dog_app

December 28, 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

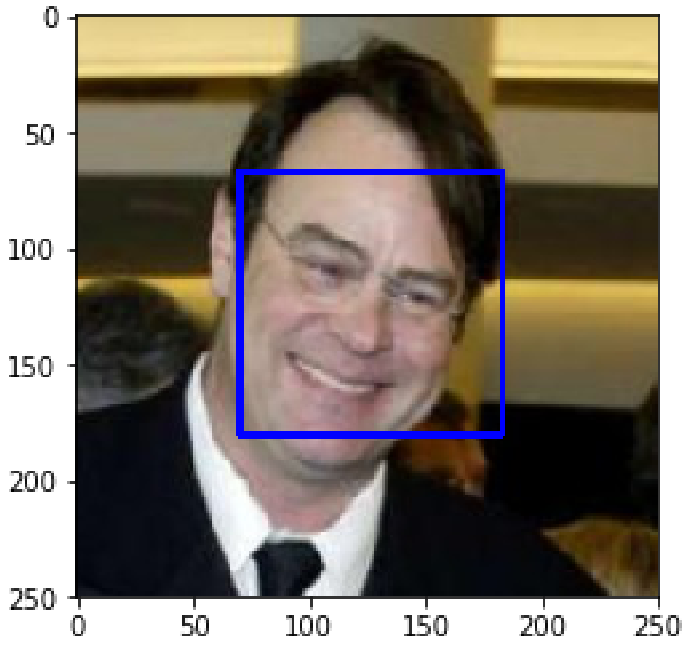
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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.
        count_human = 0
        count_dog = 0
        for data in human_files_short:
            if face_detector(data):
                count_human += 1
        for data in dog_files_short:
            if face_detector(data):
                count_dog += 1
        print("Percentage of the first 100 images in human_files have a detected human face:{} %
        print("Percentage of the first 100 images in dog_files have a detected human face:{} %".
```

```
Percentage of the first 100 images in human_files have a detected human face:98 %
Percentage of the first 100 images in dog_files have a detected human face: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 anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models

        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

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

        # move model to GPU if CUDA is available
        if use_cuda:
            VGG16 = VGG16.cuda()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 97129581.02it/s]
```

```
In [7]: print(VGG16)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```
In [8]: print(VGG16.classifier[6].in_features)
        print(VGG16.classifier[6].out_features)
```

```
4096
1000
```

```
In [9]: #freez training for all features layers to get weight update
        for param in VGG16.features.parameters():
            param.require_grad = False
```

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

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

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

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

```
In [10]: from PIL import Image
         import torchvision.transforms as transforms

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

             Args:
                 img_path: path to an image

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

             ## 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

             transform = transforms.Compose([
                 transforms.RandomResizedCrop(224),
                 transforms.RandomRotation(45),
                 transforms.ToTensor()])

             image = Image.open(img_path)
             image = transform(image)
             image.unsqueeze_(0)
             image = image.cuda()
             output = VGG16(image)
             _, index = torch.max(output, 1)
             return index.item() # predicted class index
```

```
In [11]: VGG16_predict(dog_files[0])
```

```
Out[11]: 669
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

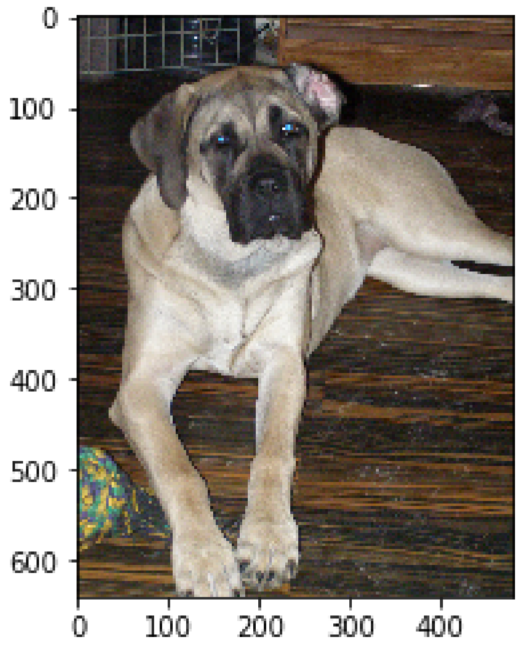
While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted 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 [12]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             index = VGG16_predict(img_path)
             if ((index >= 151) and (index <= 268)):
                 return True
             else:
                 return False
             # true/false
```

