dog_app

June 25, 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.

2 Setup

2.1 Useful imports

In [1]:
```
# PyTorch related
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.models as models
import torchvision.transforms as vtransforms
import torchvision.datasets as datasets
import torch.optim as optim
import torch.nn.init as init

import numpy as np
import math

# Image processing
import cv2
from PIL import ImageFile
from PIL import Image

# Python standard lib
import random
import os
from glob import glob

# Progress bar
from tqdm import tqdm
from tqdm import tqdm_notebook


# DataViz
import matplotlib.pyplot as plt
%matplotlib inline
```

2.2 Global parameters

```
In [2]: BATCH_SIZE = 64
        N_EPOCHS = 30
        EARLY_STOPPING = 5

        # ImageNet normalization stats
        IMAGENET_STATS = {"mean":[0.485, 0.456, 0.406],
                          "std":[0.229, 0.224, 0.225]}

        # To handle correpted images
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        # check if CUDA is available
        # use_cuda = torch.cuda.is_available()
        device = "cuda" if torch.cuda.is_available() else "cpu"
```

```
In [3]: # 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.
```

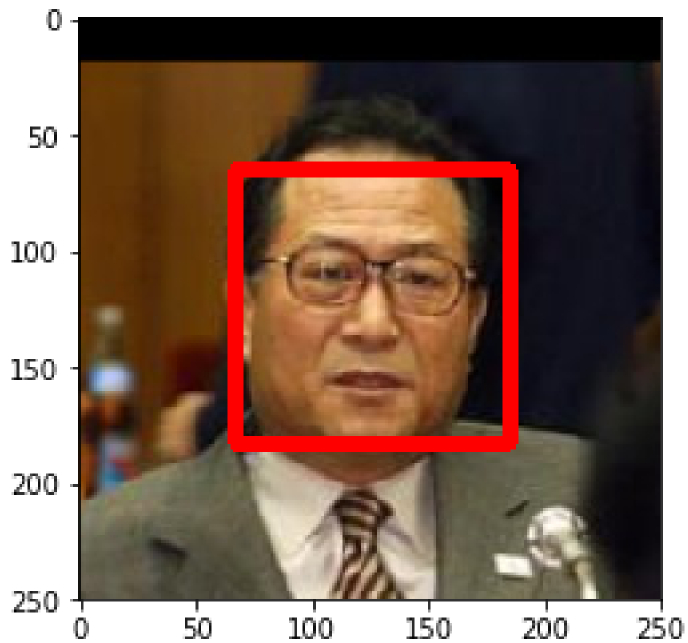
Step 1: Detect Humans
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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]: # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[13])
        # 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),(0,0,255),5)

# 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.

2.2.1 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
```

2.2.2 (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.

        # Human Face Detection rate on human images
        human_acc = 0
        # Human Face Detection rate on dog images
        dog_acc = 0

        for i in tqdm(range(100)):

            human_acc += int(face_detector(human_files_short[i]))
            dog_acc += int(face_detector(dog_files_short[i]))

        print("Human images:\tDetection rate:\t{:.1f}%".format(human_acc))
        print("Dog images:\tDetection rate:\t{:.1f}%".format(dog_acc))
```

```
100%|| 100/100 [00:43<00:00,  6.26it/s]
```

```
Human images:        Detection rate:        98.0%
Dog images:        Detection rate:        17.0%
```

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

```
In [7]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Check these data sets https://lionbridge.ai/datasets/5-million-faces-top-15-free-image-datasets-for-facial-recognition/

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

2.2.3 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 [8]: # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

        # move model to GPU if CUDA is available
        VGG16 = VGG16.to(device)
```

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.

2.2.4 (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 [9]:
```
def VGG16_predict(img_path, verbose=False):
    '''
    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)

    if verbose:
        print("Image infos {}, {}, {}".format(img.format, img.size, img.mode))

    img = img.convert(mode="RGB")

    transforms = vtransforms.Compose([vtransforms.Resize(256)
                                      ,vtransforms.CenterCrop(224)
                                      ,vtransforms.ToTensor()
                                      ,vtransforms.Normalize(IMAGENET_STATS["mean"],IMAGE
                                      ])

    img = transforms(img).unsqueeze(0)

    img = img.to(device)
    scores = VGG16(img)

    # predicted class index
    return scores.argmax(dim=1).item()
```

In [10]:
```
images = ['images/American_water_spaniel_00648.jpg',
          'images/Labrador_retriever_06455.jpg',
          'images/Welsh_springer_spaniel_08203.jpg',
          'images/Labrador_retriever_06449.jpg',
          'images/Labrador_retriever_06457.jpg',
          'images/Curly-coated_retriever_03896.jpg',
          'images/Brittany_02625.jpg']

sample_path = random.sample(images, 1)[0]
class_idx = VGG16_predict(sample_path, verbose=True)
```

```
        print("The predicted class for image {} is {}".format(sample_path, class_idx))
```

```
Image infos JPEG, (450, 664), RGB
The predicted class for image images/Brittany_02625.jpg is 215
```

2.2.5 (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 [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):

             cls_idx = VGG16_predict(img_path)

             if (cls_idx >= 151) and (cls_idx <= 268):
                 return True
             else:
                 return False
```

2.2.6 (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: (see cell output)

Note

- The `face_detector` is empirically reliable up to 98% on human images !
- The `dog_detector` is empirically reliable up to 99% on dog images !

```
In [12]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         # dog Detection rate on human images
         human_acc = 0

         # Dog Detection rate on dog images
         dog_acc = 0

         for i in tqdm(range(100)):
```

```
            human_acc += int(dog_detector(human_files_short[i]))
            dog_acc += int(dog_detector(dog_files_short[i]))

        print("Human images:\tDetection rate:\t{:.1f}%".format(human_acc))
        print("Dog images:\tDetection rate:\t{:.1f}%".format(dog_acc))
```

```
100%|| 100/100 [00:07<00:00, 14.20it/s]

Human images:        Detection rate:        2.0%
Dog images:        Detection rate:        100.0%
```

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

```
In [13]: ### (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!

2.2.7 (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 [14]: ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes

         try:
             assert set(os.listdir("/data/dog_images/train")) == set(os.listdir("/data/dog_image
         except AssertionError:
             print("Target across train and valid is not consistant")

         try:
             assert set(os.listdir("/data/dog_images/train")) == set(os.listdir("/data/dog_image
         except AssertionError:
             print("Target across train and test is not consistant")

         scratch_transforms_train = vtransforms.Compose([vtransforms.Resize(256)
                                                  ,vtransforms.CenterCrop(224)
                                                  ,vtransforms.RandomHorizontalFlip(p=0.2)
                                                  ,vtransforms.RandomRotation(30)
                                                  ,vtransforms.ToTensor()
                                                  ,vtransforms.Normalize([0.5,0.5,0.5],[0.5,0.5
                                                 ])

         scratch_transforms_test = vtransforms.Compose([vtransforms.Resize(256)
                                                  ,vtransforms.CenterCrop(224)
                                                  ,vtransforms.ToTensor()
                                                  ,vtransforms.Normalize([0.5,0.5,0.5],[0.5,0.5
                                                 ])

         data_sets = {}
         loaders_scratch = {}
```

```
for key in ("train", "valid", "test"):

    if key == "train":
        transforms = scratch_transforms_train
    else:
        transforms = scratch_transforms_test

    data_sets[key] = datasets.ImageFolder(f"/data/dog_images/{key}", transform=transfor
    loaders_scratch[key] = torch.utils.data.DataLoader(data_sets[key], batch_size=BATCH
```

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

Answer:

- My approach consists in stertching the image to 256 while preserving width/height ratio then center crop a 224 patch. I picked this size because PyTorch pretrained models expect at least 224 pixels inputs.
- I selected the following data augmentation only for training data:
  - Random Horizontal flip
  - Random Rotation

2.2.8 (IMPLEMENTATION) Model Architecture

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

```
In [15]: # Overloading Conv2D weight initialization
         class Conv2d(nn.Conv2d):
             def __init__(self, *args, **kwargs):

                 super(Conv2d, self).__init__(*args, **kwargs)

             def reset_parameters(self):

                 init.kaiming_uniform_(self.weight, a=math.sqrt(5))
                 if self.bias is not None:
                     fan_in, _ = init._calculate_fan_in_and_fan_out(self.weight)
                     bound = 1 / math.sqrt(fan_in)
                     init.uniform_(self.bias, -bound, bound)

         # 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.features = nn.Sequential(Conv2d(3,16,3,padding=1)
                                      ,nn.ReLU()
                                      ,nn.MaxPool2d(2,2) # (16,112,112)
                                      ,Conv2d(16,32,3,padding=1)
                                      ,nn.ReLU()
                                      ,nn.MaxPool2d(2,2) # (32, 56,56)
                                      ,Conv2d(32,64,3, padding=1)
                                      ,nn.ReLU()
                                      ,nn.MaxPool2d(2,2) # (64, 28,28)
                                      ,Conv2d(64,128,3, padding=1)
                                      ,nn.ReLU()
                                      ,nn.MaxPool2d(4,4) # (128, 7,7)
                                     )
        self.classifier = nn.Sequential(nn.Dropout(0.2)
                                        ,nn.Linear(128*7*7, 1024)
                                        ,nn.Dropout(0.2)
                                        ,nn.Linear(1024, 133)
                                       )


    def forward(self, x):
        ## Define forward behavior
        x = self.features(x)
        x = x.view(x.shape[0], -1)
        x = self.classifier(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
model_scratch = model_scratch.to(device)
```

In [16]:
```
print(model_scratch)
```

```
Net(
  (features): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU()
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU()
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
    (9): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU()
    (11): MaxPool2d(kernel_size=4, stride=4, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Dropout(p=0.2)
    (1): Linear(in_features=6272, out_features=1024, bias=True)
    (2): Dropout(p=0.2)
    (3): Linear(in_features=1024, out_features=133, bias=True)
  )
)
```

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

Answer:

I started with this architecture: - Features extractor - CONV(3,16,3) - ReLU - MaxPooling(2,2) - CONV(16,32,3) - ReLU - MaxPooling(2,2) - CONV(32,64,3) - ReLU - MaxPooling(2,2) - CONV(64,128,3) - ReLU - MaxPooling(2,2)

- Classifier
  - Dropout(0.2)
  - FC (25088, 8192)
  - Dropout(0.2)
  - FC (8192, 133)

But, the network hardly train at all. I suspected that the problem is with initialization and simplified it a bit by using MaxPooling layer of kernel 4 before the classifier input. So, I used He initialization that is the deault in PyTorch 1.5+ but not in PyTorch 0.4.0, the version on Udacity workspace.

- Features extractor
  - CONV(3,16,3) - ReLU - MaxPooling(2,2)
  - CONV(16,32,3) - ReLU - MaxPooling(2,2)
  - CONV(32,64,3) - ReLU - MaxPooling(2,2)
  - CONV(64,128,3) - ReLU - MaxPooling(4,4)
- Classifier
  - Dropout(0.2)
  - FC (6272, 1024)
  - Dropout(0.2)
  - FC (1024, 133)

2.2.9 (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 [17]: ### TODO: select loss function
         criterion_scratch = nn.CrossEntropyLoss()

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

2.2.10 (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, save_path):
             """returns trained model"""

             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

             # Counter: # of elapsed epochs since last validation loss decrease
             epoch_counter = 0

             for epoch in tqdm_notebook(range(1, n_epochs+1)):

                 # initialize variables to monitor training and validation loss
                 train_loss = 0.0
                 valid_loss = 0.0
                 if epoch_counter >= EARLY_STOPPING:
                     print('Validatiion loss did not improve for {} epochs, Stopping training'.f
                     break

                 epoch_counter += 1


                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     data, target = data.to(device), target.to(device)
                     ## find the loss and update the model parameters accordingly

                     optimizer.zero_grad()
                     scores = model(data)
                     loss = criterion(scores, target)
                     loss.backward()
                     optimizer.step()
```

```
            ## record the avaerage training loss, using something like
            train_loss += loss.item()

        ######################
        # validate the model #
        ######################
        model.eval()
        correct = 0
        total = 0
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            data, target = data.to(device), target.to(device)
            ## update the average validation loss
            with torch.no_grad():
                scores = model(data)
                loss = criterion(scores, target)
                valid_loss += loss.item()

                pred = scores.data.max(1, keepdim=True)[1]
                # compare predictions to true label
                correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().
                total += data.size(0)

        train_loss = train_loss/len(loaders["train"])
        valid_loss = valid_loss/len(loaders["valid"])

        # print training/validation statistics
        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tAccuracy:{
            epoch,
            train_loss,
            valid_loss,
            100*correct/total
            ))

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            print("Validation decreased: {:.6f} --> {:.6f}\t[SAVING MODEL]".format(vali
            valid_loss_min = valid_loss
            torch.save(model.state_dict(),save_path)
            epoch_counter = 0

    # return trained model
    return model
```

```
In [31]: # train the model
         model_scratch = train(N_EPOCHS, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, 'model_scratch.pt')
```

```
HBox(children=(IntProgress(value=0, max=30), HTML(value='')))
```

```
Epoch: 1         Training Loss: 4.884129        Validation Loss: 4.881108         Accuracy:1.08
Validation decreased: inf --> 4.881108        [SAVING MODEL]
Epoch: 2         Training Loss: 4.718988        Validation Loss: 4.526438         Accuracy:3.59
Validation decreased: 4.881108 --> 4.526438        [SAVING MODEL]
Epoch: 3         Training Loss: 4.333918        Validation Loss: 4.251754         Accuracy:5.63
Validation decreased: 4.526438 --> 4.251754        [SAVING MODEL]
Epoch: 4         Training Loss: 4.081512        Validation Loss: 4.016020         Accuracy:7.78
Validation decreased: 4.251754 --> 4.016020        [SAVING MODEL]
Epoch: 5         Training Loss: 3.934852        Validation Loss: 4.036582         Accuracy:9.82
Epoch: 6         Training Loss: 3.765568        Validation Loss: 3.757045         Accuracy:9.46
Validation decreased: 4.016020 --> 3.757045        [SAVING MODEL]
Epoch: 7         Training Loss: 3.609126        Validation Loss: 3.841989         Accuracy:12.7
Epoch: 8         Training Loss: 3.432876        Validation Loss: 3.694008         Accuracy:13.1
Validation decreased: 3.757045 --> 3.694008        [SAVING MODEL]
Epoch: 9         Training Loss: 3.306803        Validation Loss: 3.747915         Accuracy:14.9
Epoch: 10         Training Loss: 3.185503        Validation Loss: 3.699211         Accuracy:15.
Epoch: 11         Training Loss: 3.032308        Validation Loss: 3.600392         Accuracy:15.
Validation decreased: 3.694008 --> 3.600392        [SAVING MODEL]
Epoch: 12         Training Loss: 2.932475        Validation Loss: 3.737147         Accuracy:15.
Epoch: 13         Training Loss: 2.807554        Validation Loss: 3.551400         Accuracy:16.
Validation decreased: 3.600392 --> 3.551400        [SAVING MODEL]
Epoch: 14         Training Loss: 2.678659        Validation Loss: 3.771740         Accuracy:17.
Epoch: 15         Training Loss: 2.572263        Validation Loss: 3.837673         Accuracy:16.
Epoch: 16         Training Loss: 2.470902        Validation Loss: 3.549975         Accuracy:17.
Validation decreased: 3.551400 --> 3.549975        [SAVING MODEL]
Epoch: 17         Training Loss: 2.400993        Validation Loss: 3.661535         Accuracy:17.
Epoch: 18         Training Loss: 2.285723        Validation Loss: 3.730840         Accuracy:17.
Epoch: 19         Training Loss: 2.225770        Validation Loss: 3.787888         Accuracy:18.
Epoch: 20         Training Loss: 2.103046        Validation Loss: 3.632658         Accuracy:16.
Epoch: 21         Training Loss: 2.040761        Validation Loss: 4.141875         Accuracy:16.
Validatiion loss did not improve for 4 epochs, Stopping training
```

In [19]:
```python
# load the model that got the best validation accuracy
if device == "cpu":
    model_scratch.load_state_dict(torch.load('model_scratch.pt', map_location=device))
else:
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

2.2.11 (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 [20]:
```python
def test(loaders, model, criterion):


    # 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 appropriate device
            data, target = data.to(device), target.to(device)
            # 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 += loss.item()
            # 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/len(loaders["test"])))

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

In [21]:
```
# call test function
test(loaders_scratch, model_scratch, criterion_scratch)
```

```
Test Loss: 3.740494


Test Accuracy: 18% (151/836)
```

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.

2.2.12 (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 [51]: ## TODO: Specify data loaders
         ### TODO: Write data loaders for training, validation, and test sets
         ## Specify appropriate transforms, and batch_sizes
         BATCH_SIZE = 32

         transfer_transforms_train = vtransforms.Compose([vtransforms.Resize(256)
                                                  ,vtransforms.CenterCrop(224)
                                                  ,vtransforms.RandomHorizontalFlip(p=0.2)
                                                  ,vtransforms.RandomRotation(30)
                                                  ,vtransforms.ToTensor()
                                                 ,vtransforms.Normalize(IMAGENET_STATS["mean"],
                                                 ])

         transfer_transforms_test = vtransforms.Compose([vtransforms.Resize(256)
                                                  ,vtransforms.CenterCrop(224)
                                                  ,vtransforms.ToTensor()
                                                 ,vtransforms.Normalize(IMAGENET_STATS["mean"],
                                                 ])

         data_sets = {}
         loaders_transfer = {}

         for key in ("train", "valid", "test"):

             if key == "train":
                 transforms = transfer_transforms_train
             else:
                 transforms = transfer_transforms_test

             data_sets[key] = datasets.ImageFolder(f"/data/dog_images/{key}", transform=transfor
             loaders_transfer[key] = torch.utils.data.DataLoader(data_sets[key], batch_size=BATC
```

2.2.13 (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 [52]: ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)

         for p in model_transfer.parameters():
             p.requires_grad_ = False

         n_input = model_transfer.fc.in_features
         n_output = 133
         model_transfer.fc = nn.Linear(n_input,n_output)
         model_transfer = model_transfer.to(device)
```

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:

The Dog breed dataset is a subset of ImageNet. Hence, using a state of the art CNN on ImageNet will likely perform well on this very similar task (133 classes instead of 1000 classes) with very few changes.

My approach is first freeze all pretrained network weights because the data set is similar. Then to replace the last classifier part (Fully connected layer) with a new one with an appropriate number of outputs and train it from scratch.

2.2.14 (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 [53]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = torch.optim.Adam(model_transfer.fc.parameters(), lr = 0.001)
```

2.2.15 (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 [54]: # train the model
         N_EPOCHS = 10
         model_transfer = train(N_EPOCHS, loaders_transfer, model_transfer, optimizer_transfer,
```

```
HBox(children=(IntProgress(value=0, max=10), HTML(value='')))


Epoch: 1         Training Loss: 2.341288        Validation Loss: 0.956635         Accuracy:74.1
Validation decreased: inf --> 0.956635        [SAVING MODEL]
Epoch: 2         Training Loss: 0.904459        Validation Loss: 0.706213         Accuracy:80.1
Validation decreased: 0.956635 --> 0.706213        [SAVING MODEL]
Epoch: 3         Training Loss: 0.686605        Validation Loss: 0.605883         Accuracy:82.6
Validation decreased: 0.706213 --> 0.605883        [SAVING MODEL]
Epoch: 4         Training Loss: 0.583509        Validation Loss: 0.597497         Accuracy:81.9
Validation decreased: 0.605883 --> 0.597497        [SAVING MODEL]
Epoch: 5         Training Loss: 0.525998        Validation Loss: 0.616336         Accuracy:80.8
Epoch: 6         Training Loss: 0.485266        Validation Loss: 0.553858         Accuracy:81.7
Validation decreased: 0.597497 --> 0.553858        [SAVING MODEL]
Epoch: 7         Training Loss: 0.429861        Validation Loss: 0.532566         Accuracy:81.6
Validation decreased: 0.553858 --> 0.532566        [SAVING MODEL]
Epoch: 8         Training Loss: 0.392698        Validation Loss: 0.548948         Accuracy:81.4
Epoch: 9         Training Loss: 0.383288        Validation Loss: 0.509355         Accuracy:83.7
Validation decreased: 0.532566 --> 0.509355        [SAVING MODEL]
Epoch: 10         Training Loss: 0.364495        Validation Loss: 0.580979         Accuracy:82.
```

```
In [55]: # load the model that got the best validation accuracy (uncomment the line below)
         if device == "cpu":
             model_transfer.load_state_dict(torch.load('model_transfer.pt', map_location=device)
         else:
             model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

2.2.16 (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%.

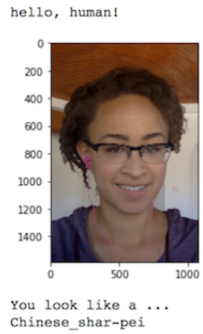
```
In [56]: test(loaders_transfer, model_transfer, criterion_transfer)
```

```
Test Loss: 0.551788


Test Accuracy: 83% (694/836)
```

2.2.17 (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 [57]: ### 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_sets['train'].classes]

         def predict_breed_transfer(img_path, verbose=False):
             # load the image and return the predicted breed

             img = Image.open(img_path)
             if verbose:
                 print("Image infos {}, {}, {}".format(img.format, img.size, img.mode))

             img = img.convert(mode="RGB")

             transforms = vtransforms.Compose([vtransforms.Resize(256)
                                              ,vtransforms.CenterCrop(224)
                                              ,vtransforms.ToTensor()
                                              ,vtransforms.Normalize(IMAGENET_STATS["mean"],IMAG
                                              ])

             img = transforms(img).unsqueeze(0)
             # Move to appropriate device
             img = img.to(device)
             scores = model_transfer(img)
```



Sample Human Output

```
        # predicted class index
        idx = scores.argmax(dim=1).item()
        return class_names[idx]
```

In [58]:
```
predict_breed_transfer("images/Labrador_retriever_06457.jpg")
```

Out[58]: 'Labrador retriever'

Step 5: Write your Algorithm
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

2.2.18 (IMPLEMENTATION) Write your Algorithm

In [59]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):

    is_human = face_detector(img_path)
    is_dog = dog_detector(img_path)

    if is_dog or is_human:

        dog_breed = predict_breed_transfer(img_path)
        if is_dog:
```

```
            output = f"There is a dog in the image, it is most likely a {dog_breed}"
        else:
            output = f"Hello human :) You resemble a lot to a {dog_breed}"
    else:
        output = "Error"

    return output
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

2.2.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

The output is really good and classify all dog images correctly. This single RenNet 50 achieves 83% accuracy which is good but not excellent.

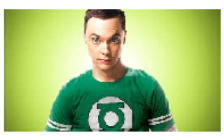
Next steps for improvements are : - Improve face detection algorithm by using deep learning based methods. - Improve dog breed classifier, by adding more layers in the classifier part - Ensembling several neural nets.

```
In [65]: ## 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.
         SIZE = 224
         input_files = glob("app_inputs/*")
         fig, axes = plt.subplots(len(input_files),2, sharex=True, sharey=True, figsize=(10,30))
         ## suggested code, below
         for i, img_path in enumerate(input_files):

             ax1 = axes[i,0]
             ax2 = axes[i,1]
             message = run_app(img_path)
             img =  Image.open(img_path)
             w,h = img.size
             img = img.resize(size=(SIZE,int(SIZE*h/w)))
             ax1.imshow(img)
             ax1.set_axis_off();
             ax2.text(0.5, 0.5, message, ha='left', rotation=0, fontsize=15)
             ax2.set_axis_off();
```

There is a dog in the image, it is most likely a French bulldog

There is a dog in the image, it is most likely a Pharaoh hound

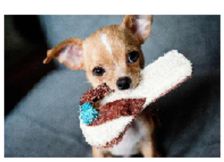
There is a dog in the image, it is most likely a Alaskan malamute



Hello human :) You resemble a lot to a Dogue de bordeaux

Hello human :) You resemble a lot to a Silky terrier

There is a dog in the image, it is most likely a Beagle



There is a dog in the image, it is most likely a Chihuahua

Hello human :) You resemble a lot to a Pomeranian

In [69]:
```
# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# load color (BGR) image
img = cv2.imread("app_inputs/img_8.jpeg")
# 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),(0,0,255),5)

# 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
```

In [ ]: