dog_app

November 16, 2018

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: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- Download the human dataset. Unzip the folder and place it in the home diretcory, 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("lfw/*/*"))
        dog_files = np.array(glob("dogImages/*/*/*"))

        # 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.
```

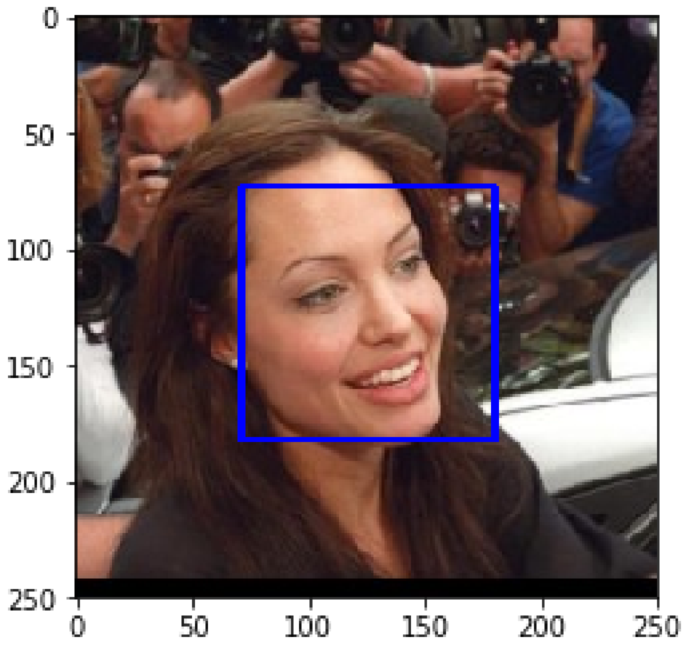
## Step 1: Detect Humans

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:
Percentage of the first 100 images in human_files have a detected human face = 99.0 %
Percentage of the first 100 images in dog_files have a detected human face = 6.0 %

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        num_human_faces_detected_in_human = 0
        num_human_faces_detected_in_dogs = 0

        for i in range(len(human_files_short)):
            if(face_detector(human_files_short[i])):
                num_human_faces_detected_in_human += 1
            if(face_detector(dog_files_short[i])):
                num_human_faces_detected_in_dogs +=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 = 99.0 %
Percentage of the first 100 images in dog_files have a detected human face = 6.0 %
```

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`. OpenCV does a pretty good job of face detection with a 99%, so we are going to use OpenCV for the face detection task.

```
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)
        VGG16.eval()

        # 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()
```

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 [7]: from PIL import Image
        from PIL import ImageFile
        import torchvision.transforms as transforms

        ImageFile.LOAD_TRUNCATED_IMAGES = True

        def VGG16_predict(img_path):
```

```
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

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

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

    image = Image.open(img_path)

    img_transforms = transforms.Compose([ #transforms.RandomResizedCrop(224),
                                          transforms.Resize((224,224)),
                                          transforms.ToTensor(),
                                          transforms.Normalize(mean=[0.485, 0.456, 0.4
                                                        std=[0.229, 0.224, 0.225])])
    img_transformed = img_transforms(image)

    if use_cuda:
        img_transformed = img_transformed.cuda()

    output = VGG16(img_transformed.view(1,3,224,224))
    _, preds_tensor = torch.max(output, 1)

    return preds_tensor.item() # predicted class index
```

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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            dog_indices = list(range(151,269))
            return VGG16_predict(img_path) in dog_indices # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:
Percentage of the first 100 images in human_files have a detected dog = 1.0 %
Percentage of the first 100 images in dog_files have a detected dog = 100.0 %

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.
        num_dogs_detected_in_human = 0
        num_dogs_detected_in_dogs = 0

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

        print("Percentage of the first 100 images in human_files have a detected dog = "+str(nu
        print("Percentage of the first 100 images in dog_files have a detected dog = "+str(num_
```

```
Percentage of the first 100 images in human_files have a detected dog = 1.0 %
Percentage of the first 100 images in dog_files have a detected dog = 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 [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

```
In [11]: # create a function the takes in an image and a model and return the predicted class
         def model_predict(img_path, model,input_size):
             '''
             Use pre-trained 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 model's prediction
             '''
```

```
        ## TODO: Complete the function.
        ## Load and pre-process an image from the given img_path
        ## Return the *index* of the predicted class for that image

        image = Image.open(img_path)

        img_transforms = transforms.Compose([ transforms.Resize((input_size,input_size)),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.
                                                          std=[0.229, 0.224, 0.225])])
        img_transformed = img_transforms(image)
        if use_cuda:
            img_transformed = img_transformed.cuda()

        output = model(img_transformed.view(1,3,input_size,input_size))
        _, preds_tensor = torch.max(output, 1)

        return preds_tensor.item() # predicted class index
```

```
In [12]: ### returns "True" if a dog is detected in the image stored at img_path for model
         def dog_detector_for_model(img_path,model,input_size=224):
             ## TODO: Complete the function.
             dog_indices = list(range(151,269))
             return model_predict(img_path,model,input_size) in dog_indices # true/false
```

```
In [13]: # define resnet-18 model
         resnet50 = models.resnet50(pretrained=True)
         resnet50.eval()

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

         num_dogs_detected_in_human = 0
         num_dogs_detected_in_dogs = 0

         for i in range(len(human_files_short)):
             if(dog_detector_for_model(human_files_short[i],resnet50)):
                 num_dogs_detected_in_human += 1
             if(dog_detector_for_model(dog_files_short[i],resnet50)):
                 num_dogs_detected_in_dogs +=1

         print("Percentage of the first 100 images in human_files have a detected dog RESNET50
         print("Percentage of the first 100 images in dog_files have a detected dog RESNET50 =
```

```
Percentage of the first 100 images in human_files have a detected dog RESNET50 = 1.0 %
Percentage of the first 100 images in dog_files have a detected dog RESNET50 = 100.0 %
```

```
In [14]: # define inception_v3 model
         inception = models.inception_v3(pretrained=True)
         inception.eval()

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

         num_dogs_detected_in_human = 0
         num_dogs_detected_in_dogs = 0

         for i in range(len(human_files_short)):
             if(dog_detector_for_model(human_files_short[i],inception,299)):
                 num_dogs_detected_in_human += 1
             if(dog_detector_for_model(dog_files_short[i],inception,299)):
                 num_dogs_detected_in_dogs +=1

         print("Percentage of the first 100 images in human_files have a detected dog INCEPTIO
         print("Percentage of the first 100 images in dog_files have a detected dog INCEPTION =
```

```
Percentage of the first 100 images in human_files have a detected dog INCEPTION = 4.0 %
Percentage of the first 100 images in dog_files have a detected dog INCEPTION = 100.0 %
```

```
In [15]: 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)
  )
)
```

## 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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [16]: import os
         import torchvision
         from torchvision import datasets

         batch_size = 32
         num_workers = 5
         input_image_size = (128,128)

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

         test_valid_transforms = transforms.Compose([transforms.Resize(input_image_size),
                                                     transforms.ToTensor(),
                                                     transforms.Normalize(mean=[0.485, 0.456,
                                                                std=[0.229, 0.224, 0.225])])

         train_data = torchvision.datasets.ImageFolder('dogImages/train', transform=train_trans
         valid_data = torchvision.datasets.ImageFolder('dogImages/valid', transform=test_valid
         test_data = torchvision.datasets.ImageFolder('dogImages/test', transform=test_valid_t

         train_data_loader = torch.utils.data.DataLoader(train_data,
                                                                  batch_size=batch_size,
                                                                  shuffle=True,
                                                                  num_workers=num_workers)


        valid_data_loader = torch.utils.data.DataLoader(valid_data,
                                                                  num_workers=num_workers)

        test_data_loader = torch.utils.data.DataLoader(test_data,
                                                                  num_workers=num_workers)

        # define loaders dictionary
        loaders_scratch={'train':train_data_loader,
                         'valid': valid_data_loader,
                         'test': test_data_loader}
```

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

Answer: - I used the PyTorch transforms to resize images to 128x128. I tried different architectures for the model, and the model with input images with that size seemed to do pretty well on the dog breed classification task

- Yes, augmentation definitely helped improve the model's performance. I augmented the data by using the random horizontal flip transform as well as the random rotation.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [17]: import torch.nn as nn
         import torch.nn.functional as F

         input_image_depth = 3

         # 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
                 # sequential
                 self.conv1 = nn.Conv2d(input_image_depth,32,kernel_size=3,padding=1)
                 self.conv11 = nn.Conv2d(32,64,kernel_size=3,padding=1)
                 self.conv2 = nn.Conv2d(64,128,kernel_size=3,padding=1)
                 self.conv21 = nn.Conv2d(128,256,kernel_size=3,padding=1)
                 self.conv3 = nn.Conv2d(256,512,kernel_size=3,padding=1)
                 self.mp = nn.MaxPool2d(2,2)
```

```
            # classifier
            self.fc1 = nn.Linear(16*16*512,1024)
            self.fc2 = nn.Linear(1024,512)
            self.fc3 = nn.Linear(512,133)
            self.dropout = nn.Dropout(0.5)
        def forward(self, x):
            x = F.relu(self.conv1(x))
            x = F.relu(self.conv11(x))
            x = self.mp(x)
            x = F.relu(self.conv2(x))
            x = F.relu(self.conv21(x))
            x = self.mp(x)
            x = F.relu(self.conv3(x))
            x = self.mp(x)

            # classifier
            x = x.view(-1,16*16*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()

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

    print(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv11): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv21): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=131072, out_features=1024, bias=True)
  (fc2): Linear(in_features=1024, out_features=512, bias=True)
  (fc3): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.5)
)
```

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

Answer: I tried a bunch of architectures with a sequence of convolutional and pooling layers, followed by a sequence of fully connected layers. The activation function for all layers except the last layer was relu, and for the last layer the sigmoid activation function was used because we are dealing with a classification task. For the sequence of convolutional layers, convolution was used to increase the depth of the layers while maintaining the spatial dimension, and the maxpooling layers were used to downsize the spatial dimensions, while maintaing the same depth as the input. For regularziaion, I used dropout for the fully connected portion of the model to avoid overfitting. Below is the structure of the model, as well as the sizes of the layers:

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv11): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv21): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(conv3): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(mp): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
(fc1): Linear(in_features=131072, out_features=1024, bias=True)
(dropout): Dropout(p=0.5)
(fc2): Linear(in_features=1024, out_features=512, bias=True)
(dropout): Dropout(p=0.5)
(fc3): Linear(in_features=512, out_features=133, bias=True)

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 [18]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr = 0.0001)
```

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 [19]: 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
    # reset gradiants to zero
    optimizer.zero_grad()
    # get output from model
    output = model(data)
    # calculate loss
    loss = criterion_scratch(output, target)
    # calculate back_prop
    loss.backward()
    # update weights and biases
    optimizer.step()
    ## record the average training loss, using something like
    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_
    train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_los

######################
# validate the model #
######################
model.eval()
for batch_idx, (data, target) in enumerate(loaders['valid']):
    # move to GPU
    if use_cuda:
        data, target = data.cuda(), target.cuda()
    ## update the average validation loss
    output = model(data)
    # calculate loss
    loss = criterion_scratch(output, target)

    valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_los


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

```
                    ))

                ## TODO: save the model if validation loss has decreased
                if(valid_loss < valid_loss_min):
                    print('Saving model with validation loss = '+str(valid_loss.item()))
                    valid_loss_min = valid_loss
                    torch.save(model.state_dict(), save_path)
                    torch.save(model, 'model_transfer_full.pt')

            # return trained model
            return model
```

```
In [20]: # train the model
         # model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch, criter

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

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [21]: def test(loaders, model, criterion, use_cuda):

             # monitor test loss and accuracy
             test_loss = 0.
             correct = 0.
             total = 0.

             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))
```

```
In [25]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.799014


Test Accuracy: 16% (136/836)
```

## 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 [26]: ## TODO: Specify data loaders
         batch_size = 32
         num_workers = 5
         input_image_size = (224,224)

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

         test_valid_transforms = transforms.Compose([transforms.Resize(input_image_size),
                                                     transforms.ToTensor(),
                                                     transforms.Normalize(mean=[0.485, 0.456, 
                                                                std=[0.229, 0.224, 0.225])])

         train_data = torchvision.datasets.ImageFolder('dogImages/train', transform=train_trans
         valid_data = torchvision.datasets.ImageFolder('dogImages/valid', transform=test_valid_
         test_data = torchvision.datasets.ImageFolder('dogImages/test', transform=test_valid_t
train_data_loader_transfer = torch.utils.data.DataLoader(train_data,
                                                          batch_size=batch_size,
                                                          shuffle=True,
                                                          num_workers=num_workers)


valid_data_loader_transfer = torch.utils.data.DataLoader(valid_data,
                                                          num_workers=num_workers)

test_data_loader_transfer = torch.utils.data.DataLoader(test_data,
                                                          num_workers=num_workers)

# define loaders dictionary
loaders_transfer={'train':train_data_loader_transfer,
                'valid': valid_data_loader_transfer,
                'test': test_data_loader_transfer}
```

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

         ## TODO: Specify model architecture to be VGG19
         model_transfer = models.vgg19(pretrained=True)

         # freeze the feature layer parameters
         for parameters in model_transfer.parameters():
             parameters.requires_grad = False

         # print model classifier to check classifier dimensions
         print(model_transfer.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 [28]: # replace the final fully connected layer of the classifier with a fully connected la
         # corresponding to the 133 dog breeds
```

```
        model_transfer.classifier[6] = nn.Linear(4096, 133)

        # print new model_transfer to check the classifier is modified
        print(model_transfer.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: To make use of transfer learning, I chose the pre-trained VGG19 model. The model was trained to classify different types of objects, including dogs, and since our dataset is small and similar to the dataset that the model was trained on, it makes sense to freeze the convolutional layers' parameters, replace the final fully connected layer with a layer with the correct dimensions to classify our 133 dog breeds, and then retrain the fully connected-portion of the model.

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 [29]: ### TODO: select loss function
         criterion_transfer = nn.CrossEntropyLoss()

         ### TODO: select optimizer
         optimizer_transfer = optim.Adam(model_transfer.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 [ ]: # move model to GPU if available
        if use_cuda:
            model_transfer = model_transfer.cuda()

        # train the model using same loader as scratch
        # model_transfer = train(50, loaders_transfer, model_transfer, optimizer_transfer, cri
```

```
In [30]: # load the model that got the best validation accuracy (uncomment the line below)
         # model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         model_transfer = torch.load('model_transfer_full.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 [31]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

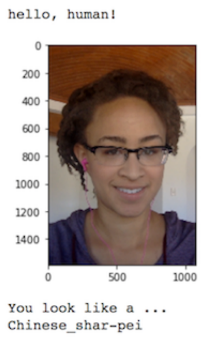
```
Test Loss: 0.639146


Test Accuracy: 80% (677/836)
```

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 [32]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

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

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

             image = Image.open(img_path)

             img_transforms = transforms.Compose([ #transforms.RandomResizedCrop(224),
                                                  transforms.Resize((224,224)),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean=[0.485, 0.456, 0.
                                                              std=[0.229, 0.224, 0.225])])
             img_transformed = img_transforms(image)

             if use_cuda:
                 img_transformed = img_transformed.cuda()

             output = model_transfer(img_transformed.view(1,3,224,224))
             _, preds_tensor = torch.max(output, 1)

             return class_names[preds_tensor.item()]
```

## 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



```
hello, human!
```

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

Sample Human Output

breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [33]: ### 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
             if(dog_detector(img_path)):
                 print("Not a human, most probably a dog of breed ...\n"+predict_breed_transfe
             elif(face_detector(img_path)):
                 print("Hello, human!, you like a ...\n"+predict_breed_transfer(img_path))
             else:
                 print("Neither a face nor a dog are deteced in the image")
```

## Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer:

The model did a pretty good job actually on classifying dog breeds with an accuracy of 80%.

Possible points of improvements to get better accuracy:

1- Adding more training data

2- Training the whole model rather than just training the fully connected portion of the model

3- Try a bigger model like ResNet150

```
In [34]: ## 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])):
             print(file)
             run_app(file)
```

```
lfw/Angelina_Jolie/Angelina_Jolie_0012.jpg
Hello, human!, you like a ...
Afghan hound
lfw/Angelina_Jolie/Angelina_Jolie_0010.jpg
Hello, human!, you like a ...
Japanese chin
lfw/Angelina_Jolie/Angelina_Jolie_0007.jpg
Hello, human!, you like a ...
Dachshund
dogImages/train/056.Dachshund/Dachshund_03969.jpg
Not a human, most probably a dog of breed ...
Dachshund
dogImages/train/056.Dachshund/Dachshund_03927.jpg
Not a human, most probably a dog of breed ...
Irish setter
dogImages/train/056.Dachshund/Dachshund_03939.jpg
Not a human, most probably a dog of breed ...
Dachshund
```