```
In [13]: image = Image.open(dog_files_short[17])
         plt.imshow(image)
         print(dog_detector(dog_files_short[17]))
```

```
True
```



1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: Percentage of the images in human_files have a detected dog:0 % Percentage of the images in dog_files have a detected dog:43 %

```
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_file = 0
         dog_in_dog_file = 0

         for data in human_files_short:
             if dog_detector(data):
                 dog_in_human_file += 1
         for data in dog_files_short:
             if dog_detector(data):
                 dog_in_dog_file += 1
         print("Percentage of the images in human_files have a detected dog:{} %".format(dog_in_
         print("Percentage of the images in dog_files have a detected dog:{} %".format(dog_in_do
```

```
Percentage of the images in human_files have a detected dog:0 %
Percentage of the images in dog_files have a detected dog:43 %
```

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

## Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [1]: import torch
        import os
        from torchvision import datasets, transforms
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

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

        image_transforms = {
            'train': transforms.Compose([
                transforms.Resize(size = 256),
                transforms.RandomHorizontalFlip(),
                transforms.RandomRotation(15),
                transforms.CenterCrop(size = 224),
                transforms.ToTensor(),
                transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
            ]),
            'valid': transforms.Compose([
                transforms.Resize(size= 256),
                transforms.CenterCrop(size = 224),
                transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
        ]),
        'test': transforms.Compose([
            transforms.Resize(size = 256),
            transforms.CenterCrop(size = 224),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
        ])
    }

    data_dir = 'images_dogs/'
    train_dataset = datasets.ImageFolder(os.path.join('/data/dog_images/train'), transform =
    valid_dataset = datasets.ImageFolder(os.path.join('/data/dog_images/valid'), transform =
    test_dataset = datasets.ImageFolder(os.path.join('/data/dog_images/test'), transform = i

    train_loader = torch.utils.data.DataLoader(train_dataset, shuffle = True, batch_size = 5
    valid_loader = torch.utils.data.DataLoader(valid_dataset, shuffle = True, batch_size = 5
    test_loader = torch.utils.data.DataLoader(test_dataset, shuffle = False, batch_size = 50
```

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:

Image Resize I choosed VGG16 to do the task. And I found that input tensors for VGG 16 is 224 x 224. So, I center cropped it to 224 x 224 first resizing it to 256 x 256 as mentioned in the original paper.

Data Augmentation Yes, I decide to augment the dataset by Random Rotation, random horizontal flip because it gives more exploring features to the model and will train better also saves from being overfitting.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [2]: 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(in_channels = 3, out_channels = 64, kernel_size = 3, padd
                self.conv2 = nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 3, pa
                self.conv3 = nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 3, p
                self.conv4 = nn.Conv2d(in_channels = 256, out_channels = 512, kernel_size = 3, p
            self.pool = nn.MaxPool2d(2, 2)
            self.fc1 = nn.Linear(14 * 14 * 512, 512)
            self.fc2 = nn.Linear(512, 256)
            self.fc3 = nn.Linear(256, 133)
            self.dropout = nn.Dropout(0.2)

        def forward(self, x):
            ## Define forward behavior
            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = self.pool(F.relu(self.conv3(x)))
            x = self.pool(F.relu(self.conv4(x)))

            x = x.view(-1, 14 * 14 * 512)
            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()

    use_cuda = torch.cuda.is_available()

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

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

Answer:

It was very difficult to get accuracy more than 10% on my own from scratch. So, I decided to follow the technique used in VGG16 model. **Input Size**: 224 x 224 **Kernel Size**: 3 x 3 **Padding**: It is 1 for 3x3 kernel, to keep the same spatial resolution **Max Pooling**: 2 x 2 with stride of 1 pixels, to reduce the size of image and the amount of parameters in half and to capture the most useful pixels(computation reduced!) **Activation Function**: Relu, quick to evaluate. **Convulational Layers**: Best layers for classification

