dog_app

March 16, 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
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

```
There are 13233 total human images.
There are 8351 total dog images.
```

Step 1: Detect Humans

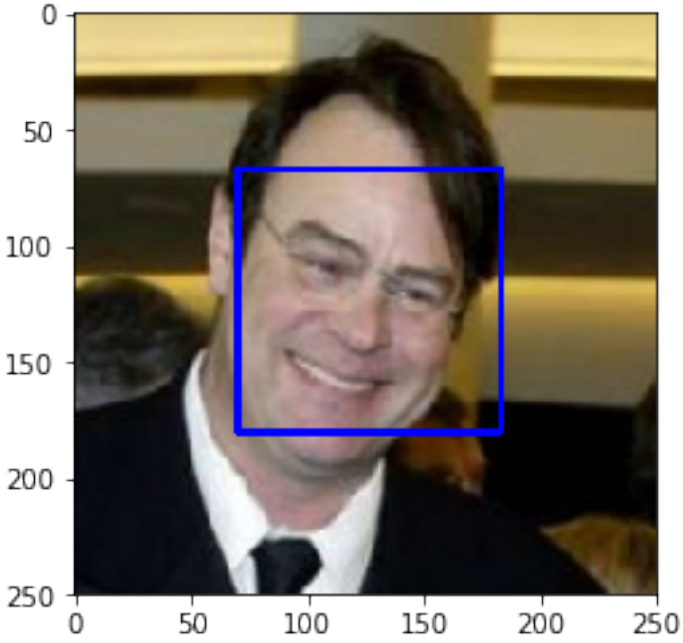
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github. We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

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

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        # find faces in image
        faces = face_cascade.detectMultiScale(gray)

        # print number of faces detected in the image
        print('Number of faces detected:', len(faces))

        # get bounding box for each detected face
        for (x,y,w,h) in faces:
            # add bounding box to color image
            cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
```

```
        # convert BGR image to RGB for plotting
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

        # display the image, along with bounding box
        plt.imshow(cv_rgb)
        plt.show()
```

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

1.1.1 Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

```
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer: (You can print out your results and/or write your percentages in this cell)

```
In [4]: from tqdm import tqdm_notebook as 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 = 0
        for i, human_image in enumerate(tqdm(human_files_short, desc="Human faces")):
            img = cv2.imread(human_image)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            if(len(faces)>0):
                faces_detected+=1

        dogs_imaged_with_detected_face_count = 0
        for i, dog_image in enumerate(tqdm(dog_files_short, desc="Dog faces")):
            img = cv2.imread(dog_image)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            if(len(faces)>0):
                dogs_imaged_with_detected_face_count += 1

        print("% of faces detected ",(faces_detected/ len(human_files_short))*100 )
        print("% of dog detected with human face ",(dogs_imaged_with_detected_face_count/ len(do

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

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

```
% of faces detected  98.0
% of dog detected with human face  17.0
```

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

In [5]:
```
### (Optional)
### TODO: Test performance of anotherface detection algorithm.
### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

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

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

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

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

```
In [7]: from PIL import Image
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        import torchvision.transforms as transforms
        from torch.autograd import Variable

        min_img_size = 220  # The min size, as noted in the PyTorch pretrained models doc, is 22
        transform_pipeline = transforms.Compose([
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                transforms.Normalize(mean=[0.485, 0.456, 0.406]
                                                                     std=[0.229, 0.224, 0.225])



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

            Args:
                img_path: path to an image

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

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

            image = Image.open(img_path)
            image = transform_pipeline(image)
            image = image.unsqueeze(0)
            image = Variable(image)

            if use_cuda:
                image = image.cuda()
```

```
    prediction = VGG16(image)

    if use_cuda:
        prediction = prediction.cpu()
        index = np.argmax(prediction.detach().numpy())
    else:
        index = prediction.data.numpy().argmax()
    return index # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index = VGG16_predict(img_path)
            if index >= 151 and index <= 268:
                return True
            else:
                return False     # 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:
% of faces detected 0.0%
% of dog detected with human face 92.0%

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


        faces_detected = 0
        for i, human_image in enumerate(tqdm(human_files_short, desc="Human faces")):
            prediction = dog_detector(human_image)
            if prediction:
                faces_detected += 1

        dogs_imaged_with_detected_face_count = 0
```

```
        for dog_image in tqdm(dog_files_short, desc="Dog faces"):
            prediction = dog_detector(dog_image)
            if prediction:
                dogs_imaged_with_detected_face_count += 1

        print("% of faces detected ",(faces_detected/ len(human_files_short))*100, "%" )
        print("% of dog detected with human face ",(dogs_imaged_with_detected_face_count/ len(do
```

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

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

```
% of faces detected  1.0 %
% of dog detected with human face  92.0 %
```

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

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.


         resnet50 = 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:
             resnet50 = resnet50.cuda()


         min_img_size = 220  # The min size, as noted in the PyTorch pretrained models doc, is 2
         transform_pipeline = transforms.Compose([
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                                    std=[0.229, 0.224, 0.225]
```

```
def RESNET50_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
    predicted ImageNet class for image at specified path

    Args:
        img_path: path to an image

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

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

    image = Image.open(img_path)
    image = transform_pipeline(image)
    image = image.unsqueeze(0)
    image = Variable(image)

    if use_cuda:
        image = image.cuda()

    prediction = resnet50(image)
    if use_cuda:
        prediction = prediction.cpu()
        index = np.argmax(prediction.detach().numpy())
    else:
        index = prediction.data.numpy().argmax()

    return index # predicted class index


def dog_detector(img_path):
    ## TODO: Complete the function.
    index = RESNET50_predict(img_path)
    #print(index)
    if index >= 151 and index <= 268:
        return True
    else:
        return False # true/false


faces_detected = 0
for i, human_image in enumerate(tqdm(human_files_short, desc="Human faces")):
```

```
            prediction = dog_detector(human_image)
            if prediction:
                faces_detected += 1

        dogs_imaged_with_detected_face_count = 0
        for dog_image in tqdm(dog_files_short, desc="Dog faces"):
            prediction = dog_detector(dog_image)
            if prediction:
                dogs_imaged_with_detected_face_count += 1

        print("% of faces detected ",(faces_detected/ len(human_files_short))*100, "%" )
        print("% of dog detected with human face ",(dogs_imaged_with_detected_face_count/ len(d
```

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

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

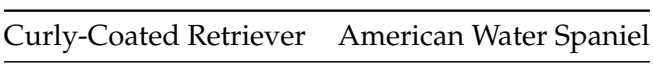
```
% of faces detected  1.0 %
% of dog detected with human face  91.0 %
```

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.

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



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 [11]: import os
         from torchvision import datasets


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

         num_workers = 1
         batch_size = 32

         data_dir = "/data/dog_images"

         transform = transforms.Compose([transforms.Resize(size=(240,240)),
                                         transforms.ToTensor(),
                                         transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0


         train_data = datasets.ImageFolder(data_dir + "/train", transform=transform)

         validation_data = datasets.ImageFolder(data_dir + "/valid", transform=transform)

         test_data = datasets.ImageFolder(data_dir + "/test", transform=transform)


         train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                    num_workers=num_workers,shuffle=True)
         valid_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size,
```

```
                                                num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=4,
                                          num_workers=num_workers)


loaders_scratch = {}
loaders_scratch['train'] = train_loader
loaders_scratch['valid'] = valid_loader
loaders_scratch['test'] = test_loader
```

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

Answer:

- I have used transforms.Resize(size=(240,240) to resize image. All pre-trained model accept images of size 240 x 240, so I chose the same.
- Have not used augmentation because we have sufficient images for each dog breed plus all the image are random (i.e. dogs in different position and with different background not like MNIST dataset). If required, we can use data agumentation techniques like flips, rotations and flips etc.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: 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, 32, 3, padding=1)
                 self.bn1 = nn.BatchNorm2d(32)

                 self.conv2 = nn.Conv2d(32, 64, 3, padding=0)
                 self.bn2 = nn.BatchNorm2d(64)

                 self.conv3 = nn.Conv2d(64, 128, 3, padding=0)
                 self.bn3 = nn.BatchNorm2d(128)

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

```
                self.fc1 = nn.Linear(28 * 28 * 128, 133)

                self.dropout_dense = nn.Dropout(0.25)


            def forward(self, x):
                ## Define forward behavior

                x = self.conv1(x)
                x = F.relu(x)
                x = self.pool(x)
                x = self.bn1(x)

                x = self.conv2(x)
                x = F.relu(x)
                x = self.pool(x)
                x = self.bn2(x)

                x = self.conv3(x)
                x = F.relu(x)
                x = self.pool(x)
                x = self.bn3(x)

                x = x.view(-1, 28 * 28 * 128)
                x = self.dropout_dense(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()
            model_scratch = nn.DataParallel(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (bn3): 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=100352, out_features=133, bias=True)
  (dropout_dense): Dropout(p=0.25)
)
```

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

Answer:

1. Used magic depths - 32, 64, 128 to increase depth i.e. 3--> 32 --> 64 --> 128
2. From CNN classic literature - Max Pooling of 2X2
3. From CNN classic literature - relu activation
4. Used BatchNormalization to speed up the process.
5. Linear of 28 28 128 --> 133 (number of classes)

Before finalizing this architecture, I tried various conv. number of layer of different depth, learning rates, with and without batch normalization on Google cloud platform

(refer notebook - dog_classification_12_perc.ipynb and model_scratch_12_perc.pt) which gave me an accuracy ~ 12 percent)

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 [13]:
```
import torch.optim as optim

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

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

optimizer_scratch = optim.Adam([
    {'params': model_scratch.parameters(), 'weight_decay': 0.1, 'amsgrad': True}
], lr=0.00001)

#optimizer_scratch = optim.Adam([
#    {'params': model_scratch.parameters(), 'amsgrad': True}
#])

#print(optimizer_scratch)
```

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 [14]:
```
import numpy as np
from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
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()
            #print(data)
            ## 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

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

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            #if(batch_idx % 20 == 0):
            #    print("batch_idx {} , train_loss {}".format(batch_idx, train_loss))

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

```
                valid_loss += loss.item()*data.size(0)
                #valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss

            #train_loss = train_loss/len(train_loader.dataset)
            valid_loss = valid_loss/len(loaders['valid'])
            # 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 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
```

In [28]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,criterion_s
```

```
Epoch: 1         Training Loss: 0.374696         Validation Loss: 122.616043
Validation loss decreased (inf --> 122.616043).  Saving model ...
Epoch: 2         Training Loss: 0.300666         Validation Loss: 122.495368
Validation loss decreased (122.616043 --> 122.495368).  Saving model ...
Epoch: 3         Training Loss: 0.270180         Validation Loss: 122.391167
Validation loss decreased (122.495368 --> 122.391167).  Saving model ...
Epoch: 4         Training Loss: 0.251777         Validation Loss: 122.329501
Validation loss decreased (122.391167 --> 122.329501).  Saving model ...
Epoch: 5         Training Loss: 0.239755         Validation Loss: 122.141388
Validation loss decreased (122.329501 --> 122.141388).  Saving model ...
Epoch: 6         Training Loss: 0.229713         Validation Loss: 121.975935
Validation loss decreased (122.141388 --> 121.975935).  Saving model ...
Epoch: 7         Training Loss: 0.226499         Validation Loss: 122.036738
Epoch: 8         Training Loss: 0.221204         Validation Loss: 121.760960
Validation loss decreased (121.975935 --> 121.760960).  Saving model ...
Epoch: 9         Training Loss: 0.218509         Validation Loss: 121.738866
Validation loss decreased (121.760960 --> 121.738866).  Saving model ...
Epoch: 10         Training Loss: 0.217720         Validation Loss: 121.731142
Validation loss decreased (121.738866 --> 121.731142).  Saving model ...
Epoch: 11         Training Loss: 0.215239         Validation Loss: 121.686240
Validation loss decreased (121.731142 --> 121.686240).  Saving model ...
Epoch: 12         Training Loss: 0.213650         Validation Loss: 121.549599
Validation loss decreased (121.686240 --> 121.549599).  Saving model ...
```

```
Epoch: 13         Training Loss: 0.212106         Validation Loss: 121.548381
Validation loss decreased (121.549599 --> 121.548381).  Saving model ...
Epoch: 14         Training Loss: 0.211994         Validation Loss: 121.489690
Validation loss decreased (121.548381 --> 121.489690).  Saving model ...
Epoch: 15         Training Loss: 0.211824         Validation Loss: 121.366655
Validation loss decreased (121.489690 --> 121.366655).  Saving model ...
Epoch: 16         Training Loss: 0.211728         Validation Loss: 121.461820
Epoch: 17         Training Loss: 0.212111         Validation Loss: 121.367900
Epoch: 18         Training Loss: 0.210061         Validation Loss: 121.179659
Validation loss decreased (121.366655 --> 121.179659).  Saving model ...
Epoch: 19         Training Loss: 0.211541         Validation Loss: 121.243202
Epoch: 20         Training Loss: 0.210933         Validation Loss: 121.121581
Validation loss decreased (121.179659 --> 121.121581).  Saving model ...
```

```
In [29]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [30]: 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))
```

```
In [31]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.871747


Test Accuracy: 13% (113/836)
```

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 [32]: ## TODO: Specify data loaders

         import os
         from torchvision import datasets
         from torchvision.transforms import transforms

         import torch

         num_workers = 1
         batch_size = 32

         data_dir = "/data/dog_images"

         train_transform = transforms.Compose([transforms.RandomResizedCrop(size=256, scale=(0.8
                                               transforms.RandomRotation(degrees=15),
                                               transforms.ColorJitter(),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.CenterCrop(size=224),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0

         valid_tranform = transforms.Compose([transforms.Resize(256),
```

```
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize([0.485, 0.456, 0.406], [0.229


transform = transforms.Compose([transforms.Resize(size=(240,240)),
                                transforms.ToTensor(),
                                transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0


train_data = datasets.ImageFolder(data_dir + "/train", transform=transform)

validation_data = datasets.ImageFolder(data_dir + "/valid", transform=transform)

test_data = datasets.ImageFolder(data_dir + "/test", transform=transform)


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

valid_loader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size,
                                           num_workers=num_workers,shuffle=True)

test_loader = torch.utils.data.DataLoader(test_data, batch_size=4,
                                          num_workers=num_workers)


loaders_transfer = {}
loaders_transfer['train'] = train_loader
loaders_transfer['valid'] = valid_loader
loaders_transfer['test'] = test_loader
```

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

         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)
         for param in model_transfer.parameters():
             param.requires_grad = False


         num_features = model_transfer.classifier[6].in_features
```

```
        features = list(model_transfer.classifier.children())[:-1] # Remove last layer
        features.extend([nn.Linear(num_features, 133)]) # Add our layer with 4 outputs
        model_transfer.classifier = nn.Sequential(*features) # Replace the model classifier
        print(model_transfer)



        use_cuda = torch.cuda.is_available()

        print(use_cuda)

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

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

```
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
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:

Here I have tried data augmentation here and pretty much used all the image augmentation available + relevant for this exercise

1. Model loaded
2. Freezed all the weights of lower layer as we want to use same weights else no point in using pre-trained model for param in model_transfer.parameters():

   param.requires_grad = False
3. Removed last classification layer to insert custom linear layer (=num_of_classes of our example)

   features = list(model_transfer.classifier.children())[:-1] # Remove last layer

   features.extend([nn.Linear(num_features, 133)]) # Add our layer with 4 outputs

Trained CNN models - VGG16 is already trained on numerous images (ImageNet) and can be used to understand the result instead reinventing the wheel everytime

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 [34]:
```
import torch.optim as optim

        parameters = filter(lambda p: p.requires_grad, model_transfer.parameters())

        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adam(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 [41]: # train the model
         n_epochs = 20
         model_transfer =  train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

         # 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: 0.306214         Validation Loss: 14.803587
Validation loss decreased (inf --> 14.803587).  Saving model ...
Epoch: 2         Training Loss: 0.263015         Validation Loss: 14.142353
Validation loss decreased (14.803587 --> 14.142353).  Saving model ...
Epoch: 3         Training Loss: 0.236002         Validation Loss: 14.395684
Epoch: 4         Training Loss: 0.199256         Validation Loss: 15.350044
Epoch: 5         Training Loss: 0.202635         Validation Loss: 15.471076
Epoch: 6         Training Loss: 0.181103         Validation Loss: 17.302559
Epoch: 7         Training Loss: 0.179589         Validation Loss: 16.443067
Epoch: 8         Training Loss: 0.153826         Validation Loss: 16.946847
Epoch: 9         Training Loss: 0.164027         Validation Loss: 16.904138
Epoch: 10         Training Loss: 0.166786         Validation Loss: 18.510301
Epoch: 11         Training Loss: 0.160393         Validation Loss: 18.548638
Epoch: 12         Training Loss: 0.151886         Validation Loss: 18.469754
Epoch: 13         Training Loss: 0.134069         Validation Loss: 16.821843
Epoch: 14         Training Loss: 0.146338         Validation Loss: 18.568031
Epoch: 16         Training Loss: 0.146727         Validation Loss: 19.763020
Epoch: 17         Training Loss: 0.135419         Validation Loss: 20.251954
Epoch: 18         Training Loss: 0.138196         Validation Loss: 21.461893
Epoch: 19         Training Loss: 0.132056         Validation Loss: 20.932858
Epoch: 20         Training Loss: 0.140512         Validation Loss: 19.641161
```

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

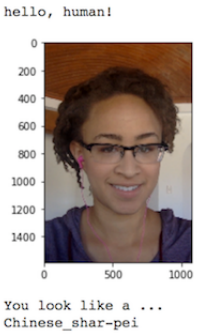
```
Test Loss: 0.545377


Test Accuracy: 85% (713/836)
```

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 [43]:
```
'''
Constructing dictionary of dog breed,
as out predictor will return a number and to show user the dog breed name, we need this
'''

import os

classes = list()

for dir_ in os.listdir("/data/dog_images/train"):
       classes.append(dir_)

loaders_transfer['train'].classes = classes

class_dict = {}
for item in loaders_transfer['train'].classes:
    index = item[:3]
    breed_name = item[4:].replace("_", " ")
    class_dict[int(index)] = breed_name

class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].classes
```

In [44]:
```
### 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_transfer['train'].classes
#print(class_names)

import numpy as np
from PIL import *
from torch.autograd import Variable
from torchvision.transforms import ToTensor

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

    data_transforms = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor()
    ])
```



Sample Human Output

```
    image = Image.open(img_path)
    image = data_transforms(image).float()
    image = Variable(image, requires_grad=False)
    image = image.unsqueeze(0)
    if use_cuda:
        image = image.cuda()

    model_transfer.eval()
    val = model_transfer(image)
    val_ = val.cpu()
    #print(val_)
    val2 = np.argmax(val_.detach().numpy())
    #print(val2)
    return class_dict[val2+1]

#indx = predict_breed_transfer('/data/dog_images/train/001.Affenpinscher/Affenpinscher_
#print(indx)
#indx = predict_breed_transfer('/data/dog_images/train/004.Akita/Akita_00280.jpg')
#print(indx)
#indx = predict_breed_transfer('/data/dog_images/train/010.Anatolian_shepherd_dog/Anato
#print(indx)
```

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!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [45]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         import numpy as np

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             if(dog_detector(img_path)):
                 return "is dog, predicted breed ->  "+predict_breed_transfer(img_path)
             elif(face_detector(img_path)):
                 return "is a human, resembling dog breed -> "+predict_breed_transfer(img_path)
             else:
                 return "Not a valid input"
                 #raise Exception("Not a valid input")
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement)

Model is performing average yes better than expected because vg166 is trained on different set of images and using it to classify as a dog and or human is working pretty good, like - is able to predict dogs as dogs, humans as humans and a invalid image (human + ape picture) as invalid.

It's getting confused between breeds that are very similar to each other.

Predicting -

```
labrador as American staffordshire terrier

Husky as Akita

Bulldog as Bulldog

Beagle as Beagle

Beagle as American foxhound
```

So with 85% accuracy model is trying it's best to predict dog breeds.
Beagle and American foxhound are very similar also Husky and Akita are very similar

To improve we can - 1. Get more images of each breed especially breeds which are similar to each other - pointer, beagle, labrador etc. 2. Using different form of image augmentation to make sure model perform good similar looking dog breeds 3. Train for more epochs with different learning rate

```
In [46]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
         #for file in np.hstack((human_files[:3], dog_files[:3])):
         for dir_ in os.listdir("downloaded_image/"):
             #print(dir_)
             if(dir_ == ".ipynb_checkpoints"):
                 continue
             print("[ input  ] = ", dir_ ,   "\n[ output ] = ",run_app("downloaded_image/" + dir
             print("\n-------------")
```

```
[ input  ] =  Beagle_dog.jpg
[ output ] =  is dog, predicted breed ->  Beagle

-------------
[ input  ] =  labrador_dog.jpg
[ output ] =  is dog, predicted breed ->  American staffordshire terrier

-------------
[ input  ] =  bulldog_dog.jpeg
[ output ] =  is dog, predicted breed ->  Bulldog

-------------
[ input  ] =  human_3.jpeg
[ output ] =  is a human, resembling dog breed -> American foxhound

-------------
[ input  ] =  invalid_input.jpeg
[ output ] =  Not a valid input

-------------
[ input  ] =  human_2.jpeg
[ output ] =  is a human, resembling dog breed -> Maltese

-------------
[ input  ] =  human_1.jpg
[ output ] =  is a human, resembling dog breed -> English toy spaniel

-------------
[ input  ] =  husky_dog.jpeg
```

```
[ output ] =  is dog, predicted breed ->  Akita

-------------
[ input  ] =  Beagle_dog.jpeg
[ output ] =  is dog, predicted breed ->  American foxhound

-------------
```