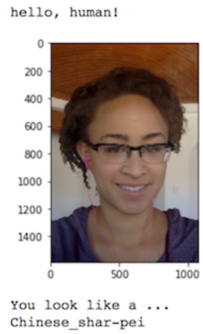
```
In [77]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.
         from torchvision import transforms
         from PIL import Image
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path)
             transform = transforms.Compose([transforms.Resize(224),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize([0.485, 0.456, 0.406],
                                                                  [0.229, 0.224, 0.225])])
             img_as_tensor = transform(img)
             img_as_tensor = img_as_tensor.unsqueeze_(0)

             if use_cuda:
                 img_as_tensor = img_as_tensor.cuda()
             output = model_transfer(img_as_tensor)
             _, preds_tensor = torch.max(output, 1)
             preds = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tens
             return class_names[preds]
```

```
In [82]: import matplotlib.pyplot as plt
         # quick test on humans
         fig = plt.figure(figsize=(20, 20))
```

```
for idx in np.arange(8):
    ax = fig.add_subplot(5, 20/5, idx+1, xticks=[], yticks=[])
    plt.imshow(Image.open(human_files[idx]))
    ax.set_title(f"probably a(n) {predict_breed_transfer(human_files[idx])}")
```


```
In [81]: # quick test on dogs
         fig = plt.figure(figsize=(20, 20))
         for idx in np.arange(8):
             ax = fig.add_subplot(5, 20/5, idx+1, xticks=[], yticks=[])
             plt.imshow(Image.open(dog_files[idx*60]))
             ax.set_title(f"Probably a(n) {predict_breed_transfer(dog_files[idx*50])}")
```

Step 5: Write your Algorithm



Sample Human Output

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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [94]:
```
### 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
    fig, ax = plt.subplots()
    title = ''
    if dog_detector(img_path):
        title = f"It's a cute\n{predict_breed_transfer(img_path)}!"
    elif face_detector(img_path):
        title = f"hi, human!,\nYou look like... \na {predict_breed_transfer(img_path)}!
    else:
        title = 'I have no idea what I\'m looking at!'

    ax.set_title(title)
    ax.imshow(plt.imread(img_path))
    ax.spines['top'].set_visible(False)
    ax.spines['right'].set_visible(False)
    ax.spines['left'].set_visible(False)
    ax.spines['bottom'].set_visible(False)
    ax.tick_params(axis='both', length=0)
    ax.set_xticklabels('')
    ax.set_yticklabels('')
```

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)

With ResNet's accuracy of 84%, I'm satisfied with the results. :) 1. Could fine tune the hyper-parameters. 2. Maybe try a larger dataset. 3. Could try more image augumentations.

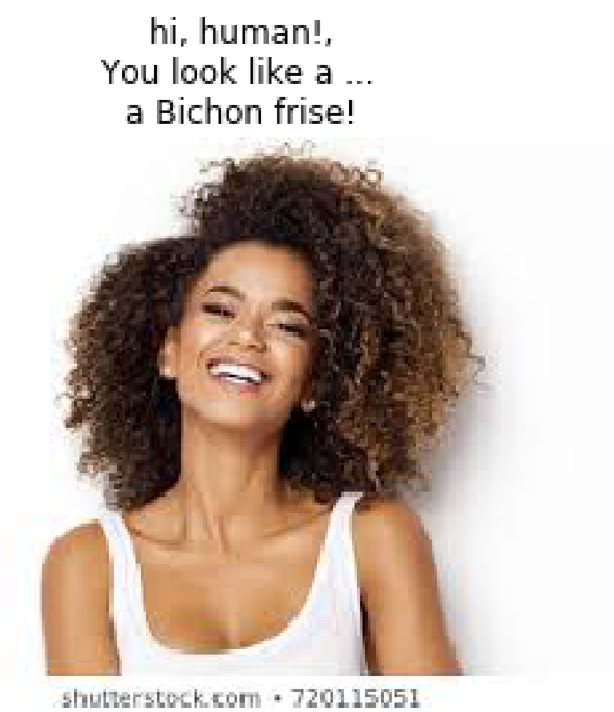
```
In [112]: ## TODO: Execute your algorithm from Step 6 on
          ## at least 6 images on your computer.
          ## Feel free to use as many code cells as needed.

          ## suggested code, below
          for file in np.hstack((human_files[:2], dog_files[:2])):
              run_app(file)
```

hi, human!,
You look like a ...
a Bloodhound!

hi, human!,
You look like a ...
a Ibizan hound!

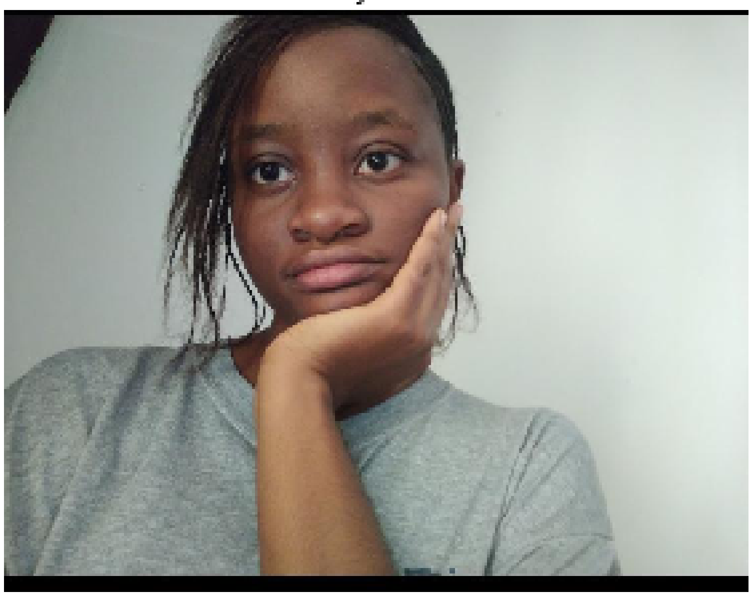
It's a cute
Mastiff!


```
In [113]: run_app("images.jpg")
```



```
In [114]: run_app("me.jpeg")
```

hi, human!,
You look like a ...
a Greyhound!

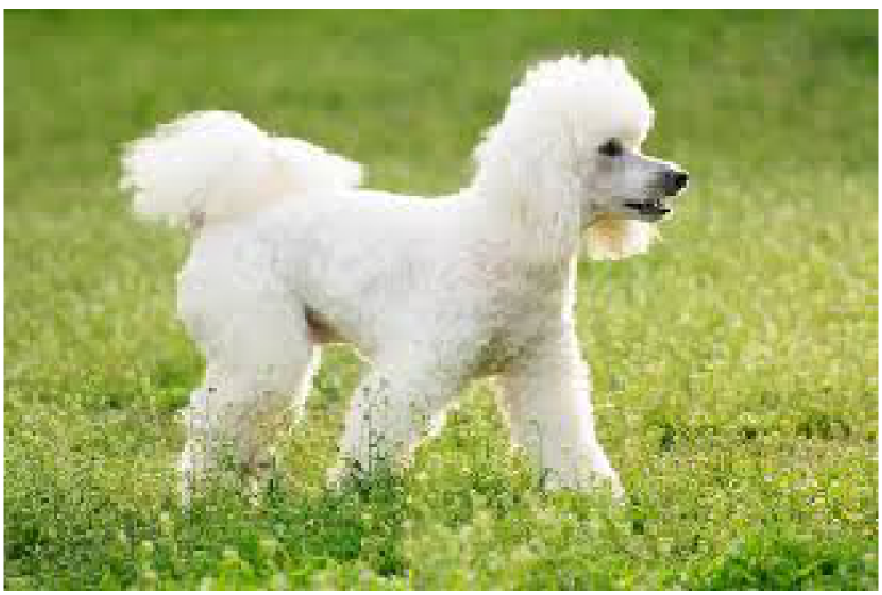


```
In [115]: run_app("elon.jpg")
```

hi, human!,
You look like a ...
a Poodle!

```
In [116]: run_app("one.jpg")
```

It's a cute
Bichon frise!



```
In [117]: run_app("susan.jpg")
```

hi, human!,
You look like a ...
a German pinscher!

```
In [118]: run_app("two.jpg")
```

It's a cute
Chihuahua!

```
In [ ]:
```