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 [3]: import torch.optim as optim
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        ### 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 [4]: 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_los

                    # clear gradients
                    optimizer.zero_grad()

                    # forward passing
                    output = model(data)

                    # calculate loss
                    loss = criterion(output, target)

                    # backward passing
                    loss.backward()
            # perform optimization
            optimizer.step()

            #training loss
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

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


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

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

    # return trained model
    return model

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

# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch, criterion_s

# load the model that got the best validation accuracy
```

```
        model_scratch.load_state_dict(torch.load('model_scratch.pt'))

Epoch: 1         Training Loss: 4.858454         Validation Loss: 4.755160
Validation loss decreased inf ---> 4.755160. Saving Model...
Epoch: 2         Training Loss: 4.623870         Validation Loss: 4.535931
Validation loss decreased 4.755160 ---> 4.535931. Saving Model...
Epoch: 3         Training Loss: 4.395303         Validation Loss: 4.298185
Validation loss decreased 4.535931 ---> 4.298185. Saving Model...
Epoch: 4         Training Loss: 4.178487         Validation Loss: 4.157563
Validation loss decreased 4.298185 ---> 4.157563. Saving Model...
Epoch: 5         Training Loss: 4.031825         Validation Loss: 4.006633
Validation loss decreased 4.157563 ---> 4.006633. Saving Model...
Epoch: 6         Training Loss: 3.915295         Validation Loss: 3.966209
Validation loss decreased 4.006633 ---> 3.966209. Saving Model...
Epoch: 7         Training Loss: 3.797299         Validation Loss: 3.940233
Validation loss decreased 3.966209 ---> 3.940233. Saving Model...
Epoch: 8         Training Loss: 3.658667         Validation Loss: 3.828481
Validation loss decreased 3.940233 ---> 3.828481. Saving Model...
Epoch: 9         Training Loss: 3.551067         Validation Loss: 3.752818
Validation loss decreased 3.828481 ---> 3.752818. Saving Model...
Epoch: 10         Training Loss: 3.442825         Validation Loss: 3.755939
Epoch: 11         Training Loss: 3.323548         Validation Loss: 3.707700
Validation loss decreased 3.752818 ---> 3.707700. Saving Model...
Epoch: 12         Training Loss: 3.219716         Validation Loss: 3.708094
Epoch: 13         Training Loss: 3.140075         Validation Loss: 3.699464
Validation loss decreased 3.707700 ---> 3.699464. Saving Model...
Epoch: 14         Training Loss: 3.000829         Validation Loss: 3.757372
Epoch: 15         Training Loss: 2.920816         Validation Loss: 3.842961
Epoch: 16         Training Loss: 2.808656         Validation Loss: 3.783407
Epoch: 17         Training Loss: 2.700765         Validation Loss: 3.734869
Epoch: 18         Training Loss: 2.637755         Validation Loss: 3.771258
Epoch: 19         Training Loss: 2.501596         Validation Loss: 3.851956
Epoch: 20         Training Loss: 2.429060         Validation Loss: 3.802113
```

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 [5]: 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: 3.658973


Test Accuracy: 16% (137/836)
```

## 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 [3]: ## TODO: Specify data loaders
        import torch
        import os
        from torchvision import datasets, transforms
        import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

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

image_transforms = {
    'train': transforms.Compose([
        transforms.Resize(size = 256),
        transforms.RandomHorizontalFlip(),
        transforms.RandomRotation(15),
        transforms.CenterCrop(size = 224),
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ]),
    'valid': transforms.Compose([
        transforms.Resize(size= 256),
        transforms.CenterCrop(size = 224),
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ]),
    'test': transforms.Compose([
        transforms.Resize(size = 256),
        transforms.CenterCrop(size = 224),
        transforms.ToTensor(),
        transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
    ])
}

data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')

train_dataset = datasets.ImageFolder(train_dir, transform = image_transforms['train'])
valid_dataset = datasets.ImageFolder(valid_dir, transform = image_transforms['valid'])
test_dataset = datasets.ImageFolder(test_dir, transform = image_transforms['test'])

print(len(train_dataset))
print(len(valid_dataset))
print(len(test_dataset))

train_loader = torch.utils.data.DataLoader(train_dataset, shuffle = True, batch_size = 5
valid_loader = torch.utils.data.DataLoader(valid_dataset, shuffle = True, batch_size = 5
test_loader = torch.utils.data.DataLoader(test_dataset, shuffle = False, batch_size = 50

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

```
6680
835
836
```

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

        ## TODO: Specify model architecture
        use_cuda = torch.cuda.is_available()

        model_transfer = models.vgg16(pretrained = True)

        # freezing features layers
        for param in model_transfer.features.parameters():
            param.require_grad = False

        dog_labels = train_dataset.classes
        number_dog_labels = len(dog_labels)
        print(number_dog_labels)

        n_inputs = model_transfer.classifier[6].in_features
        n_outputs = 133

        model_transfer.classifier[6] = nn.Linear(n_inputs, n_outputs)
        print(model_transfer)

        # classifier for dog breed
        #classifier = nn.Sequential(nn.Linear)
        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
133
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

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

Answer:

I chose the VGG16 model. I've already imitated it from scratch at the step 3. But I wondered the differences between my model and the original VGG16 model. And I also wondered how the accuracy will go from the pre-trained model, because I had limited computing power, dataset and time to train.

I thought the VGG16 is sutable for the current problem. Because it already trained large dataset. So I initialized randomly the wieght in the new fully connected layer, and the rest of the weights using the pre-trained weights. And overfitting is not as much of a concern when training on a large data set. And the model classifies like my problem needs.

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 [8]: criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = torch.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 [14]: # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1         Training Loss: 3.459293         Validation Loss: 1.805282
Validation loss decreased inf ---> 1.805282. Saving Model...
Epoch: 2         Training Loss: 2.517801         Validation Loss: 1.429558
Validation loss decreased 1.805282 ---> 1.429558. Saving Model...
Epoch: 3         Training Loss: 2.392995         Validation Loss: 1.468986
Epoch: 4         Training Loss: 2.270045         Validation Loss: 1.294340
Validation loss decreased 1.429558 ---> 1.294340. Saving Model...
Epoch: 5         Training Loss: 2.294731         Validation Loss: 1.274323
Validation loss decreased 1.294340 ---> 1.274323. Saving Model...
Epoch: 6         Training Loss: 2.136108         Validation Loss: 1.221868
Validation loss decreased 1.274323 ---> 1.221868. Saving Model...
Epoch: 7         Training Loss: 2.079087         Validation Loss: 1.223456
Epoch: 8         Training Loss: 2.065769         Validation Loss: 1.245247
Epoch: 9         Training Loss: 1.958494         Validation Loss: 1.234113
Epoch: 10         Training Loss: 2.062045         Validation Loss: 1.327363
Epoch: 11         Training Loss: 1.986166         Validation Loss: 1.113037
Validation loss decreased 1.221868 ---> 1.113037. Saving Model...
Epoch: 12         Training Loss: 1.979807         Validation Loss: 1.209898
Epoch: 13         Training Loss: 1.874383         Validation Loss: 1.084164
Validation loss decreased 1.113037 ---> 1.084164. Saving Model...
Epoch: 14         Training Loss: 1.882697         Validation Loss: 1.097248
Epoch: 15         Training Loss: 2.046471         Validation Loss: 1.237062
Epoch: 16         Training Loss: 1.898645         Validation Loss: 1.268141
Epoch: 17         Training Loss: 1.884741         Validation Loss: 1.193882
Epoch: 18         Training Loss: 1.847325         Validation Loss: 1.185208
Epoch: 19         Training Loss: 1.878242         Validation Loss: 1.191212
Epoch: 20         Training Loss: 1.756747         Validation Loss: 1.189799
```

```
In [9]: # load the model that got the best validation accuracy (uncomment the line below)
        model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [15]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.188708


Test Accuracy: 66% (560/836)
```

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.

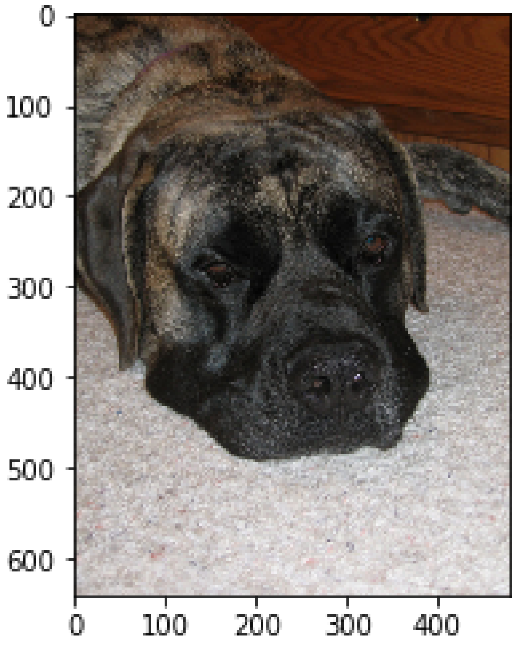
```
In [10]: from PIL import Image
         ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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

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

             #loading image
             image = Image.open(img_path)

             #preparing image
             transform = transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
             ])

             image = transform(image)
             image.unsqueeze_(0)
             if use_cuda:
                 image = image.cuda()
             output = model_transfer(image)
             softmax = nn.Softmax(dim=1)
             predictions = softmax(output)
```

```
            top_predictions = torch.topk(predictions, 2)
            probability = top_predictions[0][0][0]
            class_index = top_predictions[1][0][0]

            return top_predictions, probability, class_index
```

```
In [15]: top_predictions , probability, class_index = predict_breed_transfer(dog_files[10])
         print("Predicted Dog is: {} with probability: {:.3f}".format(class_names[class_index],
         dog = Image.open(dog_files[10])
         plt.imshow(dog)
```

```
Predicted Dog is: Belgian malinois with probability: 0.459
```

```
Out[15]: <matplotlib.image.AxesImage at 0x7f46a6d31e80>
```



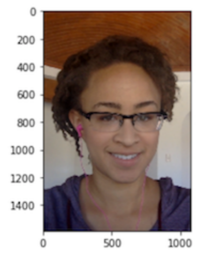
## Step 5: Write your Algorithm

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!

```
hello, human!
```



```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [18]: 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/*/*/*"))

         def show_image(image_path):
             img = Image.open(image_path)
             _, ax = plt.subplots()
             ax.imshow(img)
             plt.axis('off')
             plt.show()

         ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             top_predictions , probability, class_index = predict_breed_transfer(img_path)
             if probability > 0.4:
                 print("Hello, Dog!!")
                 show_image(img_path)
                 print("Predicted Dog is: {} with probability: {:.3f} %\n".format(class_names[cl

             elif face_detector(img_path):
                 print("Hello, Human")
                 show_image(img_path)
                 print("You look like a: {} with probability: {:.3f} %\n".format(class_names[cla
             else:
                 print("You are neither human nor dog.\n")
                 show_image(image_path)
```

## 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.

```
In [21]: run_app('./IMG_20190220_194444_826.jpg')
```

```
Hello, Human
```

```
You look like a: Cane corso with probability: 6.208 %
```

Answer: (Three possible points for improvement)

1. Although I used the transfer learning but still accuracy is less. So can work on it.
2. Some wrong predictions.
3. Can make this system more interactive.

```
In [76]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below

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

```
Hello, Human
```

```
You look like a: Dachshund with probability: 0.017

Hello, Human
```

```
You look like a: Xoloitzcuintli with probability: 0.023

Hello, Human
```

```
You look like a: Chinese crested with probability: 0.017

Hello, Dog!!
```

```
Predicted Dog is: Bullmastiff with probability: 0.947

Hello, Dog!!
```

```
Predicted Dog is: Bullmastiff with probability: 0.582

Hello, Dog!!
```

```
Predicted Dog is: Mastiff with probability: 0.476
```

```
In [ ]:
```