dog_app

October 23, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

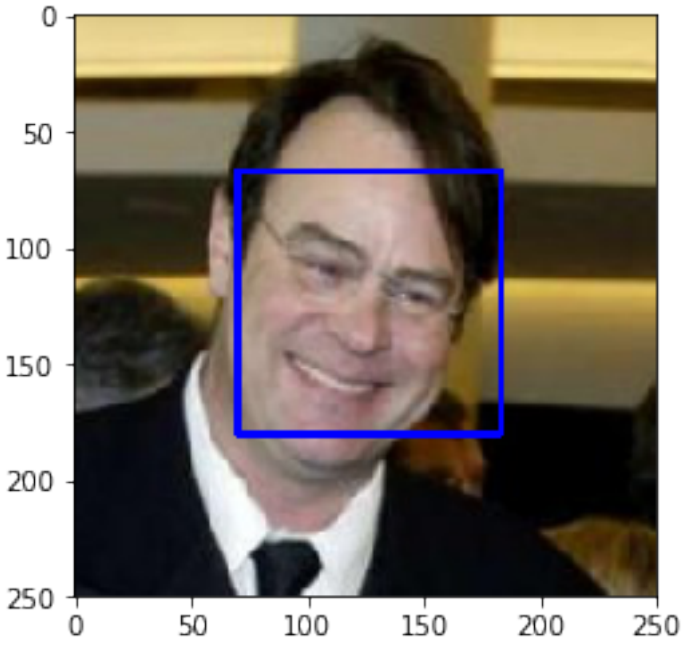
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %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 [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

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

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

        num_of_observations = len(human_files_short)

        human_face_in_human_files_count = 0.0
        dog_face_in_dog_files_count = 0.0
        for i in range(num_of_observations):
            # for human dataset
            if face_detector(human_files_short[i]):
                human_face_in_human_files_count += 1.0

            # for dog dataset
            if face_detector(dog_files_short[i]):
                dog_face_in_dog_files_count += 1.0
        print("Number of human faces detected in human files: % 2.2d%%" %(human_face_in_human_fi
        print("Number of human faces detected in dog files: % 2.2d%%" %(dog_face_in_dog_files_co
```

```
Number of human faces detected in human files:  98%
Number of human faces detected in dog files:  17%
```

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

```
In [5]: ### (Optional)
        ### TODO: Test performance of another face detection algorithm.
        ### Feel free to use as many code cells as needed.
```

```
In [6]: !pip install dlib==19.16.0
```

```
Collecting dlib==19.16.0
  Downloading https://files.pythonhosted.org/packages/35/8d/e4ddf60452e2fb1ce3164f774e68968b3f11
    100% || 3.3MB 10.3MB/s ta 0:00:01
Building wheels for collected packages: dlib
  Running setup.py bdist_wheel for dlib ... done
  Stored in directory: /root/.cache/pip/wheels/ce/f9/bc/1c51cd0b40a2b5dfd46ab79a73832b41e7c3aa91
Successfully built dlib
Installing collected packages: dlib
Successfully installed dlib-19.16.0
```

```
In [7]: import dlib # version 19.16.0 btw

        def face_detector_2(img_path):
            detector = dlib.get_frontal_face_detector()
            img = dlib.load_rgb_image(img_path)
            # The 1 in the second argument indicates that we should upsample the image
            # 1 time.  This will make everything bigger and allow us to detect more
            # faces.
            dets = detector(img, 1)

            return len(dets) > 0
```

```
In [8]: num_of_observations = len(human_files_short)

        human_face_in_human_files_count = 0.0
        dog_face_in_dog_files_count = 0.0

        dog_images_with_humans_detected = []
        for i in range(num_of_observations):
            # for human dataset
            if face_detector_2(human_files_short[i]):
                human_face_in_human_files_count += 1.0
```

```
            # for dog dataset
            if face_detector_2(dog_files_short[i]):
                # append files with humans detected to further investigate
                # why the image is detected human (maybe the image has human
                # in the background) since the behavior is pretty consistent
                # across multiple runs.
                dog_images_with_humans_detected.append(dog_files_short[i])
                print("Image with human detected: " + dog_files_short[i])
                dog_face_in_dog_files_count += 1.0

        print("Number of human faces detected in human files (second face detector): % 0.2d%%" %
        print("Number of human faces detected in dog files (second face detector): % 0.2d%%" %(d
```

```
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06865.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06813.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06841.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06811.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06860.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06829.jpg
Image with human detected: /data/dog_images/train/103.Mastiff/Mastiff_06872.jpg
Image with human detected: /data/dog_images/train/059.Doberman_pinscher/Doberman_pinscher_04174.
Number of human faces detected in human files (second face detector):  100%
Number of human faces detected in dog files (second face detector):  08%
```

As you can see, face detector 2 got the most accuracy in terms of detecting faces. I used dlib face detection library for this. You can find more information about dlib here and the sample code in C++ here.

Now what I like to investigate is how dlib algorithm got 8 dogs misclassified as human. As you can see, just like the first face_detector function's implementation, dlib just returns binary classification by comparing if the image has human or not in the image. It does not care whether the image is primarily dog but as long as it detects human, it will still classify the image as human.

I saved the image locations in the iteration of 100 images per class. Once it detects human in the dog image, it prints the file location and appends it to the list. The following is the code for finding out if dog images has/have humans in them.

```
In [9]: import matplotlib.image as mpimg

        # plot the images in the batch, along with the corresponding labels
        fig = plt.figure(figsize=(24, 10))
        for idx in np.arange(8):
            ax = fig.add_subplot(2, 8/2, idx+1, xticks=[], yticks=[])
            img = mpimg.imread(dog_images_with_humans_detected[idx])
            plt.imshow(img)
```

Well that proves my theory. Most dog images flagged as human actually has human in it. There are three images that are misclassified but that is alright. This is still most accurate implementation for human detection.

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 [10]: 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 [00:40<00:00, 13706962.71it/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 [11]: from PIL import Image
         import torchvision.transforms as transforms
         import operator

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

             Args:
                 img_path: path to an image

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

             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image
             img = Image.open(img_path)

             unnormalize = transforms.Normalize(
                                        mean=[-0.485/0.229, -0.456/0.224, -0.406/0.255],
                                        std=[1/0.229, 1/0.224, 1/0.255]
                                    )

             transformer = transforms.Compose([
                                    transforms.Resize((224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                         std=[0.229, 0.224, 0.225]),
         #                             unnormalize,
         #                             transforms.ToPILImage()
                                ])
             img = transformer(img)

             # to add batch dimensioning as per the following URL
             # https://discuss.pytorch.org/t/expected-stride-to-be-a-single-integer-value-or-a-l
             img = img.unsqueeze_(0)
```

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

    predictions = VGG16(img)

    # since predictions are of type Variable when subscripted, it cannot be directly
    # converted to numpy that is why we need to detach
    if use_cuda:
        predictions = predictions[0].detach().to('cpu').numpy()
    else:
        predictions = predictions[0].detach().numpy()

    # find the index as well as the value of the highest prediction
    # using operator
    # https://stackoverflow.com/questions/6193498/pythonic-way-to-find-maximum-value-an
    index, value = max(enumerate(predictions), key=operator.itemgetter(1))

    return index # predicted class index
```

```
In [12]: VGG16_predict(dog_files_short[0])
```

```
Out[12]: 243
```

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 [13]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             index = VGG16_predict(img_path)

             if index >= 151 and index <= 268:
                 return True
             return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [14]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         dog_in_human_files_count = 0
         dog_in_dog_files_count = 0

         for i in range(len(human_files_short)):
             if dog_detector(human_files_short[i]):
                 dog_in_human_files_count += 1
             if dog_detector(dog_files_short[i]):
                 dog_in_dog_files_count += 1

In [15]: print("Accuracy of dog detector when applied to human files: %d%%" %(100 - dog_in_human
         print("Accuracy of dog detector when applied to dog files: %d%%" %(dog_in_dog_files_cou

Accuracy of dog detector when applied to human files: 99%
Accuracy of dog detector when applied to dog files: 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 [16]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.

         # define ResNet-50 model
         resnet50 = models.resnet50(pretrained=True)

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

         # move model to GPU if CUDA is available
         if use_cuda:
             resnet50 = resnet50.cuda()

Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:02<00:00, 45088597.66it/s]


In [17]: def resnet50_predict(img_path):
             '''
             Use pre-trained VGG-16 model to obtain index corresponding to
             predicted ImageNet class for image at specified path

             Args:
                 img_path: path to an image
```

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    img = Image.open(img_path)

    unnormalize = transforms.Normalize(
                                mean=[-0.485/0.229, -0.456/0.224, -0.406/0.255],
                                std=[1/0.229, 1/0.224, 1/0.255]
                            )

    transformer = transforms.Compose([
                                        transforms.Resize(256),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize(mean=[0.485, 0.456, 0.406]
                                    ])
    img = transformer(img)

    # to add batch dimensioning as per the following URL
    # https://discuss.pytorch.org/t/expected-stride-to-be-a-single-integer-value-or-a-l
    img = img.unsqueeze_(0)

    if use_cuda:
        img = img.cuda()

    # Strangely, the pretrained model does always predict bucket/pail.
    # Obviously, dog is not bucket/pail :D
    # The following link gives me a very straightforward answer:
    # https://discuss.pytorch.org/t/pretrained-resnet-constant-output/2760
    resnet50.eval()

    with torch.no_grad():
        predictions = resnet50(img)

    # since predictions are of type Variable when subscripted, it cannot be directly
    # converted to numpy that is why we need to detach
    if use_cuda:
        predictions = predictions[0].detach().to('cpu').numpy()
    else:
        predictions = predictions[0].detach().numpy()

    # find the index as well as the value of the highest prediction
    # using operator
    # https://stackoverflow.com/questions/6193498/pythonic-way-to-find-maximum-value-an
```

```
            index, value = max(enumerate(predictions), key=operator.itemgetter(1))

            return index # predicted class index
```

```
In [18]: resnet50_predict(dog_files_short[1])
```

```
Out[18]: 243
```

```
In [19]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector_resnet50(img_path):
             index = resnet50_predict(img_path)
             if index >= 151 and index <= 268:
                 return True
             return False
```

```
In [20]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         dog_in_human_files_count_resnet50 = 0
         dog_in_dog_files_count_resnet50 = 0

         for i in range(len(human_files_short)):
             if dog_detector_resnet50(human_files_short[i]):
                 dog_in_human_files_count_resnet50 += 1
             if dog_detector_resnet50(dog_files_short[i]):
                 dog_in_dog_files_count_resnet50 += 1
```

```
In [21]: print("======================USING RESNET50=======================")
         print("Accuracy of dog detector when applied to human files: %d%%" %(100 - dog_in_human
         print("Accuracy of dog detector when applied to dog files: %d%%" %(dog_in_dog_files_cou
```

```
======================USING RESNET50=======================
Accuracy of dog detector when applied to human files: 100%
Accuracy of dog detector when applied to dog files: 100%
```

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 [22]: import os

         # number of dog classes
         len(os.listdir('/data/dog_images/train'))

Out[22]: 133

In [23]: from torchvision import datasets
         from PIL import ImageFile

         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

```
test_valid_transforms = transforms.Compose([transforms.Resize(255),
                                            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=train_transforms)
test_data = datasets.ImageFolder('/data/dog_images/test', transform=test_valid_transfor
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=test_valid_transf

trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32)
validloader = torch.utils.data.DataLoader(valid_data, batch_size=32)

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

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: In training transform, I used random rotation as data augmentation technique that will rotate images by 30 degrees. Then after rotation, I used RandomResizedCrop to crop images in 224 x 224 sizes, Finally, resulting data are converted to tensor and I utilized normalization to contain RGB values in 0-1 range with means and stds acquired from the ImageNet training parameters.

Test and validation datasets do not need data augmentation since we normally feed real-world data during prediction. By going that way, we can grasp quantitatively how accurate our model is with real-world inputs. So I just used resizing, center-cropping to center zoomed out images to clearly see the dog breed to predict, the usual conversion of image data to tensor and normalization to prevent exploding/vanishing gradient problem with extreme values during gradient calculation and weight updates.

1.1.8 (IMPLEMENTATION) Model Architecture

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

In [24]:
```
torch.cuda.empty_cache()
```

In [25]:
```
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)
            self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
            self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
            self.pool = nn.MaxPool2d(3, 3)
            self.fc1 = nn.Linear(64 * 8 * 8, 1024)
            self.fc2 = nn.Linear(1024, 512)
            self.fc3 = nn.Linear(512, 133)
            self.dropout = nn.Dropout(0.25)


        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            x = x.view(-1, 64 * 8 * 8)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = F.relu(self.fc2(x))
            x = self.dropout(x)
            x = self.fc3(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

```
In [26]: model_scratch
```

```
Out[26]: Net(
           (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (pool): MaxPool2d(kernel_size=3, stride=3, padding=0, dilation=1, ceil_mode=False)
           (fc1): Linear(in_features=4096, out_features=1024, bias=True)
           (fc2): Linear(in_features=1024, out_features=512, bias=True)
           (fc3): 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 reason-

ing at each step.

Answer:

My approach to CNN is similar to what we have been doing in class, in that the features are getting deeper and deeper as the convolution progresses from 16 to 32 to 64. My thinking is that the features such as the shapes are can be more extracted as the neural network goes deeper.

Also, just like what vgg16 implemented, I used convolutional kernels with 3 x 3 shape with stride of one. vgg16 has good reported accuracy that is why I implemented it in mine.

I also used max pooling layer per activated convolutions of 3 x 3 also just like the kernels to shrink down the size of convolutions bearable to the later fully connected networks.

Linear layers are shrink down to multiples of 2 from 4096 to 1024 to 512 to the number of predictors as this is the pattern that I have seen in several implementations not limited to PyTorch. I used dropout layers in between to control overfitting.

NOTE: Of course, colors and style elements are in the early convolutions and since differentiating factor of different dog breeds are primarily color, it must have higher convolutions. But surprisingly, the network got 25% accuracy when trained (see below)

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 [27]: 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 [29]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
```

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

    optimizer.zero_grad()
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()
    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
    output = model(data)
    # calculate the loss
    loss = criterion(output, target)
    # update running validation loss
    valid_loss += loss.item()*data.size(0)

# calculate average loss over an epoch
train_loss = train_loss/len(loaders['train'].sampler)
valid_loss = valid_loss/len(loaders['valid'].sampler)


# 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 ...'.fo
    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(100, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.848206         Validation Loss: 4.711152
Validation loss decreased (inf --> 4.711152).  Saving model ...
Epoch: 2         Training Loss: 4.741490         Validation Loss: 4.640937
Validation loss decreased (4.711152 --> 4.640937).  Saving model ...
Epoch: 3         Training Loss: 4.700927         Validation Loss: 4.603989
Validation loss decreased (4.640937 --> 4.603989).  Saving model ...
Epoch: 4         Training Loss: 4.654142         Validation Loss: 4.592833
Validation loss decreased (4.603989 --> 4.592833).  Saving model ...
Epoch: 5         Training Loss: 4.637500         Validation Loss: 4.543089
Validation loss decreased (4.592833 --> 4.543089).  Saving model ...
Epoch: 6         Training Loss: 4.576017         Validation Loss: 4.457306
Validation loss decreased (4.543089 --> 4.457306).  Saving model ...
Epoch: 7         Training Loss: 4.478208         Validation Loss: 4.286544
Validation loss decreased (4.457306 --> 4.286544).  Saving model ...
Epoch: 8         Training Loss: 4.384484         Validation Loss: 4.198633
Validation loss decreased (4.286544 --> 4.198633).  Saving model ...
Epoch: 9         Training Loss: 4.311645         Validation Loss: 4.199944
Epoch: 10         Training Loss: 4.236989         Validation Loss: 4.105891
Validation loss decreased (4.198633 --> 4.105891).  Saving model ...
Epoch: 11         Training Loss: 4.170686         Validation Loss: 4.059120
Validation loss decreased (4.105891 --> 4.059120).  Saving model ...
Epoch: 12         Training Loss: 4.103631         Validation Loss: 3.989876
Validation loss decreased (4.059120 --> 3.989876).  Saving model ...
Epoch: 13         Training Loss: 4.066977         Validation Loss: 3.922915
Validation loss decreased (3.989876 --> 3.922915).  Saving model ...
Epoch: 14         Training Loss: 4.032341         Validation Loss: 3.936182
Epoch: 15         Training Loss: 4.007343         Validation Loss: 3.829592
Validation loss decreased (3.922915 --> 3.829592).  Saving model ...
Epoch: 16         Training Loss: 3.959878         Validation Loss: 3.847633
Epoch: 17         Training Loss: 3.938544         Validation Loss: 3.880148
Epoch: 18         Training Loss: 3.871862         Validation Loss: 3.733067
Validation loss decreased (3.829592 --> 3.733067).  Saving model ...
Epoch: 19         Training Loss: 3.822917         Validation Loss: 3.756559
Epoch: 20         Training Loss: 3.818150         Validation Loss: 3.724278
Validation loss decreased (3.733067 --> 3.724278).  Saving model ...
```

```
Epoch: 21         Training Loss: 3.780034         Validation Loss: 3.682679
Validation loss decreased (3.724278 --> 3.682679).  Saving model ...
Epoch: 22         Training Loss: 3.740766         Validation Loss: 3.632796
Validation loss decreased (3.682679 --> 3.632796).  Saving model ...
Epoch: 23         Training Loss: 3.738192         Validation Loss: 3.689418
Epoch: 24         Training Loss: 3.719148         Validation Loss: 3.625513
Validation loss decreased (3.632796 --> 3.625513).  Saving model ...
Epoch: 25         Training Loss: 3.698142         Validation Loss: 3.582609
Validation loss decreased (3.625513 --> 3.582609).  Saving model ...
Epoch: 26         Training Loss: 3.663729         Validation Loss: 3.565333
Validation loss decreased (3.582609 --> 3.565333).  Saving model ...
Epoch: 27         Training Loss: 3.668590         Validation Loss: 3.538687
Validation loss decreased (3.565333 --> 3.538687).  Saving model ...
Epoch: 28         Training Loss: 3.600926         Validation Loss: 3.533484
Validation loss decreased (3.538687 --> 3.533484).  Saving model ...
Epoch: 29         Training Loss: 3.600114         Validation Loss: 3.494256
Validation loss decreased (3.533484 --> 3.494256).  Saving model ...
Epoch: 30         Training Loss: 3.573396         Validation Loss: 3.496124
Epoch: 31         Training Loss: 3.548670         Validation Loss: 3.522339
Epoch: 32         Training Loss: 3.533115         Validation Loss: 3.469489
Validation loss decreased (3.494256 --> 3.469489).  Saving model ...
Epoch: 33         Training Loss: 3.537295         Validation Loss: 3.418218
Validation loss decreased (3.469489 --> 3.418218).  Saving model ...
Epoch: 34         Training Loss: 3.496524         Validation Loss: 3.396425
Validation loss decreased (3.418218 --> 3.396425).  Saving model ...
Epoch: 35         Training Loss: 3.483696         Validation Loss: 3.374457
Validation loss decreased (3.396425 --> 3.374457).  Saving model ...
Epoch: 36         Training Loss: 3.454241         Validation Loss: 3.444821
Epoch: 37         Training Loss: 3.456127         Validation Loss: 3.392918
Epoch: 38         Training Loss: 3.438924         Validation Loss: 3.367291
Validation loss decreased (3.374457 --> 3.367291).  Saving model ...
Epoch: 39         Training Loss: 3.429539         Validation Loss: 3.451349
Epoch: 40         Training Loss: 3.396802         Validation Loss: 3.403779
Epoch: 41         Training Loss: 3.396842         Validation Loss: 3.374386
Epoch: 42         Training Loss: 3.357901         Validation Loss: 3.377760
Epoch: 43         Training Loss: 3.368677         Validation Loss: 3.320759
Validation loss decreased (3.367291 --> 3.320759).  Saving model ...
Epoch: 44         Training Loss: 3.323177         Validation Loss: 3.384522
Epoch: 45         Training Loss: 3.335377         Validation Loss: 3.354990
Epoch: 46         Training Loss: 3.293592         Validation Loss: 3.312866
Validation loss decreased (3.320759 --> 3.312866).  Saving model ...
Epoch: 47         Training Loss: 3.320755         Validation Loss: 3.327278
Epoch: 48         Training Loss: 3.265420         Validation Loss: 3.348955
Epoch: 49         Training Loss: 3.290519         Validation Loss: 3.309814
Validation loss decreased (3.312866 --> 3.309814).  Saving model ...
Epoch: 50         Training Loss: 3.267837         Validation Loss: 3.313844
Epoch: 51         Training Loss: 3.255474         Validation Loss: 3.300098
Validation loss decreased (3.309814 --> 3.300098).  Saving model ...
```

```
Epoch: 52         Training Loss: 3.267746         Validation Loss: 3.275481
Validation loss decreased (3.300098 --> 3.275481).  Saving model ...
Epoch: 53         Training Loss: 3.233187         Validation Loss: 3.319953
Epoch: 54         Training Loss: 3.239301         Validation Loss: 3.319187
Epoch: 55         Training Loss: 3.211895         Validation Loss: 3.305513
Epoch: 56         Training Loss: 3.238685         Validation Loss: 3.290401
Epoch: 57         Training Loss: 3.197057         Validation Loss: 3.283764
Epoch: 58         Training Loss: 3.217199         Validation Loss: 3.288142
Epoch: 59         Training Loss: 3.184631         Validation Loss: 3.211124
Validation loss decreased (3.275481 --> 3.211124).  Saving model ...
Epoch: 60         Training Loss: 3.179991         Validation Loss: 3.255598
Epoch: 61         Training Loss: 3.166562         Validation Loss: 3.306285
Epoch: 62         Training Loss: 3.180026         Validation Loss: 3.235815
Epoch: 63         Training Loss: 3.161516         Validation Loss: 3.252468
Epoch: 64         Training Loss: 3.168768         Validation Loss: 3.289305
Epoch: 65         Training Loss: 3.155468         Validation Loss: 3.287292
Epoch: 66         Training Loss: 3.144165         Validation Loss: 3.256140
Epoch: 67         Training Loss: 3.140753         Validation Loss: 3.233050
Epoch: 68         Training Loss: 3.096769         Validation Loss: 3.235964
Epoch: 69         Training Loss: 3.113732         Validation Loss: 3.242048
Epoch: 70         Training Loss: 3.117565         Validation Loss: 3.252852
Epoch: 71         Training Loss: 3.078007         Validation Loss: 3.233078
Epoch: 72         Training Loss: 3.087185         Validation Loss: 3.288702
Epoch: 73         Training Loss: 3.105389         Validation Loss: 3.244031
Epoch: 74         Training Loss: 3.106323         Validation Loss: 3.259827
Epoch: 75         Training Loss: 3.101550         Validation Loss: 3.213428
Epoch: 76         Training Loss: 3.064592         Validation Loss: 3.269095
Epoch: 77         Training Loss: 3.050073         Validation Loss: 3.268479
Epoch: 78         Training Loss: 3.069689         Validation Loss: 3.222138
Epoch: 79         Training Loss: 3.029058         Validation Loss: 3.225187
Epoch: 80         Training Loss: 2.992042         Validation Loss: 3.313363
Epoch: 81         Training Loss: 2.993890         Validation Loss: 3.209863
Validation loss decreased (3.211124 --> 3.209863).  Saving model ...
Epoch: 82         Training Loss: 3.022712         Validation Loss: 3.207238
Validation loss decreased (3.209863 --> 3.207238).  Saving model ...
Epoch: 83         Training Loss: 3.007371         Validation Loss: 3.250382
Epoch: 84         Training Loss: 2.987728         Validation Loss: 3.271427
Epoch: 85         Training Loss: 3.021452         Validation Loss: 3.241181
Epoch: 86         Training Loss: 3.011028         Validation Loss: 3.299006
Epoch: 87         Training Loss: 2.991635         Validation Loss: 3.266591
Epoch: 88         Training Loss: 2.991585         Validation Loss: 3.236610
Epoch: 89         Training Loss: 2.954184         Validation Loss: 3.261233
Epoch: 90         Training Loss: 2.960198         Validation Loss: 3.177333
Validation loss decreased (3.207238 --> 3.177333).  Saving model ...
Epoch: 91         Training Loss: 2.958988         Validation Loss: 3.253648
Epoch: 92         Training Loss: 2.911733         Validation Loss: 3.327003
Epoch: 93         Training Loss: 2.924597         Validation Loss: 3.276435
Epoch: 94         Training Loss: 2.940905         Validation Loss: 3.218979
```

```
Epoch: 95         Training Loss: 2.947257         Validation Loss: 3.225422
Epoch: 96         Training Loss: 2.952301         Validation Loss: 3.240171
Epoch: 97         Training Loss: 2.918084         Validation Loss: 3.254660
Epoch: 98         Training Loss: 2.904867         Validation Loss: 3.208831
Epoch: 99         Training Loss: 2.927141         Validation Loss: 3.254721
Epoch: 100         Training Loss: 2.942124         Validation Loss: 3.153031
Validation loss decreased (3.177333 --> 3.153031).  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 [30]: 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: 3.125665
```

```
Test Accuracy: 25% (213/836)
```

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 [74]:
```
## TODO: Specify data loaders
tl_train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                          transforms.RandomResizedCrop(224),
                                          transforms.RandomHorizontalFlip(),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406],
                                                               [0.229, 0.224, 0.225])])

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

tl_train_data = datasets.ImageFolder('/data/dog_images/train', transform=tl_train_trans
tl_test_data = datasets.ImageFolder('/data/dog_images/test', transform=tl_test_valid_tr
tl_valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=tl_test_valid_

data_transfer = {
    "train" : tl_train_data,
    "valid" : tl_test_data,
    "test" : tl_valid_data
}

tl_trainloader = torch.utils.data.DataLoader(tl_train_data, batch_size=64, shuffle=True
tl_testloader = torch.utils.data.DataLoader(tl_test_data, batch_size=64)
tl_validloader = torch.utils.data.DataLoader(tl_valid_data, batch_size=64)

loaders_transfer = {
    "train": tl_trainloader,
    "test": tl_testloader,
```

```
    "valid": tl_validloader
}
```

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 [81]: import torchvision.models as models
         import torch.nn as nn

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

         # out features
         num_dog_classes = 133

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

         # change the vgg16's final layer
         second_to_the_last_out_features = model_transfer.classifier[3].out_features
         last_layer = nn.Linear(in_features=second_to_the_last_out_features,
                                out_features=num_dog_classes, bias=True)
         model_transfer.classifier[6] = last_layer

         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:

As you can see vgg16 has reasonable result when we use pretrained model in recognizing dog photos on par with resnet50's model. By using transfer learning with only the last layer changed (changing the final out features) and with enough training epochs, we can expect similar results from pretrained model. I did not change anything on the classifier part since I believe that vgg16 is as good as it is for our purpose of classifying dog breed images, judging by the fact that it received novel top k errors when trained on ImageNet dataset.

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [83]: # train the model
         n_epochs = 5
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

         # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1         Training Loss: 3.472136         Validation Loss: 1.408361
Validation loss decreased (inf --> 1.408361).  Saving model ...
Epoch: 2         Training Loss: 2.737491         Validation Loss: 1.236918
Validation loss decreased (1.408361 --> 1.236918).  Saving model ...
Epoch: 3         Training Loss: 2.669960         Validation Loss: 1.190699
Validation loss decreased (1.236918 --> 1.190699).  Saving model ...
Epoch: 4         Training Loss: 2.690644         Validation Loss: 1.133079
Validation loss decreased (1.190699 --> 1.133079).  Saving model ...
Epoch: 5         Training Loss: 2.574999         Validation Loss: 1.148018
```

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

```
Test Loss: 1.159124


Test Accuracy: 63% (533/836)
```

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 [107]: ### 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 data_transfer['train'].classes]

          def predict_breed_transfer(img_path):
              # load the image and return the predicted breed
```

```
    img = Image.open(img_path)

    img = tl_test_valid_transforms(img)

    # to add batch dimensioning as per the following URL
    # https://discuss.pytorch.org/t/expected-stride-to-be-a-single-integer-value-or-a-
    img = img.unsqueeze_(0)

    if use_cuda:
        img = img.cuda()

    # using the model_transfer we implemented before
    predictions = model_transfer(img)

    # since predictions are of type Variable when subscripted, it cannot be directly
    # converted to numpy that is why we need to detach
    if use_cuda:
        predictions = predictions[0].detach().to('cpu').numpy()
    else:
        predictions = predictions[0].detach().numpy()

    # find the index as well as the value of the highest prediction
    # using operator
    # https://stackoverflow.com/questions/6193498/pythonic-way-to-find-maximum-value-a
    index, value = max(enumerate(predictions), key=operator.itemgetter(1))

    return class_names[index]
```

Step 5: Write your Algorithm

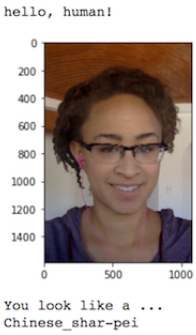
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 [149]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def is_vowel(ch):
```



Sample Human Output

```
    ch = ch.lower()
    if(ch=='A' or ch=='a' or ch=='E' or ch =='e' or ch=='I'
     or ch=='i' or ch=='O' or ch=='o' or ch=='U' or ch=='u'):
        return True
    return False

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    prediction = predict_breed_transfer(img_path)
    a_or_an = "an" if is_vowel(prediction[0]) else "a"
    img = mpimg.imread(img_path)

    if(face_detector_2(img_path)):
        plt.title("Hello Human\nYou look like " + a_or_an + " " + prediction)
    else:
        plt.title("Hello Dog\nYou look like " + a_or_an + " " + prediction)
    plt.imshow(img)
```

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)

1. As you pointed out, this works only on dog breeds and filtering if the image inputted is human or not. If the image is outside that scope, the algorithm would simply go to the else case, in this case, image being a dog and predict wrongly. I suggest that we augment other model strategies that would encompass other classes.

2. Previously, you see that the dlib got a great job to filter out dog images to be human images due to the presence of human in each of those pictures. The algorithm for detecting human is overly simplistic, just see whether there is AT LEAST one person in the picture and return True if that is, which is really problematic. I would suggest to devise an algorithm that would detect what is the major focus of the image (i.e. is it majorly dog image or human image), then predict based on that threshold.
3. Another thing, it is important for the model to be fed by data that really represents the class it represents. With dog images with humans in them, this proves that data is not filtered well. Also, observing the number of images per class, it is significantly skewed. To earn the best model accuracy (or any other metric), I would suggest to collect quality images with no biases in them and equal sample sizes per class.

In [155]:
```
## 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
human_file_names = ['human_1.jpg', 'human_2.jpg', 'human_3.jpg']
dog_file_names = ['dog_1.jpg', 'dog_2.jpg', 'dog_3.jpg']

fig = plt.figure(figsize=(24, 10))
for idx, name in enumerate(human_file_names + dog_file_names):
    ax = fig.add_subplot(2, 6/2, idx+1, xticks=[], yticks=[])
    run_app(name)
```