dog_app (1)

June 19, 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: * 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

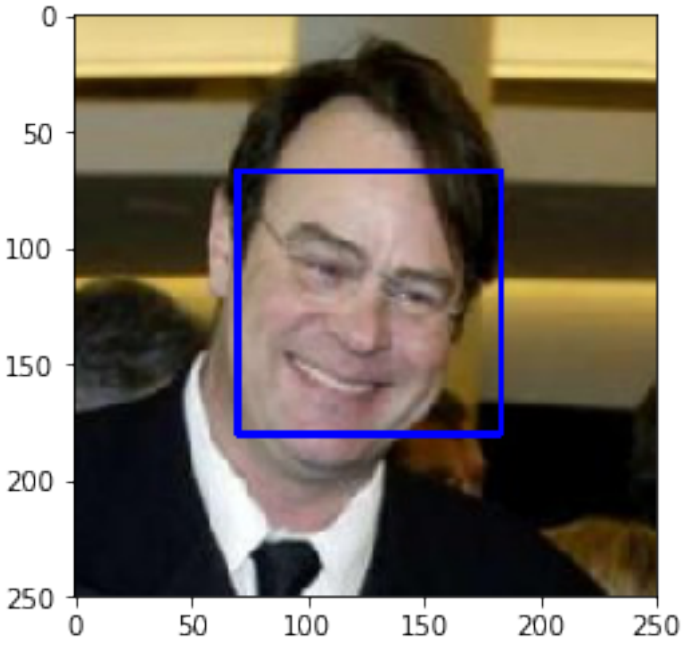
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
list_face_correct = []
list_face_false = []

face_count_correctlyDetected = 0
face_count_detected_in_dogFiles = 0

for i in range(len(human_files_short)):
    list_face_correct.append(face_detector(human_files_short[i]))
    if list_face_correct[i] == True:
        face_count_correctlyDetected +=1

for i in range(len(dog_files_short)):
    list_face_false.append(face_detector(dog_files_short[i]))
    if list_face_false[i] == True:
        face_count_detected_in_dogFiles +=1


#the accuracy of the human_images
accuracy_human_face = face_count_correctlyDetected / len(human_files_short)

#the accuracy of the dog_images
accuracy_dog_images = face_count_detected_in_dogFiles / len(dog_files_short)
```

```
        print('Faces detected in {:.2f}% of the sample human dataset.'.format(accuracy_human_fac
        print('Faces detected in {:.2f}% of the sample dog dataset.'.format(accuracy_dog_images*
```

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

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

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

Step 2: Detect Dogs

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:07<00:00, 76788605.57it/s]
```

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
        import torchvision.transforms as transforms

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

            Args:
                img_path: path to an image

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

            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            ## Return the *index* of the predicted class for that image
            #Open the image and convert it to RGB
            image = Image.open(img_path).convert('RGB')

            # All pre-trained models expect input images normalized in the same way,
            # i.e. mini-batches of 3-channel RGB images of shape (3 x H x W),
            # where H and W are expected to be at least 224. The images have to be loaded in to
            #then normalized using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].
            # We can use the following transform to normalize

            normalize = transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225])

            #convert data to a normalized torch.FloatTensor
            transform = transforms.Compose([transforms.Scale(224),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            normalize])

            # adding that extra "batch dimension" (the model wants a batch of images)
            image =  transform(image)[:3,:,:].unsqueeze(0)
```

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


    # getting a prediciton
    # Returns a Tensor of shape (batch, num class labels)
    predicition = VGG16(image)

    # Our prediction will be the index of the class label with the largest value
    # The second return value is the index location of each maximum value found (argmax)
    return torch.max(predicition,1)[1].item()   # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            prediction = VGG16_predict(img_path)

            return ((prediction <= 268) & (prediction >= 151))  # 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: - In the human_files : 0% dogs - In the dog_files : 100% dogs

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

        for h_file in human_files_short:
          if(dog_detector(h_file)):
            h_perc += 1

        for d_file in dog_files_short:
          if(dog_detector(d_file)):
            d_perc += 1
```

```
        print('In the human_files : {}% dogs'.format(h_perc))
        print('In the dog_files : {}% dogs'.format(d_perc))
```

```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
```

```
In the human_files : 0% dogs
In the dog_files : 100% dogs
```

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.
```

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

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

         import torchvision.transforms as transforms


         batch_size = 32

          # Image transformations
         image_transforms = {
             # Train uses data augmentation
             'train':
             transforms.Compose([
                 transforms.RandomResizedCrop(size=256, scale=(0.8, 1.0)),
                 transforms.RandomRotation(degrees=15),
                 transforms.ColorJitter(),
                 transforms.RandomHorizontalFlip(),
                 transforms.CenterCrop(size=224),  # Image net standards
                 transforms.ToTensor(),
                 transforms.Normalize([0.485, 0.456, 0.406],
                                      [0.229, 0.224, 0.225])  # Imagenet standards
             ]),
             # Validation does not use augmentation
             'valid':
             transforms.Compose([
                 transforms.Resize(size=256),
                 transforms.CenterCrop(size=224),
                 transforms.ToTensor(),
```

```
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
        # Test does not use augmentation
        'test':
        transforms.Compose([
            transforms.Resize(size=256),
            transforms.CenterCrop(size=224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ]),
    }

    # Location of data
    dog_file='/data/dog_images/'

    train=os.path.join(dog_file,'train')
    valid=os.path.join(dog_file,'valid')
    test=os.path.join(dog_file,'test')


    # Datasets from each folder
    train_file=datasets.ImageFolder(train,transform=image_transforms['train'])
    valid_file=datasets.ImageFolder(valid,transform=image_transforms['valid'])
    test_file=datasets.ImageFolder(test,transform=image_transforms['test'])


    # To avoid loading all of the data into memory at once, I used training DataLoaders. Fi
    # a dataset object from the image folders, and then I passed these to a DataLoader.
    # At training time, the DataLoader will load the images from disk, apply the transforma
    # and yield a batch. To train and validation, we'll iterate through all the batches in
    # DataLoader.
    # One crucial aspect is to shuffle the data before passing it to the network.
    # This means that the ordering of the image categories changes on each pass through the
    # (one pass through the data is one training epoch).



    loaders={
        'train':torch.utils.data.DataLoader(train_file,batch_size,shuffle=True),
        'valid':torch.utils.data.DataLoader(valid_file,batch_size,shuffle=True),
        'test': torch.utils.data.DataLoader(test_file,batch_size,shuffle=True)
    }
```

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:

During image preprocessing, I simultaneously prepare the images for our network and apply data augmentation to the training set. Each model will have different input requirements, but if I read through what Imagenet requires, I figure out that our images need to be 224x224 and normalized to a range. The transforms I used are as follows: - Resized - Image dataset is centered and cropped to 224 x 224 - Then it is converted to a tensor - Normalized with mean and standard deviation The end result of passing through these transforms are tensors that can go into our network.

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, 64, 11, stride=4, padding=2)

                 self.conv2 = nn.Conv2d(64, 192, 5, padding=2)

                 self.conv3 = nn.Conv2d(192, 256, 3, padding=2)

                 self.conv4 = nn.Conv2d(256, 512, 3, padding=2)



                 # max pooling layer
                 self.pool = nn.MaxPool2d(kernel_size=3, stride=2)


                 self.fc1 = nn.Linear(512 * 4 * 4, 5000)

                 self.fc2 = nn.Linear(5000, 2500)

                 self.fc3 = nn.Linear(2500, 500)

                 self.fc4 = nn.Linear(500, 133)
                 # dropout layer (p=0.25)
                 self.dropout = nn.Dropout(0.25)
                 # batch normalization
                 self.batn = nn.BatchNorm2d(192)
                 self.batn2 = nn.BatchNorm2d(256)
```

```
    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.batn(x)
        x = self.pool(F.relu(self.conv3(x)))
        x = self.batn2(x)
        x = self.pool(F.relu(self.conv4(x)))


        # flatten image input
        x = x.view(-1, 512 * 4 * 4)

        # fully connected layers
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = F.relu(self.fc3(x))
        x = self.fc4(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: General Architecture Gaussian filters, probably one of the most used filters in image processing, are based on gaussian function in which the top value is achieved on the axis of symmetry. Therefore, whatever this number of pixels maybe, the length of each side of our symmetrically shaped kernel is 2*n+1 (each side of the anchor + the anchor pixel), and therefore filter/kernels are always odd sized. This is the main reason why such kinds of kernels are preferably to be odd. So I used 3x3 kernel size with zero padding. Using a 3x3 filter expresses most information about the image across all the channels (zero padding allows us to achieve this). So I used 3x3 kernel size with zero padding. My CNN architecture consists of 4 Convolutional Layers and 4 Fully Connected Layers. Between fully connected layers, dropout technique(with probability = 0.25) is used to overcome the overfitting problem.

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.SGD(model_scratch.parameters(), lr=0.005)
```

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]:
```
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()
            ## find the loss and update the model parameters accordingly
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
```

```
            optimizer.step()
            # update training loss
            ## record the average training loss, using something like
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

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

            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

        # calculate average losses
        train_loss = train_loss/len(train_file)
        valid_loss = valid_loss/len(valid_file)


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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:
            torch.save(model.state_dict(), save_path)

            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.fo
            valid_loss_min,
            valid_loss))

            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
        # train the model
        model_scratch = train(50, loaders, 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: 0.000732         Validation Loss: 0.005851
Validation loss decreased (inf --> 0.005851).  Saving model ...
Epoch: 2         Training Loss: 0.000731         Validation Loss: 0.005847
Validation loss decreased (0.005851 --> 0.005847).  Saving model ...
Epoch: 3         Training Loss: 0.000730         Validation Loss: 0.005837
Validation loss decreased (0.005847 --> 0.005837).  Saving model ...
Epoch: 4         Training Loss: 0.000729         Validation Loss: 0.005829
Validation loss decreased (0.005837 --> 0.005829).  Saving model ...
Epoch: 5         Training Loss: 0.000728         Validation Loss: 0.005806
Validation loss decreased (0.005829 --> 0.005806).  Saving model ...
Epoch: 6         Training Loss: 0.000725         Validation Loss: 0.005792
Validation loss decreased (0.005806 --> 0.005792).  Saving model ...
Epoch: 7         Training Loss: 0.000721         Validation Loss: 0.005743
Validation loss decreased (0.005792 --> 0.005743).  Saving model ...
Epoch: 8         Training Loss: 0.000716         Validation Loss: 0.005704
Validation loss decreased (0.005743 --> 0.005704).  Saving model ...
Epoch: 9         Training Loss: 0.000709         Validation Loss: 0.005577
Validation loss decreased (0.005704 --> 0.005577).  Saving model ...
Epoch: 10         Training Loss: 0.000698         Validation Loss: 0.005532
Validation loss decreased (0.005577 --> 0.005532).  Saving model ...
Epoch: 11         Training Loss: 0.000682         Validation Loss: 0.005409
Validation loss decreased (0.005532 --> 0.005409).  Saving model ...
Epoch: 12         Training Loss: 0.000666         Validation Loss: 0.005332
Validation loss decreased (0.005409 --> 0.005332).  Saving model ...
Epoch: 13         Training Loss: 0.000649         Validation Loss: 0.005170
Validation loss decreased (0.005332 --> 0.005170).  Saving model ...
Epoch: 14         Training Loss: 0.000637         Validation Loss: 0.005098
Validation loss decreased (0.005170 --> 0.005098).  Saving model ...
Epoch: 15         Training Loss: 0.000626         Validation Loss: 0.005053
Validation loss decreased (0.005098 --> 0.005053).  Saving model ...
Epoch: 16         Training Loss: 0.000615         Validation Loss: 0.005019
Validation loss decreased (0.005053 --> 0.005019).  Saving model ...
Epoch: 17         Training Loss: 0.000605         Validation Loss: 0.005148
Epoch: 18         Training Loss: 0.000596         Validation Loss: 0.004940
Validation loss decreased (0.005019 --> 0.004940).  Saving model ...
Epoch: 19         Training Loss: 0.000588         Validation Loss: 0.004872
Validation loss decreased (0.004940 --> 0.004872).  Saving model ...
Epoch: 20         Training Loss: 0.000579         Validation Loss: 0.004723
Validation loss decreased (0.004872 --> 0.004723).  Saving model ...
Epoch: 21         Training Loss: 0.000571         Validation Loss: 0.004932
Epoch: 22         Training Loss: 0.000561         Validation Loss: 0.004658
```

```
Validation loss decreased (0.004723 --> 0.004658).  Saving model ...
Epoch: 23         Training Loss: 0.000554         Validation Loss: 0.004455
Validation loss decreased (0.004658 --> 0.004455).  Saving model ...
Epoch: 24         Training Loss: 0.000545         Validation Loss: 0.004563
Epoch: 25         Training Loss: 0.000539         Validation Loss: 0.004447
Validation loss decreased (0.004455 --> 0.004447).  Saving model ...
Epoch: 26         Training Loss: 0.000533         Validation Loss: 0.004530
Epoch: 27         Training Loss: 0.000521         Validation Loss: 0.004430
Validation loss decreased (0.004447 --> 0.004430).  Saving model ...
Epoch: 28         Training Loss: 0.000518         Validation Loss: 0.004680
Epoch: 29         Training Loss: 0.000509         Validation Loss: 0.004232
Validation loss decreased (0.004430 --> 0.004232).  Saving model ...
Epoch: 30         Training Loss: 0.000502         Validation Loss: 0.004614
Epoch: 31         Training Loss: 0.000494         Validation Loss: 0.004374
Epoch: 32         Training Loss: 0.000487         Validation Loss: 0.004546
Epoch: 33         Training Loss: 0.000478         Validation Loss: 0.004146
Validation loss decreased (0.004232 --> 0.004146).  Saving model ...
Epoch: 34         Training Loss: 0.000471         Validation Loss: 0.004114
Validation loss decreased (0.004146 --> 0.004114).  Saving model ...
Epoch: 35         Training Loss: 0.000465         Validation Loss: 0.004440
Epoch: 36         Training Loss: 0.000459         Validation Loss: 0.004240
Epoch: 37         Training Loss: 0.000449         Validation Loss: 0.003974
Validation loss decreased (0.004114 --> 0.003974).  Saving model ...
Epoch: 38         Training Loss: 0.000445         Validation Loss: 0.003937
Validation loss decreased (0.003974 --> 0.003937).  Saving model ...
Epoch: 39         Training Loss: 0.000436         Validation Loss: 0.004369
Epoch: 40         Training Loss: 0.000426         Validation Loss: 0.003868
Validation loss decreased (0.003937 --> 0.003868).  Saving model ...
Epoch: 41         Training Loss: 0.000419         Validation Loss: 0.003560
Validation loss decreased (0.003868 --> 0.003560).  Saving model ...
Epoch: 42         Training Loss: 0.000414         Validation Loss: 0.003594
Epoch: 43         Training Loss: 0.000407         Validation Loss: 0.003820
Epoch: 44         Training Loss: 0.000400         Validation Loss: 0.003544
Validation loss decreased (0.003560 --> 0.003544).  Saving model ...
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)

<ipython-input-14-054014f2de5c> in <module>()
 84 # train the model
 85 model_scratch = train(50, loaders, model_scratch, optimizer_scratch,
---> 86                       criterion_scratch, use_cuda, 'model_scratch.pt')
 87
 88 # load the model that got the best validation accuracy
```

```
    <ipython-input-14-054014f2de5c> in train(n_epochs, loaders, model, optimizer, criterion,
     17         ###################
     18         model.train()
---> 19         for batch_idx, (data, target) in enumerate(loaders['train']):
     20             # move to GPU
     21             if use_cuda:


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in __next__(self)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torch/utils/data/dataloader.py in <listcomp>(.0)
    262         if self.num_workers == 0:  # same-process loading
    263             indices = next(self.sample_iter)  # may raise StopIteration
--> 264             batch = self.collate_fn([self.dataset[i] for i in indices])
    265             if self.pin_memory:
    266                 batch = pin_memory_batch(batch)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
     99         """
    100         path, target = self.samples[index]
--> 101         sample = self.loader(path)
    102         if self.transform is not None:
    103             sample = self.transform(sample)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    145         return accimage_loader(path)
    146     else:
--> 147         return pil_loader(path)
    148
    149


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/datasets/
    128     with open(path, 'rb') as f:
    129         img = Image.open(f)
--> 130         return img.convert('RGB')
    131
    132
```

```
/opt/conda/lib/python3.6/site-packages/PIL/Image.py in convert(self, mode, matrix, dithe
890         """
891
--> 892         self.load()
893
894         if not mode and self.mode == "P":


/opt/conda/lib/python3.6/site-packages/PIL/ImageFile.py in load(self)
204                 prefix = b""
205
--> 206             for decoder_name, extents, offset, args in self.tile:
207                 decoder = Image._getdecoder(self.mode, decoder_name,
208                                             args, self.decoderconfig)


KeyboardInterrupt:
```

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 [15]: 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, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.984491


Test Accuracy: 23% (194/836)
```

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 [16]: ## TODO: Specify data loaders
```

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 [17]: import torchvision.models as models
         import torch.nn as nn
         use_cuda = torch.cuda.is_available()

         ## TODO: Specify model architecture
         # Loading in a pre-trained model
         model_transfer=models.resnet50(pretrained=True)

         # Freeze model weights
         for param in model_transfer.parameters():
             param.requires_grad = False
```

```
        num_features = model_transfer.fc.in_features

        #retrieve the fully connected layer of pre-trained model and replace it with our own li

        model_transfer.fc = nn.Linear(num_features, 133)

        print(model_transfer)

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 94000210.84it/s]
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

```
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
```

```
(0): Bottleneck(
  (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
  (downsample): Sequential(
    (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
    (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
)
(1): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(2): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(3): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
)
(4): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace)
```

```
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace)
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

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

Answer: I loaded pre-trained weights from a network trained on a large dataset. Freezed all the weights in the lower (convolutional) layers: the layers to freeze are adjusted depending on similarity of new task to original dataset. Last fully connected layer of pre-trained network was taken to construct a new last linear layer. Also we could completely replaced the upper layer of the network with a custom Sequential linear layers. To avoid overfitting on this small data set, the weights of the original network was held constant rather than re-training the weights. The chosen arhitecture includes a great deal of Batch Normalizations for the variance considerations. For this reason, I think this architecture is suitable for the current problem.

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 [18]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.005)
```

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 [19]: # train the model
         n_epochs=10

         model_transfer = train(n_epochs, loaders, model_transfer, optimizer_transfer, criterion

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

```
Epoch: 1         Training Loss: 0.000391         Validation Loss: 0.001667
Validation loss decreased (inf --> 0.001667).  Saving model ...
Epoch: 2         Training Loss: 0.000196         Validation Loss: 0.001615
Validation loss decreased (0.001667 --> 0.001615).  Saving model ...
Epoch: 3         Training Loss: 0.000193         Validation Loss: 0.001542
Validation loss decreased (0.001615 --> 0.001542).  Saving model ...
Epoch: 4         Training Loss: 0.000170         Validation Loss: 0.001725
Epoch: 5         Training Loss: 0.000147         Validation Loss: 0.001641
Epoch: 6         Training Loss: 0.000135         Validation Loss: 0.001798
Epoch: 7         Training Loss: 0.000143         Validation Loss: 0.001702
Epoch: 8         Training Loss: 0.000141         Validation Loss: 0.001469
Validation loss decreased (0.001542 --> 0.001469).  Saving model ...
Epoch: 9         Training Loss: 0.000133         Validation Loss: 0.001799
Epoch: 10         Training Loss: 0.000129         Validation Loss: 0.001957
```

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

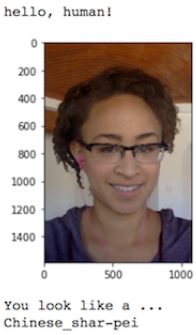
```
Test Loss: 1.336849


Test Accuracy: 80% (675/836)
```

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 [21]: ### 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 = train_file.classes

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img=Image.open(img_path)

             transform=transforms.Compose([ transforms.Resize((224,224)),
                 transforms.ToTensor(),
                 transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
             #from image to tensor
             image_tensor = transform(img).float()
             #adds a dimension with a length of one
             image_ten = image_tensor.unsqueeze_(0)

             if use_cuda:
                 img=image_ten.cuda()

             model_transfer.eval()  #evaluation mode
             output = model_transfer(img)
             index = output.data.cpu().numpy().argmax() #index of the max value (predicted)

             return class_names[index]
```

Step 5: Write your Algorithm



Sample Human Output

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

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
              ## handle cases for a human face, dog, and neither
             img = cv2.imread(img_path)
             ## convert image to RGB color
             rgb_image = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
             # display the image
             plt.imshow(rgb_image)
             plt.show()

             #Find whether the image is a dog a human or none
             if dog_detector(img_path):
                 prediction = predict_breed_transfer(img_path)
                 print("Dogs Detected! That is a dog. Breed:  {0}".format(prediction))
             elif face_detector(img_path) > 0:
                 prediction = predict_breed_transfer(img_path)
                 print("That's a human, but it looks like a {0}".format(prediction))
             else:
                 print("Error! Can't detect anything..")
```

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?

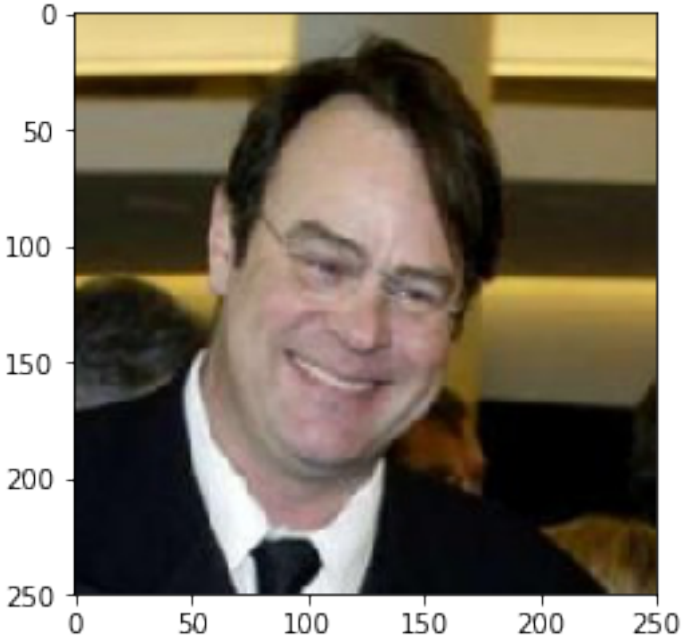
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

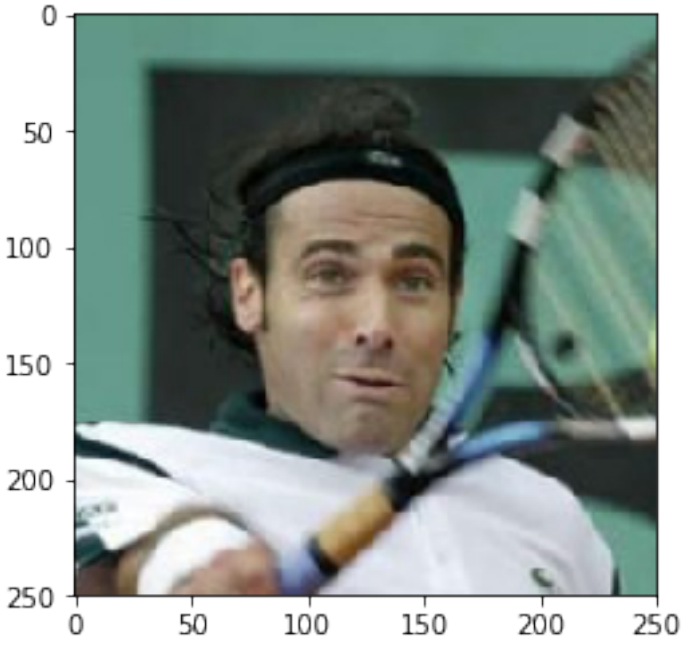
Answer: (Three possible points for improvement) We can apply Bayesian optimization for hyper-parameter tuning and it often selects gradual increase of drop probability from the first convolutional layer down the network. This makes sense because the number of filters also increases, so does the chance of co-adaptation. As a result, the architecture often looks like this: CONV-1: filter=3x3, size=32, dropout between 0.0-0.1 CONV-2: filter=3x3, size=64, dropout between 0.1-0.25 This optimization could increase the performance oand acccuracy of our model.

In [23]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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

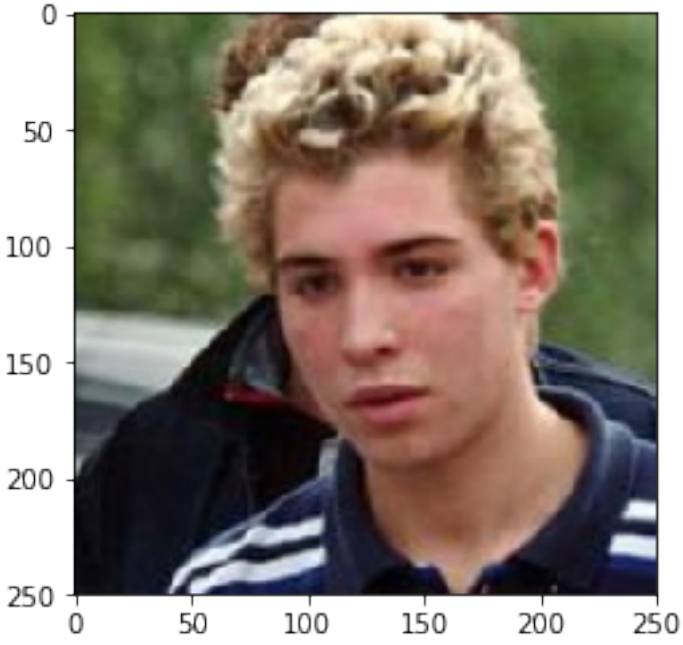


```
/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/transforms/transf
```

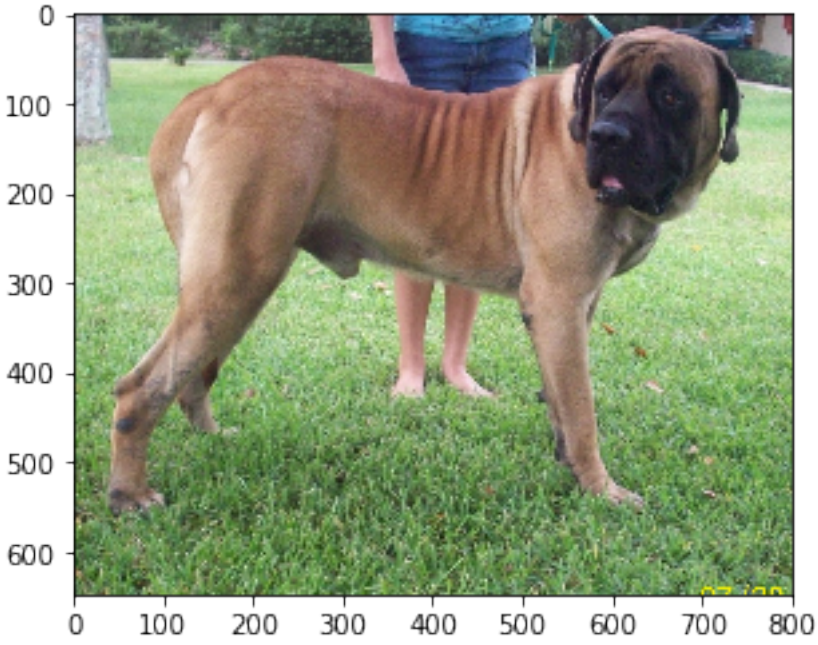
```
That's a human, but it looks like a 111.Norwich_terrier
```



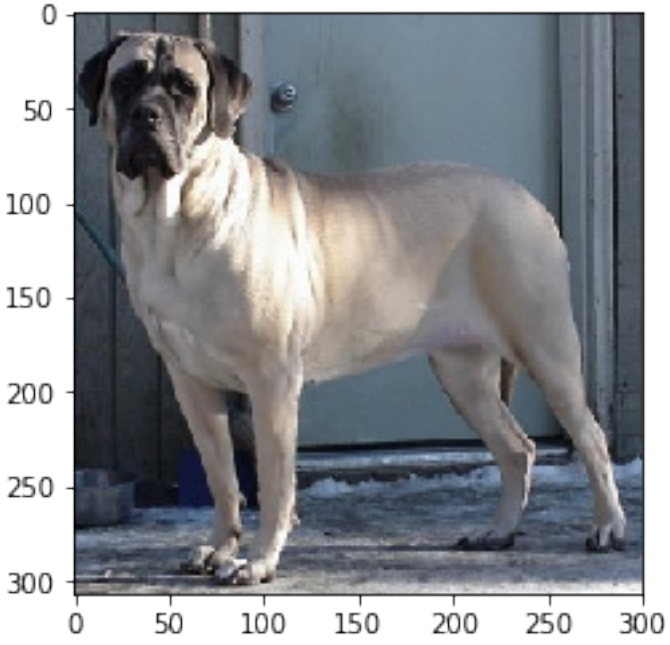

```
That's a human, but it looks like a 059.Doberman_pinscher
```



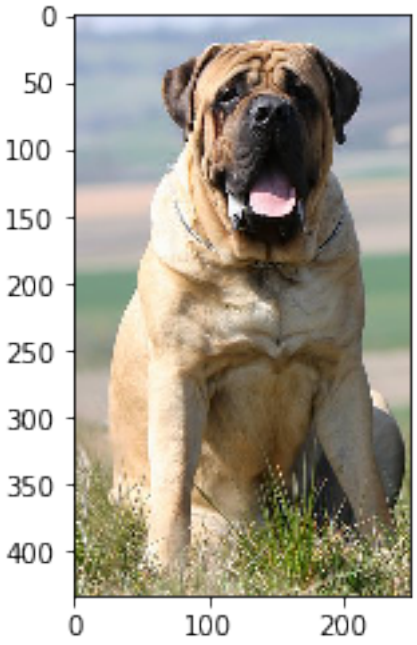
```
That's a human, but it looks like a 009.American_water_spaniel
```




```
Dogs Detected! That is a dog. Breed:  041.Bullmastiff
```



```
Dogs Detected! That is a dog. Breed:  103.Mastiff
```




```
Dogs Detected! That is a dog. Breed:  041.Bullmastiff
```

In [ ]: