dog_app

June 7, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [0]: #from google.colab import drive
        #drive.mount('/content/drive')
```

```
In [0]: import os
        import shutil

        if not os.path.exists("/content/data/"):
          os.makedirs("/content/data/")
          if not os.path.exists("/content/data/dog_images") and not os.path.exists("/content/dat
            !wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip  #&&
            shutil.unpack_archive("/content/dogImages.zip", "/content/data")
            os.rename("/content/data/dogImages", "/content/data/dog_images")
            !rm /content/dogImages.zip
            !wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip #-o /conte
            shutil.unpack_archive("/content/lfw.zip", "/content/data")

            !rm /content/lfw.zip
```

```
--2020-06-07 08:20:58--  https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.z
Resolving s3-us-west-1.amazonaws.com (s3-us-west-1.amazonaws.com)... 52.219.112.80
Connecting to s3-us-west-1.amazonaws.com (s3-us-west-1.amazonaws.com)|52.219.112.80|:443... conn
HTTP request sent, awaiting response... 200 OK
Length: 1132023110 (1.1G) [application/zip]
Saving to: dogImages.zip

dogImages.zip       100%[===================>]   1.05G  21.1MB/s    in 53s

2020-06-07 08:21:51 (20.5 MB/s) - dogImages.zip saved [1132023110/1132023110]

--2020-06-07 08:22:04--  https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
Resolving s3-us-west-1.amazonaws.com (s3-us-west-1.amazonaws.com)... 52.219.116.192
Connecting to s3-us-west-1.amazonaws.com (s3-us-west-1.amazonaws.com)|52.219.116.192|:443... con
HTTP request sent, awaiting response... 200 OK
Length: 196739509 (188M) [application/zip]
Saving to: lfw.zip

lfw.zip             100%[===================>] 187.62M  21.2MB/s    in 9.8s

2020-06-07 08:22:14 (19.2 MB/s) - lfw.zip saved [196739509/196739509]
```

```
In [0]: import numpy as np
        from glob import glob


        human_files = np.array(glob("/content/data/lfw/*/*"))
        dog_files = np.array(glob("/content/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.

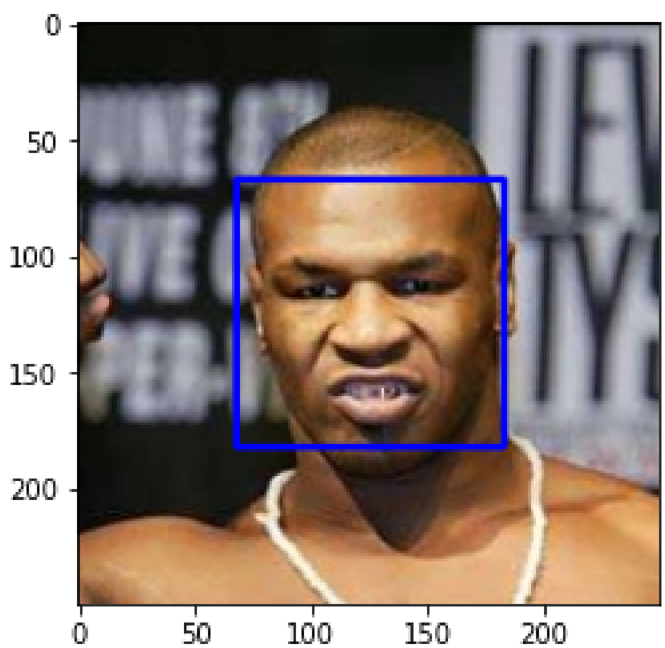
```
In [0]: if not os.path.exists("/content/haarcascades/"):
          os.makedirs("/content/haarcascades/")
        !wget https://raw.githubusercontent.com/udacity/deep-learning-v2-pytorch/master/project-
        shutil.copy("/content/haarcascade_frontalface_alt.xml", "/content/haarcascades/haarcasca
        os.remove("/content/haarcascade_frontalface_alt.xml")
```

```
--2020-06-07 08:22:28--  https://raw.githubusercontent.com/udacity/deep-learning-v2-pytorch/mast
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.0.133, 151.101.64.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.0.133|:443... connec
HTTP request sent, awaiting response... 200 OK
Length: 676709 (661K) [text/plain]
Saving to: haarcascade_frontalface_alt.xml

haarcascade_frontal 100%[===================>] 660.85K  --.-KB/s    in 0.04s

2020-06-07 08:22:29 (14.6 MB/s) - haarcascade_frontalface_alt.xml saved [676709/676709]
```

```
In [0]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline
        from shutil import copy
```

```
# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('/content/haarcascades/haarcascade_frontalface_alt.

# 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 [0]: # 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 [0]: 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_detected_in_human_files = 0
        faces_detected_in_dog_files = 0

        for i in range(len(human_files_short)):
            if face_detector(human_files_short[i]):
```

```
                faces_detected_in_human_files += 1
            if face_detector(dog_files_short[i]):
                faces_detected_in_dog_files += 1
        print("Percentage of faces detected in human files: {:.2f}%\n".format(faces_detected_in_
              "Percentage of faces detected in dog files: {:.2f}%".format(faces_detected_in_dog_
```

```
Percentage of faces detected in human files: 99.00%
Percentage of faces detected in dog files: 10.00%
```

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 [0]: ### (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 [0]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/torch/chec
```

```
HBox(children=(FloatProgress(value=0.0, max=553433881.0), HTML(value='')))
```

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 [0]: from PIL import Image
        import torchvision.transforms as transforms
        from torchvision import datasets
        from torch.autograd import Variable
        from PIL import ImageFile
        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
            '''
            normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                             std=[0.229, 0.224, 0.225])
            image_transform = transforms.Compose([transforms.RandomSizedCrop(224),
                    transforms.ToTensor(),
                    normalize])
            image = Image.open(img_path)
        #     plt.imshow(image)
        #     plt.show()
            image_tensor = image_transform(image)
            image_tensor.unsqueeze_(0)

            model_input = Variable(image_tensor)
            if use_cuda:
```

```
            model_input=model_input.cuda()

        output = VGG16(model_input)
        _, pred_tensor = torch.max(output, 1)
        pred = np.squeeze(pred_tensor.numpy()) if not use_cuda else np.squeeze(pred_tensor.c
        ## 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
        return pred # predicted class index
    #Test run with dog image from the list
    print(VGG16_predict(dog_files_short[47]))
```

```
195
```

```
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:698: UserWarning: Th
  "please use transforms.RandomResizedCrop instead.")
```

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 [0]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            pred_index = VGG16_predict(img_path)
            return  pred_index>150 and pred_index<269  # 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 dog detected in dog files short: 97.00%
- Percentage of dog detected in human files short: 1.00%

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

```
for i in range(len(dog_files_short)):
    if dog_detector(dog_files_short[i]):
        dog_detected_in_dog_files +=1;
    if dog_detector(human_files_short[i]):
        dog_detected_in_human_files +=1;
print("Percentage of dog detected in dog files short: {:.2f}%\n".format(dog_detected_in_
      +"Percentage of dog detected in human files short: {:.2f}%".format(dog_detected_in
```

```
/usr/local/lib/python3.6/dist-packages/torchvision/transforms/transforms.py:698: UserWarning: Th
  "please use transforms.RandomResizedCrop instead.")
```

```
Percentage of dog detected in dog files short: 97.00%
Percentage of dog detected in human files short: 1.00%
```

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [0]: import os
        from torchvision import datasets
        from shutil import copy
        import random

        ### TODO: Write data loaders for training, validation, and test sets
        ## Specify appropriate transforms, and batch_sizes
        normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                         std=[0.229, 0.224, 0.225])



        # normalize = transforms.Normalize(mean=[0.5, 0.5, 0.5],
                                    #  std=[0.5, 0.5, 0.5])
        train_transform = transforms.Compose([transforms.RandomRotation(30),
                                    transforms.RandomResizedCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(), normalize])

        test_transform = transforms.Compose([transforms.Resize(256), transforms.CenterCrop(224),

        train_path = "/content/data/dog_images/train"
        valid_path = "/content/data/dog_images/valid"
        test_path = "/content/data/dog_images/test"
        train_dataset = datasets.ImageFolder(train_path, transform=train_transform)
        valid_dataset = datasets.ImageFolder(valid_path, transform=test_transform)
        test_dataset = datasets.ImageFolder(test_path, transform=test_transform)
```

```
no_classes = len(next(os.walk(train_path))[1])
print("numbers of breeds in folders: {}".format( no_classes))

batch_size = 32
num_worker = 0
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, num_wor
valid_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, num_wor
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, num_worke
```

```
numbers of breeds in folders: 133
```

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 resized the images to 224 which will help keep the input tensor uniform during training so as to avoid disparity between input images.
- Yes the dataset were augmented. The augmentation applied includes randomly rotating it 30 degree and also fliping it horizontally (left and right). This will help it get variety of images to train on that will match with real world scenario.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [0]: 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__()
                self.conv1 = nn.Conv2d(3, 16, 3)
                self.bn1_2 = nn.BatchNorm2d(16)
                self.conv2 = nn.Conv2d(16, 32, 3)
                self.bn2_3 = nn.BatchNorm2d(32)
                self.conv3 = nn.Conv2d(32, 64, 3)
                self.bn3_4 = nn.BatchNorm2d(64)
                self.conv4 = nn.Conv2d(64, 128, 3)
                self.bn4 = nn.BatchNorm2d(128)

                self.pool = nn.MaxPool2d(2, 2)

                self.fc1 = nn.Linear(128 * 12 * 12, 1024)
                self.fc_out = nn.Linear(1024, no_classes)
```

```
            self.dropout = nn.Dropout(p=0.2)
            ## Define layers of a CNN

            # nn.init.kaiming_normal_(self.conv1.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv2.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv3.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv4.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv5.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv6.weight, nonlinearity='relu')
            # nn.init.kaiming_normal_(self.conv7.weight, nonlinearity='relu')

        def forward(self, x):
            ## Define forward behavior
            x = self.bn1_2(self.pool(F.relu(self.conv1(x))))
            x = self.bn2_3(self.pool(F.relu(self.conv2(x))))
            x = self.bn3_4(self.pool(F.relu(self.conv3(x))))
            x = self.bn4(self.pool(F.relu(self.conv4(x))))
            # print(x.shape)
            x = F.relu(self.fc1(self.dropout(x.view(-1,128 * 12 * 12))))
            x = self.fc_out(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))
  (bn1_2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (bn2_3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (bn3_4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (bn4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=18432, out_features=1024, bias=True)
  (fc_out): Linear(in_features=1024, out_features=133, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reason-

ing at each step.
Answer:

1. Four convolutional layers were added to the network with Max Pooling layers in between them to reduce the size of the input. This layers are used to extract the feautures in the images.
2. The output of each of this resized layers are normalized using Batch Normalization.
3. Two fully connected layers were added to help classify the images.

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 [0]: import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [0]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
            """returns trained model"""
            # initialize tracker for minimum validation loss
            valid_loss_min = np.Inf

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

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

```
            output = model(data)
            loss = criterion(output, target)
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            loss.backward()
            optimizer.step()

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


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

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

    # return trained model
    return model

loaders_scratch ={'train':train_loader,
                  'valid':valid_loader,
                  'test':test_loader}
# train the model
model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 6.677519        Validation Loss: 4.795866
Validation loss has decreased FROM inf To 4.795866.  Saving model ...
Epoch: 2         Training Loss: 4.711567        Validation Loss: 4.563931
Validation loss has decreased FROM 4.795866 To 4.563931.  Saving model ...
Epoch: 3         Training Loss: 4.556044        Validation Loss: 4.451049
Validation loss has decreased FROM 4.563931 To 4.451049.  Saving model ...
Epoch: 4         Training Loss: 4.459962        Validation Loss: 4.352761
Validation loss has decreased FROM 4.451049 To 4.352761.  Saving model ...
Epoch: 5         Training Loss: 4.394208        Validation Loss: 4.250066
Validation loss has decreased FROM 4.352761 To 4.250066.  Saving model ...
Epoch: 6         Training Loss: 4.351172        Validation Loss: 4.130901
Validation loss has decreased FROM 4.250066 To 4.130901.  Saving model ...
Epoch: 7         Training Loss: 4.269627        Validation Loss: 4.260086
Epoch: 8         Training Loss: 4.205959        Validation Loss: 4.073722
Validation loss has decreased FROM 4.130901 To 4.073722.  Saving model ...
Epoch: 9         Training Loss: 4.147186        Validation Loss: 4.311418
Epoch: 10         Training Loss: 4.075283        Validation Loss: 3.861610
Validation loss has decreased FROM 4.073722 To 3.861610.  Saving model ...
Epoch: 11         Training Loss: 4.017252        Validation Loss: 4.139404
Epoch: 12         Training Loss: 3.973511        Validation Loss: 3.793162
Validation loss has decreased FROM 3.861610 To 3.793162.  Saving model ...
Epoch: 13         Training Loss: 3.907269        Validation Loss: 3.663012
Validation loss has decreased FROM 3.793162 To 3.663012.  Saving model ...
Epoch: 14         Training Loss: 3.874718        Validation Loss: 3.881743
Epoch: 15         Training Loss: 3.804715        Validation Loss: 3.585264
Validation loss has decreased FROM 3.663012 To 3.585264.  Saving model ...
Epoch: 16         Training Loss: 3.747337        Validation Loss: 3.526040
Validation loss has decreased FROM 3.585264 To 3.526040.  Saving model ...
Epoch: 17         Training Loss: 3.700963        Validation Loss: 3.403691
Validation loss has decreased FROM 3.526040 To 3.403691.  Saving model ...
Epoch: 18         Training Loss: 3.653000        Validation Loss: 3.316816
Validation loss has decreased FROM 3.403691 To 3.316816.  Saving model ...
Epoch: 19         Training Loss: 3.596281        Validation Loss: 3.399192
Epoch: 20         Training Loss: 3.565479        Validation Loss: 3.321988
Epoch: 21         Training Loss: 3.522856        Validation Loss: 3.234654
Validation loss has decreased FROM 3.316816 To 3.234654.  Saving model ...
Epoch: 22         Training Loss: 3.509855        Validation Loss: 3.468999
Epoch: 23         Training Loss: 3.453242        Validation Loss: 3.247322
Epoch: 24         Training Loss: 3.453266        Validation Loss: 3.211021
Validation loss has decreased FROM 3.234654 To 3.211021.  Saving model ...
Epoch: 25         Training Loss: 3.392643        Validation Loss: 3.028210
Validation loss has decreased FROM 3.211021 To 3.028210.  Saving model ...
Epoch: 26         Training Loss: 3.358542        Validation Loss: 3.334310
Epoch: 27         Training Loss: 3.362387        Validation Loss: 3.079508
Epoch: 28         Training Loss: 3.340489        Validation Loss: 3.159021
Epoch: 29         Training Loss: 3.300622        Validation Loss: 3.091383
Epoch: 30         Training Loss: 3.292021        Validation Loss: 3.102950
Epoch: 31         Training Loss: 3.245835        Validation Loss: 2.999722
```

```
Validation loss has decreased FROM 3.028210 To 2.999722.  Saving model ...
Epoch: 32        Training Loss: 3.227304        Validation Loss: 2.987042
Validation loss has decreased FROM 2.999722 To 2.987042.  Saving model ...
Epoch: 33        Training Loss: 3.221271        Validation Loss: 3.033463
Epoch: 34        Training Loss: 3.159656        Validation Loss: 3.025815
Epoch: 35        Training Loss: 3.150428        Validation Loss: 3.031792
Epoch: 36        Training Loss: 3.161345        Validation Loss: 2.948383
Validation loss has decreased FROM 2.987042 To 2.948383.  Saving model ...
Epoch: 37        Training Loss: 3.106787        Validation Loss: 2.892911
Validation loss has decreased FROM 2.948383 To 2.892911.  Saving model ...
Epoch: 38        Training Loss: 3.121562        Validation Loss: 2.947083
Epoch: 39        Training Loss: 3.061282        Validation Loss: 2.976289
Epoch: 40        Training Loss: 3.053018        Validation Loss: 2.882959
Validation loss has decreased FROM 2.892911 To 2.882959.  Saving model ...
Epoch: 41        Training Loss: 3.048322        Validation Loss: 2.942566
Epoch: 42        Training Loss: 3.000646        Validation Loss: 2.961752
Epoch: 43        Training Loss: 2.974874        Validation Loss: 2.928413
Epoch: 44        Training Loss: 2.974280        Validation Loss: 2.758594
Validation loss has decreased FROM 2.882959 To 2.758594.  Saving model ...
Epoch: 45        Training Loss: 2.957414        Validation Loss: 2.897807
Epoch: 46        Training Loss: 2.932049        Validation Loss: 2.826521
Epoch: 47        Training Loss: 2.910461        Validation Loss: 2.849344
Epoch: 48        Training Loss: 2.916368        Validation Loss: 2.799824
Epoch: 49        Training Loss: 2.867408        Validation Loss: 2.815382
Epoch: 50        Training Loss: 2.899520        Validation Loss: 2.676108
Validation loss has decreased FROM 2.758594 To 2.676108.  Saving model ...
```

```
Out[0]: <All keys matched successfully>
```

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 [0]: 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: 2.664031


Test Accuracy: 30% (259/836)
```

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 [0]: ## TODO: Specify data loaders
        train_loader_transfer = train_loader
        valid_loader_transfer = valid_loader
        test_loader_transfer = test_loader
        loaders_transfer = {
            "train":train_loader_transfer,
            "valid":valid_loader_transfer,
            "test":test_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 [0]: import torchvision.models as models
        import torch.nn as nn

        ## TODO: Specify model architecture
        model_transfer = models.vgg16(pretrained=True)
        # print(model_transfer)
        for params in model_transfer.features.parameters():
          params.requires_grad = False
        in_feature = model_transfer.classifier[6].in_features
        model_transfer.classifier[6] = nn.Linear(in_feature, no_classes)
        print(model_transfer)
        # for param in models_tranfer.
        if use_cuda:
            model_transfer = model_transfer.cuda()
        #Initialize the weights of the classifier
        # classname = model_transfer.__class__.__name__
        #if classname.find("Linear") != -1:
        #   n = model_transfer.in_features
        #   y = 1.0/np.sqrt(n)
        #   model_transfer.weight.data.normal_(0.0,y)
        #   model_transfer.bias.data.fill_(0)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (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=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (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=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (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=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (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=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (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:

1. VGG16 was selected due to the fact that amongst its dataset it was trained on earlier, includes dog breeds which makes it closer to our intended use case.
2. Since it was trained on the dataset that is similar or the same as our dataset, I decided to freeze the feature extractor and only update the weights of the classifier
3. To ensure that the output matches the numbers of classes I would want to predict on, I modified the last fully connected layer of the pretrained VGG16

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [0]: # train the model
        n_epochs = 40
        model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, c

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

```
Epoch: 1         Training Loss: 4.561782         Validation Loss: 3.670566
Validation loss has decreased FROM inf To 3.670566.  Saving model ...
Epoch: 2         Training Loss: 3.572931         Validation Loss: 2.344476
Validation loss has decreased FROM 3.670566 To 2.344476.  Saving model ...
Epoch: 3         Training Loss: 2.661295         Validation Loss: 1.415962
Validation loss has decreased FROM 2.344476 To 1.415962.  Saving model ...
Epoch: 4         Training Loss: 2.042117         Validation Loss: 0.970399
Validation loss has decreased FROM 1.415962 To 0.970399.  Saving model ...
Epoch: 5         Training Loss: 1.720336         Validation Loss: 0.731824
Validation loss has decreased FROM 0.970399 To 0.731824.  Saving model ...
Epoch: 6         Training Loss: 1.551549         Validation Loss: 0.667210
Validation loss has decreased FROM 0.731824 To 0.667210.  Saving model ...
Epoch: 7         Training Loss: 1.413841         Validation Loss: 0.581027
Validation loss has decreased FROM 0.667210 To 0.581027.  Saving model ...
Epoch: 8         Training Loss: 1.390169         Validation Loss: 0.545203
Validation loss has decreased FROM 0.581027 To 0.545203.  Saving model ...
Epoch: 9         Training Loss: 1.273497         Validation Loss: 0.513324
Validation loss has decreased FROM 0.545203 To 0.513324.  Saving model ...
Epoch: 10         Training Loss: 1.241659         Validation Loss: 0.467019
Validation loss has decreased FROM 0.513324 To 0.467019.  Saving model ...
Epoch: 11         Training Loss: 1.208951         Validation Loss: 0.439257
Validation loss has decreased FROM 0.467019 To 0.439257.  Saving model ...
Epoch: 12         Training Loss: 1.167869         Validation Loss: 0.453921
Epoch: 13         Training Loss: 1.153257         Validation Loss: 0.415434
Validation loss has decreased FROM 0.439257 To 0.415434.  Saving model ...
Epoch: 14         Training Loss: 1.098086         Validation Loss: 0.424927
Epoch: 15         Training Loss: 1.089763         Validation Loss: 0.436207
Epoch: 16         Training Loss: 1.082598         Validation Loss: 0.413654
Validation loss has decreased FROM 0.415434 To 0.413654.  Saving model ...
Epoch: 17         Training Loss: 1.045529         Validation Loss: 0.381369
Validation loss has decreased FROM 0.413654 To 0.381369.  Saving model ...
Epoch: 18         Training Loss: 1.042546         Validation Loss: 0.440838
Epoch: 19         Training Loss: 1.032608         Validation Loss: 0.395911
Epoch: 20         Training Loss: 0.991486         Validation Loss: 0.373329
Validation loss has decreased FROM 0.381369 To 0.373329.  Saving model ...
Epoch: 21         Training Loss: 0.996877         Validation Loss: 0.381414
Epoch: 22         Training Loss: 0.984308         Validation Loss: 0.355496
Validation loss has decreased FROM 0.373329 To 0.355496.  Saving model ...
Epoch: 23         Training Loss: 0.968447         Validation Loss: 0.386851
Epoch: 24         Training Loss: 0.961907         Validation Loss: 0.367771
Epoch: 25         Training Loss: 0.944104         Validation Loss: 0.362869
```

```
Epoch: 26         Training Loss: 0.967790         Validation Loss: 0.375892
Epoch: 27         Training Loss: 0.980886         Validation Loss: 0.347586
Validation loss has decreased FROM 0.355496 To 0.347586.  Saving model ...
Epoch: 28         Training Loss: 0.957565         Validation Loss: 0.344028
Validation loss has decreased FROM 0.347586 To 0.344028.  Saving model ...
Epoch: 29         Training Loss: 0.911307         Validation Loss: 0.342556
Validation loss has decreased FROM 0.344028 To 0.342556.  Saving model ...
Epoch: 30         Training Loss: 0.930559         Validation Loss: 0.344033
Epoch: 31         Training Loss: 0.921142         Validation Loss: 0.359886
Epoch: 32         Training Loss: 0.909437         Validation Loss: 0.384672
Epoch: 33         Training Loss: 0.907102         Validation Loss: 0.338666
Validation loss has decreased FROM 0.342556 To 0.338666.  Saving model ...
Epoch: 34         Training Loss: 0.898809         Validation Loss: 0.340922
Epoch: 35         Training Loss: 0.876800         Validation Loss: 0.336083
Validation loss has decreased FROM 0.338666 To 0.336083.  Saving model ...
Epoch: 36         Training Loss: 0.897590         Validation Loss: 0.342708
Epoch: 37         Training Loss: 0.899676         Validation Loss: 0.339065
Epoch: 38         Training Loss: 0.868810         Validation Loss: 0.333839
Validation loss has decreased FROM 0.336083 To 0.333839.  Saving model ...
Epoch: 39         Training Loss: 0.870824         Validation Loss: 0.344824
Epoch: 40         Training Loss: 0.848262         Validation Loss: 0.337090
```

```
Out[0]: <All keys matched successfully>
```

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

```
Test Loss: 0.396553


Test Accuracy: 86% (722/836)
```

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

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