dog_app

May 3, 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: * 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
        # 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

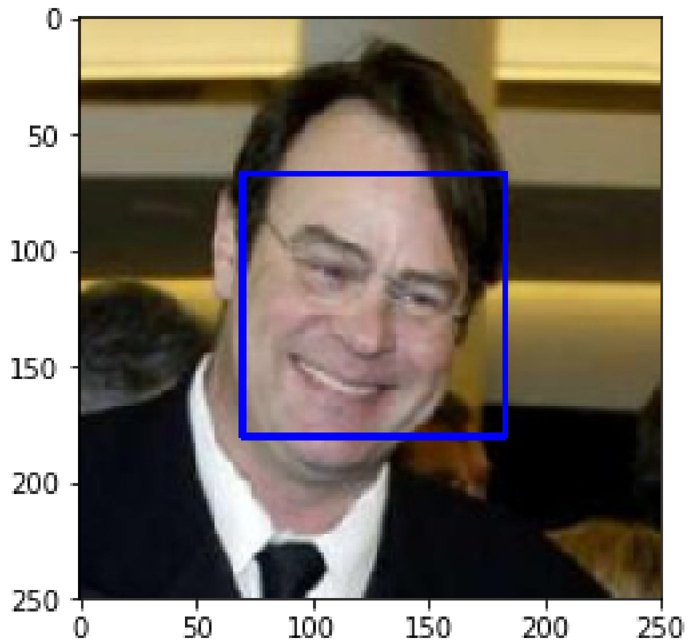
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [4]: # 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 [5]: from tqdm import tqdm

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        faces_vfunc = np.vectorize(face_detector)

        # Detect faces in both sets
        human_faces = faces_vfunc(human_files_short)
        dog_faces = faces_vfunc(dog_files_short)

        # Calculate and print percentage of faces in each set
        print('Faces detected in {:.2f}% of the sample human dataset.'.format((sum(human_faces)/
        print('Faces detected in {:.2f}% of the sample dog dataset.'.format((sum(dog_faces)/len(
```

```
Faces detected in 98.00% of the sample human dataset.
Faces detected in 17.00% of the sample dog dataset.
```

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 [6]: ## (Optional) TODO: Report the performance of another
        ## face detection algorithm on the LFW dataset
        ### Feel free to use as many code cells as needed.
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

In [7]:
```
#import torch
#import torchvision.models as models
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()
use_cuda = torch.cuda.is_available()
# move model to GPU if CUDA is available
if use_cuda:
    VGG16 = VGG16.cuda()
    print('CUDA is available!  Training on GPU ...')
else:
    print('CUDA is not available.  Training on CPU ...')
```

```
CUDA is available!  Training on GPU ...
```

In [8]:
```
VGG16
```

Out[8]:
```
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)
  )
)
```

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 [9]:
```
from PIL import Image
import torchvision.transforms as transforms

def VGG16_predict(img_path):

    '''
```

```python
    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


    # Import image from img_path in PIL format
    img = Image.open(img_path)

    # Define normalization step for image
    normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                     std=(0.229, 0.224, 0.225))

    # Define transformations of image
    preprocess = transforms.Compose([transforms.Resize(256),
                                     transforms.CenterCrop(224),
                                     transforms.ToTensor(),
                                     normalize])

    # Preprocess image to 4D Tensor (.unsqueeze(0) adds a dimension)
    img_tensor = preprocess(img).unsqueeze_(0)

    # Move tensor to GPU if available
    if use_cuda:
        img_tensor = img_tensor.cuda()

    ## Inference
    # Turn on evaluation mode
    VGG16.eval()

    # Get predicted category for image
    with torch.no_grad():
        output = VGG16(img_tensor)
        prediction = torch.argmax(output).item()

    # Turn off evaluation mode
    VGG16.train()

    return prediction # predicted class index
```

```
In [10]: dog_image = Image.open('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jp
         plt.imshow(dog_image)
         plt.show()
```

1.1.5 (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?

```
In [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             prediction = VGG16_predict(img_path)
             return True if 151 <= prediction <= 268 else False # true/false
```

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         from tqdm import tqdm_notebook
         # human_files_short
         dogs_in_human_files_VGG16 = 0

         for file in tqdm_notebook(human_files_short, desc='human_files'):
             if dog_detector(file):
                 dogs_in_human_files_VGG16 += 1
```

```
# dog_files_short
dogs_in_dog_files_VGG16 = 0

for file in tqdm_notebook(dog_files_short, desc='dog_files'):
    if dog_detector(file):
        dogs_in_dog_files_VGG16 += 1

print('#### VGG16 ####')
print(f'Dogs detected in "human_files_short": {dogs_in_human_files_VGG16 / len(human_fi
print(f'Dogs detected in "dog_files_short": {dogs_in_dog_files_VGG16 / len(dog_files_sh
```

```
HBox(children=(IntProgress(value=0, description='human_files: '), HTML(value='')))
```

```
HBox(children=(IntProgress(value=0, description='dog_files: '), HTML(value='')))
```

```
#### VGG16 ####
Dogs detected in "human_files_short": 0.0%
Dogs detected in "dog_files_short": 100.0%
```

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

Answer:

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`.

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!

1.1.6 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [14]:
```
import os
import torchvision.transforms as transforms
from torchvision import datasets

num_workers = 0
batch_size = 10

data_dir = '/data/dog_images'
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes


data_transforms = {
    'train' : transforms.Compose([
    transforms.Resize(256),
    transforms.RandomResizedCrop(224),
```

```
    transforms.RandomHorizontalFlip(), # randomly flip and rotate
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),

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

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


train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

image_datasets = {
    'train' : datasets.ImageFolder(root=train_dir,transform=data_transforms['train']),
    'valid' : datasets.ImageFolder(root=valid_dir,transform=data_transforms['valid']),
    'test' : datasets.ImageFolder(root=test_dir,transform=data_transforms['test'])
}

# Loading Dataset
loaders_scratch = {
    'train' : torch.utils.data.DataLoader(image_datasets['train'],batch_size = batch_si
    'valid' : torch.utils.data.DataLoader(image_datasets['valid'],batch_size = batch_si
    'test' : torch.utils.data.DataLoader(image_datasets['test'],batch_size = batch_size
}



#loaders_scratch = {"train" : train_loader, "test" : test_loader}



print(loaders_scratch['train'])
#print(len(train_data))
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7faef8764128>
```

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 resizing

I decided to follow the original VGG16 paper (Simonyan K, Zisserman A 2015), where the authors chose a 224x224 px image as input tensor, randomly cropped from a rescaled version of the orignal image. It is not entirely clear to my why the authors chose these exact numbers, but I read somewhere online that it has something to do with the fact, that after 5 maxpool layers we end up with a 7x7 image which has a center point. The rescaling of the orignial image is necessary, because cropping a 224x224 image out of a much larger orignal is unlikely to contain the features we are interested in. Thus following the Simonyan et al. paper I rescaled the original images to 256x256 px before cropping.

Data Augmentation

I chose to augment the image data by random rotation up to 10 degrees and by random horizontal flipping. Data augmentation is an easy way to extend a dataset and improve generalization when training the model.

1.1.7 (IMPLEMENTATION) Model Architecture

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

```
In [24]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 16, 3)
                 self.conv2 = nn.Conv2d(16, 32, 3)
                 self.conv3 = nn.Conv2d(32, 64, 3)
                 self.conv4 = nn.Conv2d(64, 128, 3)
                 self.conv5 = nn.Conv2d(128, 256, 3)

                 self.fc1 = nn.Linear(256 * 6 * 6, 133)
                 self.max_pool = nn.MaxPool2d(2, 2,ceil_mode=True)

                 self.dropout = nn.Dropout(0.20)

                 self.conv_bn1 = nn.BatchNorm2d(224,3)
                 self.conv_bn2 = nn.BatchNorm2d(16)
```

```
        self.conv_bn3 = nn.BatchNorm2d(32)
        self.conv_bn4 = nn.BatchNorm2d(64)
        self.conv_bn5 = nn.BatchNorm2d(128)
        self.conv_bn6 = nn.BatchNorm2d(256)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.max_pool(x)
        x = self.conv_bn2(x)

        x = F.relu(self.conv2(x))
        x = self.max_pool(x)
        x = self.conv_bn3(x)

        x = F.relu(self.conv3(x))
        x = self.max_pool(x)
        x = self.conv_bn4(x)

        x = F.relu(self.conv4(x))
        x = self.max_pool(x)
        x = self.conv_bn5(x)

        x = F.relu(self.conv5(x))
        x = self.max_pool(x)
        x = self.conv_bn6(x)

        x = x.view(-1, 256 * 6 * 6)

        x = self.dropout(x)
        x = self.fc1(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)
# move tensors to GPU if CUDA is available
if use_cuda:
    model_scratch.cuda()
```

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
```

```
  (fc1): Linear(in_features=9216, out_features=133, bias=True)
  (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (dropout): Dropout(p=0.2)
  (conv_bn1): BatchNorm2d(224, eps=3, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer:

I used the 5 convolution layers all with the colvolution of kernel size = 3, stride = 1 and padding = 0 Relu activations are used after each convoltution layers except the last one. Max pooling layers of 2Œ2 are applied. Batch normalization is applied after each max pool layer. Dropout is applied with the probability of 0.2. First layer will take three inputs for RGB because the in_channel is 3 and produces 16 output, the next layer to be a convolutional layer with 16 filters. Input = 224x224 RGB image Kernel Size = 3x3 Padding = 1 for 3x3 kernel MaxPooling = 2x2 with stride of 2 pixels, which will reduce the size of image and by the result the number of parameters will be half. Activation Function = ReLU (No vanishing gradient, there will be very vevry small output for input very larg or very small). Batch Normalization 2D is a technique to provide inputs that are zero mean or variance 1.

Layer 1: (3,16) input channels =3 , output channels = 16 Layer 2: (16,32) input channels = 16 , output channels = 32 Layer 3: (32,64) input channels =32 , output channels = 64 Layer 4: (64,128) input channels =64 , output channels = 128 Layer 5: (128,256) input channels =128 , output channels = 256 One fully connected layer with 9216 input channels and 133 output channel as dog breeds

1.1.8 (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 [25]: import torch.optim as optim
         import torch.nn as nn

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

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

1.1.9 (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 [20]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [21]:
```
import os
import numpy as np
from PIL import Image
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms, models
```

In [26]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            # train_loss += loss.item()*data.size(0)
            train_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
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


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

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

            ## TODO: save the model if validation loss has decreased
            ## save the model if validation loss has decreased
            if valid_loss <= valid_loss_min:
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
                valid_loss_min,
                valid_loss))
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss
        # return trained model
        return model


    # train the model
    model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch, criterion_

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

```
Epoch: 1        Training Loss: 0.000847         Validation Loss: 0.005714
Validation loss decreased (inf --> 0.005714).  Saving model ...
```

```
Epoch: 2 	Training Loss: 0.000702 	Validation Loss: 0.005242
Validation loss decreased (0.005714 --> 0.005242).  Saving model ...
Epoch: 3 	Training Loss: 0.000666 	Validation Loss: 0.005039
Validation loss decreased (0.005242 --> 0.005039).  Saving model ...
Epoch: 4 	Training Loss: 0.000648 	Validation Loss: 0.004844
Validation loss decreased (0.005039 --> 0.004844).  Saving model ...
Epoch: 5 	Training Loss: 0.000627 	Validation Loss: 0.004794
Validation loss decreased (0.004844 --> 0.004794).  Saving model ...
Epoch: 6 	Training Loss: 0.000615 	Validation Loss: 0.004690
Validation loss decreased (0.004794 --> 0.004690).  Saving model ...
Epoch: 7 	Training Loss: 0.000602 	Validation Loss: 0.004560
Validation loss decreased (0.004690 --> 0.004560).  Saving model ...
Epoch: 8 	Training Loss: 0.000593 	Validation Loss: 0.004404
Validation loss decreased (0.004560 --> 0.004404).  Saving model ...
Epoch: 9 	Training Loss: 0.000574 	Validation Loss: 0.004435
Epoch: 10 	Training Loss: 0.000568 	Validation Loss: 0.004345
Validation loss decreased (0.004404 --> 0.004345).  Saving model ...
```

1.1.10 (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 [27]: 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: 3.623679


Test Accuracy: 16% (139/836)
```

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.

1.1.11 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [ ]: ## TODO: Specify data loaders
        data_dir = '/data/dog_images'

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

        from torchvision import utils

        #data_transform = transforms.Compose([transforms.RandomResizedCrop(224),transforms.ToTen

        data_transforms = {
            'train' : transforms.Compose([
            transforms.Resize(256),
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(), # randomly flip and rotate
            transforms.RandomRotation(10),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
            ]),
```

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

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

train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])
valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['valid'])

print('Num training images: ',len(train_data))
print('Num test images: ',len(test_data))
print('Num validaion images: ',len(valid_data))

batch_size = 20
num_workers = 0
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_worker
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, num_worker

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

#Displaying Trainign data
############
def display_img(inp):
    inp = inp.numpy().transpose((1, 2, 0))
    inp = np.clip(inp, 0, 1)

    fig = plt.figure(figsize=(50, 25))
    plt.axis('off')
    plt.imshow(inp)
    plt.pause(0.001)
# Display!
dataiter = iter(test_loader)
images, labels = dataiter.next()
# Convert the batch to a grid.
grid = utils.make_grid(images[:5])
display_img(grid)
```

1.1.12 (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 [ ]: import torchvision.models as models
        import torch.nn as nn


        use_cuda = torch.cuda.is_available()

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


        for param in model_transfer.parameters():
            param.requires_grad = False

        model_transfer.fc = nn.Linear(2048, 133, bias=True)
        fc_parameters = model_transfer.fc.parameters()

        for param in fc_parameters:
            param.requires_grad = True

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

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

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

1.1.14 (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 [ ]: print(use_cuda)
        n_epochs = 15

        #model_transfer.train()

        def train(n_epochs, loader, model, optimizer, criterion, use_cuda, save_path):
            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_loss = 0.0
                model.train()
                for batch_i, (data, target) in enumerate(loader['train']):
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    optimizer.zero_grad()
                    output = model(data)
                    loss = criterion(output, target)
                    loss.backward()
                    optimizer.step()
                    train_loss = train_loss + ((1 / (batch_i + 1)) * (loss.data - train_loss))

                    if batch_i % 100 == 0:
                        print('Epoch %d, Batch %d loss: %.6f' %
                          (epoch, batch_i + 1, train_loss))

                model.eval()
                for batch_i, (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 = valid_loss + ((1 / (batch_i + 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 < valid_loss_min:
                torch.save(model.state_dict(), save_path)
                print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model'.format(
                valid_loss_min,
                valid_loss))
                valid_loss_min = valid_loss

        return model

    # train the model
    model_transfer = train(n_epochs, loaders, model_transfer, optimizer_transfer, criterion_

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

1.1.15 (IMPLEMENTATION) Test the Model

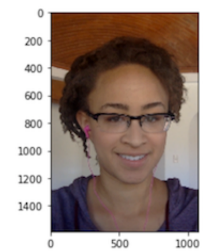
Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

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

1.1.16 (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 [ ]: ### 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 loaders['train'].dataset.classes]

        def predict_breed_transfer(img_path):
            # load the image and return the predicted breed
            image = Image.open(img_path).convert('RGB')
            prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.485, 0.456, 0.406], st

            # discard the transparent, alpha channel (that's the :3) and add the batch dimension
            image = prediction_transform(image)[:3,:,:].unsqueeze(0)
            image = image.cuda()



            model_transfer.eval()
```



Sample Human Output

```
        idx = torch.argmax(model_transfer(image))
        return class_names[idx]
```

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

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

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

1.1.17 (IMPLEMENTATION) Write your Algorithm

```
In [ ]: ### 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
            img = Image.open(img_path)
            plt.imshow(img)
            plt.show()
            if(face_detector(img_path)):
                print("Human Detected")
            elif(dog_detector(img_path)):
                print("Dog detected")
                prediction = predict_breed_transfer(img_path)
                print("Dog Breed: {0}".format(prediction))
            else:
                print("Neither dog not human detected")
```

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?

1.1.18 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

Hyper-parameter tuning and training will improve Large datasets (big data) will improve the performance of the model Random transformations: More rotating, flipping, cropping and then training with these transformations will improve

In [28]:
```
## 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
count = 0;
for file in np.hstack((human_files[:3], dog_files[:8])):
    print("Input:",count+1)
    run_app(file)
    count=count+1;
```

```
Input: 1


    ---------------------------------------------------------------------------

    NameError                                 Traceback (most recent call last)

    <ipython-input-28-b00182b3ba4c> in <module>()
      8 for file in np.hstack((human_files[:3], dog_files[:8])):
      9     print("Input:",count+1)
---> 10     run_app(file)
     11     count=count+1;


    NameError: name 'run_app' is not defined
```

In [ ]: