dog_app

February 19, 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

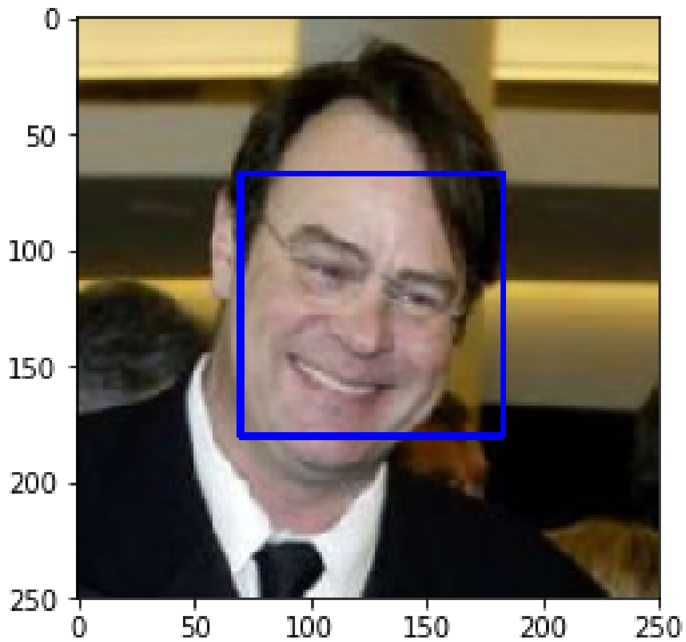
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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: 98% of first 100 images in human files were deteced as human faces. 17% of first 100 images in dog files were detected as human faces.

```
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.
        human_correct = 0
        dog_wrong = 0
        for img in human_files_short:
            if face_detector(img):
                human_correct += 1
        print(human_correct)

        for img in dog_files_short:
            if face_detector(img):
                dog_wrong += 1
        print(dog_wrong)
```

```
98
17
```

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 [00:05<00:00, 99541879.33it/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
from PIL import ImageFile
import torchvision.transforms as transforms
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

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

    Args:
        img_path: path to an image

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

    ## TODO: Complete the function.
    ## Load and pre-process an image from the given img_path
    ## Return the *index* of the predicted class for that image
    image = Image.open(img_path).convert('RGB')
    in_transforms = transforms.Compose([transforms.Resize(256),
                                        transforms.CenterCrop(224),
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.485, 0.456, 0.406),
                                              (0.229, 0.224, 0.225))])
    image = in_transforms(image)
    VGG16.eval()
    image = image.unsqueeze(0)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    image = image.to(device)
    with torch.no_grad():
        log_ps = VGG16(image)
        topk = 1
        topk_probs, topk_idx = torch.exp(log_ps).topk(topk)
        topk_probs, topk_idx = topk_probs.cpu().numpy(), topk_idx.cpu().numpy()
        topk_probs = np.resize(topk_probs,(topk))

    return topk_idx[0][0]
```

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.
    class_id = VGG16_predict(img_path)
    return (class_id in range(151,269))
```

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: 0% of the images in `human_files_short` have a detected dog. 100% of the images in `dog_files_short` have a detected dog

In [9]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
human_wrong = 0
dog_correct = 0
for img in human_files_short:
    if dog_detector(img):
        human_wrong += 1
print(human_wrong)

for img in dog_files_short:
    if dog_detector(img):
        dog_correct += 1
print(dog_correct)
```

```
0
100
```

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

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

         train = np.array(glob("/data/dog_images/train/*/*"))
         train_classes = np.array(glob("/data/dog_images/train/*"))
         valid = np.array(glob("/data/dog_images/valid/*/*"))
         test = np.array(glob("/data/dog_images/test/*/*"))
         print('No. of train images:', len(train))
         print('No. of train image classes:', len(train_classes))
         print('No. of validation images:', len(valid))
         print('No. of test images:', len(test))

         train_dir = "/data/dog_images/train/"
         valid_dir = "/data/dog_images/valid/"
         test_dir = "/data/dog_images/test/"
```

```python
### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
train_transforms = transforms.Compose([transforms.Resize(255),
                                       #transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       #transforms.ColorJitter(brightness = 0.1, hue=0.
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],
                                                            [0.229, 0.224, 0.225])])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])

# TODO: Load the datasets with ImageFolder
train_data = datasets.ImageFolder(train_dir, transform = train_transforms)
valid_data = datasets.ImageFolder(valid_dir, transform = valid_transforms)
test_data = datasets.ImageFolder(test_dir, transform = valid_transforms)

# TODO: Using the image datasets and the trainforms, define the dataloaders
batch_size = 32
train_loader = torch.utils.data.DataLoader(train_data, batch_size =  batch_size, shuffl
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size = batch_size, shuffle
test_loader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, shuffle =

loaders_scratch = {'train' : train_loader, 'valid' : valid_loader, 'test' : test_loader

dataiter = iter(train_loader)
images, target = dataiter.next()
print(images.shape)
```

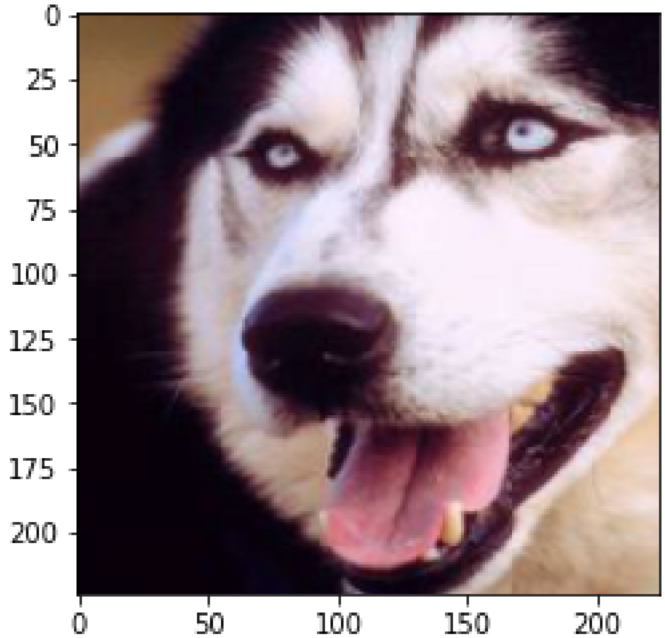
```
No. of train images: 6680
No. of train image classes: 133
No. of validation images: 835
No. of test images: 836
torch.Size([32, 3, 224, 224])
```

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: For the traning set, I am resizing the image first and following that by a crop to create a square without streching. I also use random horizontal flip as a way of augmentation, with will increase randomness and provide more data for training and helps with avoiding overfitting. I could also use things like rotation and color edits. For the final image tensors I chose to apply the standard size and normalization that will be accepted by pretrained models so that I can use the same sets later in my code. For validation and test sets I just applied the resizing and normalization. No augmentation for this data as they will not be used for training.

In [12]:
```
#Check 1 image
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()

def imshow(image, ax=None, title=None):
    """Imshow for Tensor."""
    if ax is None:
        fig, ax = plt.subplots()

    # PyTorch tensors assume the color channel is the first dimension
    # but matplotlib assumes is the third dimension
    image = image.numpy().transpose((1, 2, 0))

    # Undo preprocessing
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean

    # Image needs to be clipped between 0 and 1 or it looks like noise when displayed
    image = np.clip(image, 0, 1)
    ax.imshow(image)

    return ax

imshow(images[1])
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fbc17a792b0>
```



1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [13]: 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
                 # convolutional layer (sees 3x224x224 image tensor)
                 self.conv1 = nn.Conv2d(3, 64, 11, stride =4)
                 self.conv2 = nn.Conv2d(64, 128, 5, padding=2)
                 self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
                 #self.conv4 = nn.Conv2d(256, 512, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 # linear layers
                 self.fc1 = nn.Linear(9216, 1000)
                 self.fc2 = nn.Linear(1000, 500)
                 self.fc3 = nn.Linear(500, 133)
                 # dropout layer (p=0.25)
                 self.dropout = nn.Dropout(0.25)
```

```
    def forward(self, x):
        ## Define forward behavior
        # add sequence of convolutional and max pooling layers
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        #x = self.pool(F.relu(self.conv4(x)))
        # flatten image input
        x = x.view(-1, 9216)
        # add dropout layer
        x = self.dropout(x)
        # add 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add 3rd hidden layer, with relu activation function
        x = self.fc3(x)

        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: My CNN has 3 conv layers, each followed up by a maxpooling layer. Then I have added 3 linear layers reducing the number of node to the final number of classes:

Input image: 3x224x224

In terms of depth (channels), the 3 conv layers increase depth from 3 to 256, to extract more features. I terms of H and W, the first conv layer downsizes by a factor of 4 but the other two layers have stride of 1 and do not downsize (to conserve some info). However, each maxpooling (total 3) downsizes the image by a factor of 2.

When we get to the first linear layer, the flattened image has total of 9216 parameters. This number is reduced to 1000, 500 and 133 through linear layers with dropout between to avoid overfitting. Activation is always done using ReLu except for the classifier layer.

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 [14]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05)
```

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 [15]: 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):
                 print('epoch: ', epoch)
                 # 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
                     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)
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     if batch_idx % 100 == 0:
                         print('Epoch %d, Batch %d loss: %.6f' %
```

```
                            (epoch, batch_idx + 1, 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 = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)

            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('valid loss decreased from ', valid_loss_min, ' to ', valid_loss)
                torch.save(model.state_dict(), save_path)
                valid_loss_min = valid_loss
        # return trained model
        return model

    import workspace_utils
    from workspace_utils import active_session

    with active_session():
        model_scratch = train(35, loaders_scratch, model_scratch, optimizer_scratch,
                              criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
epoch:  1
Epoch 1, Batch 1 loss: 4.895534
Epoch 1, Batch 101 loss: 4.886809
Epoch 1, Batch 201 loss: 4.882680
Epoch: 1         Training Loss: 4.881288         Validation Loss: 4.875809
valid loss decreased from  inf  to  tensor(4.8758, device='cuda:0')
epoch:  2
Epoch 2, Batch 1 loss: 5.018872
Epoch 2, Batch 101 loss: 4.849266
```

```
Epoch 2, Batch 201 loss: 4.831444
Epoch: 2        Training Loss: 4.829422         Validation Loss: 4.767841
valid loss decreased from  tensor(4.8758, device='cuda:0')  to  tensor(4.7678, device='cuda:0')
epoch:  3
Epoch 3, Batch 1 loss: 4.622593
Epoch 3, Batch 101 loss: 4.762261
Epoch 3, Batch 201 loss: 4.751817
Epoch: 3        Training Loss: 4.747633         Validation Loss: 4.629561
valid loss decreased from  tensor(4.7678, device='cuda:0')  to  tensor(4.6296, device='cuda:0')
epoch:  4
Epoch 4, Batch 1 loss: 4.819359
Epoch 4, Batch 101 loss: 4.653384
Epoch 4, Batch 201 loss: 4.645288
Epoch: 4        Training Loss: 4.643981         Validation Loss: 4.527909
valid loss decreased from  tensor(4.6296, device='cuda:0')  to  tensor(4.5279, device='cuda:0')
epoch:  5
Epoch 5, Batch 1 loss: 4.451943
Epoch 5, Batch 101 loss: 4.591776
Epoch 5, Batch 201 loss: 4.586250
Epoch: 5        Training Loss: 4.586477         Validation Loss: 4.465522
valid loss decreased from  tensor(4.5279, device='cuda:0')  to  tensor(4.4655, device='cuda:0')
epoch:  6
Epoch 6, Batch 1 loss: 4.704865
Epoch 6, Batch 101 loss: 4.553176
Epoch 6, Batch 201 loss: 4.546210
Epoch: 6        Training Loss: 4.542343         Validation Loss: 4.363665
valid loss decreased from  tensor(4.4655, device='cuda:0')  to  tensor(4.3637, device='cuda:0')
epoch:  7
Epoch 7, Batch 1 loss: 4.737431
Epoch 7, Batch 101 loss: 4.498604
Epoch 7, Batch 201 loss: 4.497906
Epoch: 7        Training Loss: 4.495812         Validation Loss: 4.319691
valid loss decreased from  tensor(4.3637, device='cuda:0')  to  tensor(4.3197, device='cuda:0')
epoch:  8
Epoch 8, Batch 1 loss: 4.340785
Epoch 8, Batch 101 loss: 4.451553
Epoch 8, Batch 201 loss: 4.451702
Epoch: 8        Training Loss: 4.448515         Validation Loss: 4.274125
valid loss decreased from  tensor(4.3197, device='cuda:0')  to  tensor(4.2741, device='cuda:0')
epoch:  9
Epoch 9, Batch 1 loss: 4.197806
Epoch 9, Batch 101 loss: 4.381349
Epoch 9, Batch 201 loss: 4.397630
Epoch: 9        Training Loss: 4.398832         Validation Loss: 4.277610
epoch:  10
Epoch 10, Batch 1 loss: 4.087823
Epoch 10, Batch 101 loss: 4.374114
Epoch 10, Batch 201 loss: 4.358187
```

```
Epoch: 10         Training Loss: 4.364063         Validation Loss: 4.246711
valid loss decreased from  tensor(4.2741, device='cuda:0')  to  tensor(4.2467, device='cuda:0')
epoch:  11
Epoch 11, Batch 1 loss: 4.481061
Epoch 11, Batch 101 loss: 4.305691
Epoch 11, Batch 201 loss: 4.306990
Epoch: 11         Training Loss: 4.308173         Validation Loss: 4.070160
valid loss decreased from  tensor(4.2467, device='cuda:0')  to  tensor(4.0702, device='cuda:0')
epoch:  12
Epoch 12, Batch 1 loss: 4.275544
Epoch 12, Batch 101 loss: 4.265171
Epoch 12, Batch 201 loss: 4.267090
Epoch: 12         Training Loss: 4.267263         Validation Loss: 4.057867
valid loss decreased from  tensor(4.0702, device='cuda:0')  to  tensor(4.0579, device='cuda:0')
epoch:  13
Epoch 13, Batch 1 loss: 3.796736
Epoch 13, Batch 101 loss: 4.230784
Epoch 13, Batch 201 loss: 4.238123
Epoch: 13         Training Loss: 4.241964         Validation Loss: 4.281120
epoch:  14
Epoch 14, Batch 1 loss: 4.416212
Epoch 14, Batch 101 loss: 4.164008
Epoch 14, Batch 201 loss: 4.183263
Epoch: 14         Training Loss: 4.186869         Validation Loss: 3.986063
valid loss decreased from  tensor(4.0579, device='cuda:0')  to  tensor(3.9861, device='cuda:0')
epoch:  15
Epoch 15, Batch 1 loss: 3.980146
Epoch 15, Batch 101 loss: 4.180257
Epoch 15, Batch 201 loss: 4.161677
Epoch: 15         Training Loss: 4.163218         Validation Loss: 3.931766
valid loss decreased from  tensor(3.9861, device='cuda:0')  to  tensor(3.9318, device='cuda:0')
epoch:  16
Epoch 16, Batch 1 loss: 4.311951
Epoch 16, Batch 101 loss: 4.112730
Epoch 16, Batch 201 loss: 4.113408
Epoch: 16         Training Loss: 4.110558         Validation Loss: 3.901447
valid loss decreased from  tensor(3.9318, device='cuda:0')  to  tensor(3.9014, device='cuda:0')
epoch:  17
Epoch 17, Batch 1 loss: 4.280809
Epoch 17, Batch 101 loss: 4.096035
Epoch 17, Batch 201 loss: 4.080647
Epoch: 17         Training Loss: 4.082810         Validation Loss: 3.830290
valid loss decreased from  tensor(3.9014, device='cuda:0')  to  tensor(3.8303, device='cuda:0')
epoch:  18
Epoch 18, Batch 1 loss: 4.492016
Epoch 18, Batch 101 loss: 4.013468
Epoch 18, Batch 201 loss: 4.014163
Epoch: 18         Training Loss: 4.013809         Validation Loss: 3.862594
```

```
epoch:  19
Epoch 19, Batch 1 loss: 4.192379
Epoch 19, Batch 101 loss: 3.984332
Epoch 19, Batch 201 loss: 3.988458
Epoch: 19         Training Loss: 3.982227         Validation Loss: 4.061631
epoch:  20
Epoch 20, Batch 1 loss: 4.285929
Epoch 20, Batch 101 loss: 3.966508
Epoch 20, Batch 201 loss: 3.942158
Epoch: 20         Training Loss: 3.940603         Validation Loss: 3.823156
valid loss decreased from  tensor(3.8303, device='cuda:0')  to  tensor(3.8232, device='cuda:0')
epoch:  21
Epoch 21, Batch 1 loss: 3.853507
Epoch 21, Batch 101 loss: 3.873794
Epoch 21, Batch 201 loss: 3.882252
Epoch: 21         Training Loss: 3.883894         Validation Loss: 3.651997
valid loss decreased from  tensor(3.8232, device='cuda:0')  to  tensor(3.6520, device='cuda:0')
epoch:  22
Epoch 22, Batch 1 loss: 4.039166
Epoch 22, Batch 101 loss: 3.874387
Epoch 22, Batch 201 loss: 3.863183
Epoch: 22         Training Loss: 3.858973         Validation Loss: 4.038293
epoch:  23
Epoch 23, Batch 1 loss: 3.858865
Epoch 23, Batch 101 loss: 3.847278
Epoch 23, Batch 201 loss: 3.830787
Epoch: 23         Training Loss: 3.827644         Validation Loss: 3.675496
epoch:  24
Epoch 24, Batch 1 loss: 3.452949
Epoch 24, Batch 101 loss: 3.785282
Epoch 24, Batch 201 loss: 3.790619
Epoch: 24         Training Loss: 3.787759         Validation Loss: 3.774135
epoch:  25
Epoch 25, Batch 1 loss: 3.306952
Epoch 25, Batch 101 loss: 3.715002
Epoch 25, Batch 201 loss: 3.740900
Epoch: 25         Training Loss: 3.744439         Validation Loss: 3.585464
valid loss decreased from  tensor(3.6520, device='cuda:0')  to  tensor(3.5855, device='cuda:0')
epoch:  26
Epoch 26, Batch 1 loss: 3.692962
Epoch 26, Batch 101 loss: 3.681363
Epoch 26, Batch 201 loss: 3.706226
Epoch: 26         Training Loss: 3.707630         Validation Loss: 3.738204
epoch:  27
Epoch 27, Batch 1 loss: 4.490313
Epoch 27, Batch 101 loss: 3.682530
Epoch 27, Batch 201 loss: 3.661528
Epoch: 27         Training Loss: 3.660451         Validation Loss: 3.475726
```

```
valid loss decreased from  tensor(3.5855, device='cuda:0')  to  tensor(3.4757, device='cuda:0')
epoch:  28
Epoch 28, Batch 1 loss: 3.815551
Epoch 28, Batch 101 loss: 3.657420
Epoch 28, Batch 201 loss: 3.631387
Epoch: 28         Training Loss: 3.631820         Validation Loss: 3.458314
valid loss decreased from  tensor(3.4757, device='cuda:0')  to  tensor(3.4583, device='cuda:0')
epoch:  29
Epoch 29, Batch 1 loss: 3.529519
Epoch 29, Batch 101 loss: 3.590559
Epoch 29, Batch 201 loss: 3.599618
Epoch: 29         Training Loss: 3.602759         Validation Loss: 3.403983
valid loss decreased from  tensor(3.4583, device='cuda:0')  to  tensor(3.4040, device='cuda:0')
epoch:  30
Epoch 30, Batch 1 loss: 3.212768
Epoch 30, Batch 101 loss: 3.536619
Epoch 30, Batch 201 loss: 3.544228
Epoch: 30         Training Loss: 3.540731         Validation Loss: 3.401852
valid loss decreased from  tensor(3.4040, device='cuda:0')  to  tensor(3.4019, device='cuda:0')
epoch:  31
Epoch 31, Batch 1 loss: 3.826680
Epoch 31, Batch 101 loss: 3.533992
Epoch 31, Batch 201 loss: 3.522552
Epoch: 31         Training Loss: 3.529032         Validation Loss: 3.274340
valid loss decreased from  tensor(3.4019, device='cuda:0')  to  tensor(3.2743, device='cuda:0')
epoch:  32
Epoch 32, Batch 1 loss: 3.281089
Epoch 32, Batch 101 loss: 3.491206
Epoch 32, Batch 201 loss: 3.492147
Epoch: 32         Training Loss: 3.493273         Validation Loss: 3.247547
valid loss decreased from  tensor(3.2743, device='cuda:0')  to  tensor(3.2475, device='cuda:0')
epoch:  33
Epoch 33, Batch 1 loss: 3.879552
Epoch 33, Batch 101 loss: 3.423073
Epoch 33, Batch 201 loss: 3.439435
Epoch: 33         Training Loss: 3.439326         Validation Loss: 3.310119
epoch:  34
Epoch 34, Batch 1 loss: 3.094059
Epoch 34, Batch 101 loss: 3.413871
Epoch 34, Batch 201 loss: 3.428639
Epoch: 34         Training Loss: 3.430513         Validation Loss: 3.223046
valid loss decreased from  tensor(3.2475, device='cuda:0')  to  tensor(3.2230, device='cuda:0')
epoch:  35
Epoch 35, Batch 1 loss: 3.244784
Epoch 35, Batch 101 loss: 3.392665
Epoch 35, Batch 201 loss: 3.367958
Epoch: 35         Training Loss: 3.370670         Validation Loss: 3.245029
```

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 [17]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders['test']):
                 # move to GPU
                 if use_cuda:
                     data, target = data.cuda(), target.cuda()
                 # forward pass: compute predicted outputs by passing inputs to the model
                 output = model(data)
                 # calculate the loss
                 loss = criterion(output, target)
                 # update average test loss
                 test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
                 # convert output probabilities to predicted class
                 pred = output.data.max(1, keepdim=True)[1]
                 # compare predictions to true label
                 correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
                 total += data.size(0)

             print('Test Loss: {:.6f}\n'.format(test_loss))

             print('\nTest Accuracy: %2d%% (%2d/%2d)' % (
                 100. * correct / total, correct, total))

         # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.229065


Test Accuracy: 20% (174/836)
```

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 [18]:
```
## TODO: Specify data loaders
loaders_transfer = loaders_scratch.copy()
```

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 [19]:
```
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.features.parameters():
    param.requires_grad = False

num_inputs = model_transfer.classifier[6].in_features
final_layer = nn.Linear(num_inputs, 133)
model_transfer.classifier[6] = final_layer

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

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 have used the VGG16 pretrained model to be used in transfer learning. I have only pulled out the final classifier layer and changed it to a dog breed classifier with 133 classes. I have freezed all the pretrained features and will only train the classifier on the new data.

apart from exceptional performance on the Imagenet data set, VGG16 was shown to generalize very well for other datasets. As seen above, VGG16 did a very good job on the task of detecting dogs. So it works well with the pretrained weights for general features. I can just train the classifier with data on various dog breeds to get reasonable accuracy. Offcourse finetuning can be used to improve results.

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 [20]:
```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

In [21]:
```
# train the model
with active_session():
    model_transfer = train(35, loaders_transfer, model_transfer, optimizer_transfer,
                           criterion_transfer, use_cuda, 'model_transfer.pt')
# load the model that got the best validation accuracy
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
epoch:  1
Epoch 1, Batch 1 loss: 5.154062
Epoch 1, Batch 101 loss: 4.759295
Epoch 1, Batch 201 loss: 4.515614
Epoch: 1         Training Loss: 4.496827         Validation Loss: 3.628063
valid loss decreased from  inf  to  tensor(3.6281, device='cuda:0')
epoch:  2
Epoch 2, Batch 1 loss: 4.142445
Epoch 2, Batch 101 loss: 3.711993
Epoch 2, Batch 201 loss: 3.479299
Epoch: 2         Training Loss: 3.462790         Validation Loss: 2.303020
valid loss decreased from  tensor(3.6281, device='cuda:0')  to  tensor(2.3030, device='cuda:0')
epoch:  3
Epoch 3, Batch 1 loss: 2.801722
Epoch 3, Batch 101 loss: 2.736456
Epoch 3, Batch 201 loss: 2.544197
Epoch: 3         Training Loss: 2.528568         Validation Loss: 1.387266
valid loss decreased from  tensor(2.3030, device='cuda:0')  to  tensor(1.3873, device='cuda:0')
epoch:  4
Epoch 4, Batch 1 loss: 2.108459
Epoch 4, Batch 101 loss: 2.054661
Epoch 4, Batch 201 loss: 1.949820
Epoch: 4         Training Loss: 1.942327         Validation Loss: 0.950864
valid loss decreased from  tensor(1.3873, device='cuda:0')  to  tensor(0.9509, device='cuda:0')
epoch:  5
Epoch 5, Batch 1 loss: 1.922371
Epoch 5, Batch 101 loss: 1.708559
Epoch 5, Batch 201 loss: 1.651476
Epoch: 5         Training Loss: 1.651702         Validation Loss: 0.731224
valid loss decreased from  tensor(0.9509, device='cuda:0')  to  tensor(0.7312, device='cuda:0')
epoch:  6
Epoch 6, Batch 1 loss: 0.929842
Epoch 6, Batch 101 loss: 1.482562
Epoch 6, Batch 201 loss: 1.448082
Epoch: 6         Training Loss: 1.444872         Validation Loss: 0.611922
valid loss decreased from  tensor(0.7312, device='cuda:0')  to  tensor(0.6119, device='cuda:0')
epoch:  7
```

```
Epoch 7, Batch 1 loss: 1.390367
Epoch 7, Batch 101 loss: 1.358692
Epoch 7, Batch 201 loss: 1.329249
Epoch: 7         Training Loss: 1.330260         Validation Loss: 0.546804
valid loss decreased from  tensor(0.6119, device='cuda:0')  to  tensor(0.5468, device='cuda:0')
epoch:  8
Epoch 8, Batch 1 loss: 1.230391
Epoch 8, Batch 101 loss: 1.263984
Epoch 8, Batch 201 loss: 1.246097
Epoch: 8         Training Loss: 1.247470         Validation Loss: 0.503688
valid loss decreased from  tensor(0.5468, device='cuda:0')  to  tensor(0.5037, device='cuda:0')
epoch:  9
Epoch 9, Batch 1 loss: 1.018873
Epoch 9, Batch 101 loss: 1.248821
Epoch 9, Batch 201 loss: 1.217937
Epoch: 9         Training Loss: 1.214726         Validation Loss: 0.476963
valid loss decreased from  tensor(0.5037, device='cuda:0')  to  tensor(0.4770, device='cuda:0')
epoch:  10
Epoch 10, Batch 1 loss: 1.406891
Epoch 10, Batch 101 loss: 1.135654
Epoch 10, Batch 201 loss: 1.135895
Epoch: 10         Training Loss: 1.139935         Validation Loss: 0.465071
valid loss decreased from  tensor(0.4770, device='cuda:0')  to  tensor(0.4651, device='cuda:0')
epoch:  11
Epoch 11, Batch 1 loss: 1.653991
Epoch 11, Batch 101 loss: 1.109479
Epoch 11, Batch 201 loss: 1.093493
Epoch: 11         Training Loss: 1.091271         Validation Loss: 0.442676
valid loss decreased from  tensor(0.4651, device='cuda:0')  to  tensor(0.4427, device='cuda:0')
epoch:  12
Epoch 12, Batch 1 loss: 1.111418
Epoch 12, Batch 101 loss: 1.095998
Epoch 12, Batch 201 loss: 1.080114
Epoch: 12         Training Loss: 1.082043         Validation Loss: 0.435828
valid loss decreased from  tensor(0.4427, device='cuda:0')  to  tensor(0.4358, device='cuda:0')
epoch:  13
Epoch 13, Batch 1 loss: 0.956760
Epoch 13, Batch 101 loss: 1.046680
Epoch 13, Batch 201 loss: 1.039564
Epoch: 13         Training Loss: 1.036380         Validation Loss: 0.410687
valid loss decreased from  tensor(0.4358, device='cuda:0')  to  tensor(0.4107, device='cuda:0')
epoch:  14
Epoch 14, Batch 1 loss: 1.063223
Epoch 14, Batch 101 loss: 1.025508
Epoch 14, Batch 201 loss: 1.023633
Epoch: 14         Training Loss: 1.021780         Validation Loss: 0.410755
epoch:  15
Epoch 15, Batch 1 loss: 0.687074
```

```
Epoch 15, Batch 101 loss: 1.033625
Epoch 15, Batch 201 loss: 1.008450
Epoch: 15         Training Loss: 1.004854         Validation Loss: 0.408880
valid loss decreased from  tensor(0.4107, device='cuda:0')  to  tensor(0.4089, device='cuda:0')
epoch:  16
Epoch 16, Batch 1 loss: 1.124287
Epoch 16, Batch 101 loss: 1.014286
Epoch 16, Batch 201 loss: 0.985776
Epoch: 16         Training Loss: 0.985522         Validation Loss: 0.387051
valid loss decreased from  tensor(0.4089, device='cuda:0')  to  tensor(0.3871, device='cuda:0')
epoch:  17
Epoch 17, Batch 1 loss: 1.203688
Epoch 17, Batch 101 loss: 0.963714
Epoch 17, Batch 201 loss: 0.967229
Epoch: 17         Training Loss: 0.961828         Validation Loss: 0.388755
epoch:  18
Epoch 18, Batch 1 loss: 0.880023
Epoch 18, Batch 101 loss: 0.927227
Epoch 18, Batch 201 loss: 0.944797
Epoch: 18         Training Loss: 0.946302         Validation Loss: 0.377274
valid loss decreased from  tensor(0.3871, device='cuda:0')  to  tensor(0.3773, device='cuda:0')
epoch:  19
Epoch 19, Batch 1 loss: 0.657808
Epoch 19, Batch 101 loss: 0.959977
Epoch 19, Batch 201 loss: 0.944279
Epoch: 19         Training Loss: 0.936968         Validation Loss: 0.372191
valid loss decreased from  tensor(0.3773, device='cuda:0')  to  tensor(0.3722, device='cuda:0')
epoch:  20
Epoch 20, Batch 1 loss: 0.947421
Epoch 20, Batch 101 loss: 0.877303
Epoch 20, Batch 201 loss: 0.900056
Epoch: 20         Training Loss: 0.897550         Validation Loss: 0.374983
epoch:  21
Epoch 21, Batch 1 loss: 0.928322
Epoch 21, Batch 101 loss: 0.911745
Epoch 21, Batch 201 loss: 0.905282
Epoch: 21         Training Loss: 0.905903         Validation Loss: 0.361394
valid loss decreased from  tensor(0.3722, device='cuda:0')  to  tensor(0.3614, device='cuda:0')
epoch:  22
Epoch 22, Batch 1 loss: 0.741127
Epoch 22, Batch 101 loss: 0.873779
Epoch 22, Batch 201 loss: 0.889729
Epoch: 22         Training Loss: 0.884954         Validation Loss: 0.356964
valid loss decreased from  tensor(0.3614, device='cuda:0')  to  tensor(0.3570, device='cuda:0')
epoch:  23
Epoch 23, Batch 1 loss: 1.253862
Epoch 23, Batch 101 loss: 0.931090
Epoch 23, Batch 201 loss: 0.915739
```

```
Epoch: 23         Training Loss: 0.916535         Validation Loss: 0.362401
epoch:  24
Epoch 24, Batch 1 loss: 1.102068
Epoch 24, Batch 101 loss: 0.889769
Epoch 24, Batch 201 loss: 0.897361
Epoch: 24         Training Loss: 0.897752         Validation Loss: 0.350188
valid loss decreased from  tensor(0.3570, device='cuda:0')  to  tensor(0.3502, device='cuda:0')
epoch:  25
Epoch 25, Batch 1 loss: 0.850676
Epoch 25, Batch 101 loss: 0.849013
Epoch 25, Batch 201 loss: 0.858508
Epoch: 25         Training Loss: 0.861203         Validation Loss: 0.353887
epoch:  26
Epoch 26, Batch 1 loss: 0.826591
Epoch 26, Batch 101 loss: 0.846415
Epoch 26, Batch 201 loss: 0.857488
Epoch: 26         Training Loss: 0.862118         Validation Loss: 0.362344
epoch:  27
Epoch 27, Batch 1 loss: 0.708086
Epoch 27, Batch 101 loss: 0.830226
Epoch 27, Batch 201 loss: 0.850474
Epoch: 27         Training Loss: 0.851958         Validation Loss: 0.346242
valid loss decreased from  tensor(0.3502, device='cuda:0')  to  tensor(0.3462, device='cuda:0')
epoch:  28
Epoch 28, Batch 1 loss: 0.847862
Epoch 28, Batch 101 loss: 0.884771
Epoch 28, Batch 201 loss: 0.861671
Epoch: 28         Training Loss: 0.862534         Validation Loss: 0.340620
valid loss decreased from  tensor(0.3462, device='cuda:0')  to  tensor(0.3406, device='cuda:0')
epoch:  29
Epoch 29, Batch 1 loss: 0.747054
Epoch 29, Batch 101 loss: 0.821552
Epoch 29, Batch 201 loss: 0.829327
Epoch: 29         Training Loss: 0.831050         Validation Loss: 0.341578
epoch:  30
Epoch 30, Batch 1 loss: 0.815440
Epoch 30, Batch 101 loss: 0.841008
Epoch 30, Batch 201 loss: 0.839986
Epoch: 30         Training Loss: 0.833142         Validation Loss: 0.340602
valid loss decreased from  tensor(0.3406, device='cuda:0')  to  tensor(0.3406, device='cuda:0')
epoch:  31
Epoch 31, Batch 1 loss: 0.610570
Epoch 31, Batch 101 loss: 0.855719
Epoch 31, Batch 201 loss: 0.835067
Epoch: 31         Training Loss: 0.837529         Validation Loss: 0.335563
valid loss decreased from  tensor(0.3406, device='cuda:0')  to  tensor(0.3356, device='cuda:0')
epoch:  32
Epoch 32, Batch 1 loss: 0.958753
```

```
Epoch 32, Batch 101 loss: 0.810672
Epoch 32, Batch 201 loss: 0.824537
Epoch: 32 	Training Loss: 0.827779 	Validation Loss: 0.330760
valid loss decreased from  tensor(0.3356, device='cuda:0')  to  tensor(0.3308, device='cuda:0')
epoch:  33
Epoch 33, Batch 1 loss: 0.672585
Epoch 33, Batch 101 loss: 0.809552
Epoch 33, Batch 201 loss: 0.816705
Epoch: 33 	Training Loss: 0.815535 	Validation Loss: 0.332397
epoch:  34
Epoch 34, Batch 1 loss: 0.468413
Epoch 34, Batch 101 loss: 0.814834
Epoch 34, Batch 201 loss: 0.813401
Epoch: 34 	Training Loss: 0.815359 	Validation Loss: 0.334733
epoch:  35
Epoch 35, Batch 1 loss: 1.170538
Epoch 35, Batch 101 loss: 0.838508
Epoch 35, Batch 201 loss: 0.836722
Epoch: 35 	Training Loss: 0.833872 	Validation Loss: 0.342729
```

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

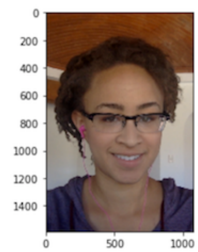
```
Test Loss: 0.381477


Test Accuracy: 87% (732/836)
```

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 [26]: ### 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'].dataset

         print(class_names[:5])

         def predict_breed_transfer(img_path, model,class_names):
```

```
hello, human!
```



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

Sample Human Output

```
        # load the image and return the predicted breed
        image = Image.open(img_path).convert('RGB')
        in_transforms = transforms.Compose([transforms.Resize(256),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize((0.485, 0.456, 0.406),
                                                (0.229, 0.224, 0.225))])
        image = in_transforms(image)
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        image = image.to(device)
        model = model.to(device)
        model.eval()
        image = image.unsqueeze(0)
        with torch.no_grad():
            idx = torch.argmax(model(image))
        return class_names[idx]
```

```
['Affenpinscher', 'Afghan hound', 'Airedale terrier', 'Akita', 'Alaskan malamute']
```

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

```
def run_app(img_path):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path):
        breed = predict_breed_transfer(img_path, model_transfer, class_names)
        print("Found a dog!\nSeems like a ", breed)
    elif face_detector(img_path) > 0:
        breed = predict_breed_transfer(img_path, model_transfer, class_names)
        print("Hi, human!\nIn a dog's world you look like a ", breed)
    else:
        print("Can't find a dog or human... boring picture maybe?")
    ## handle cases for a human face, dog, and neither
```

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) The model is working reasonably well but can always be improved by:

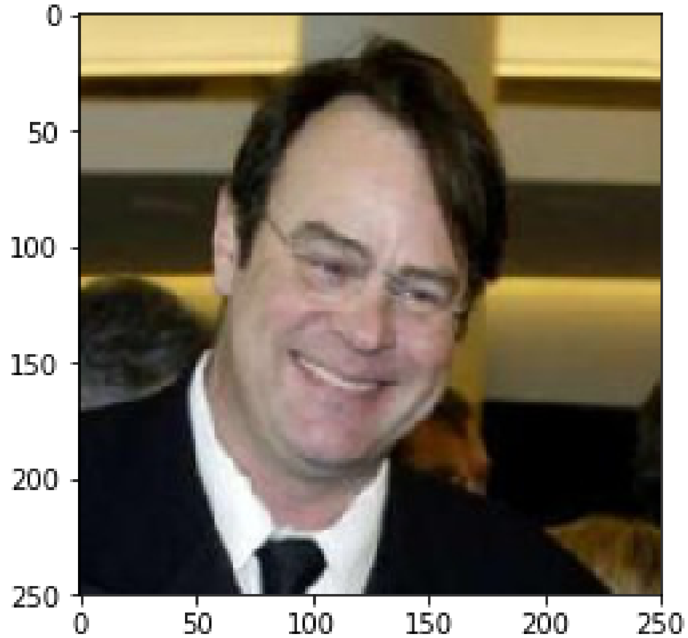
1- for our classifier layer, we can tune the parameters like batch sizes, learning rate (with schedulers), initial weights and try different optimizers to improve.

2- We can release a few of the layers before our classifier to fine-tune the weights.

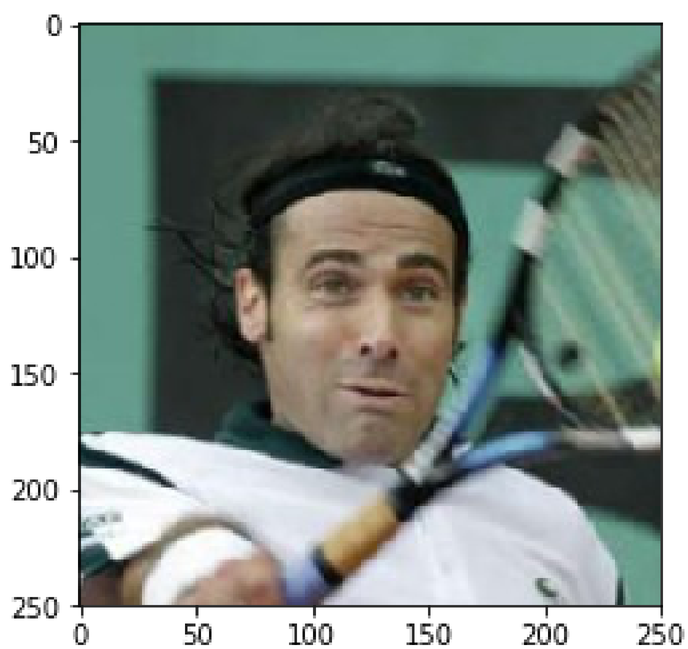
3- We can add additional dog images to our training set, and use more data augmentation techniques.

```
In [36]: ## 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[6], dog_files[1000], dog_files[5000])
             print('----------------\nTesting the model for file: ', file)
             run_app(file)
```

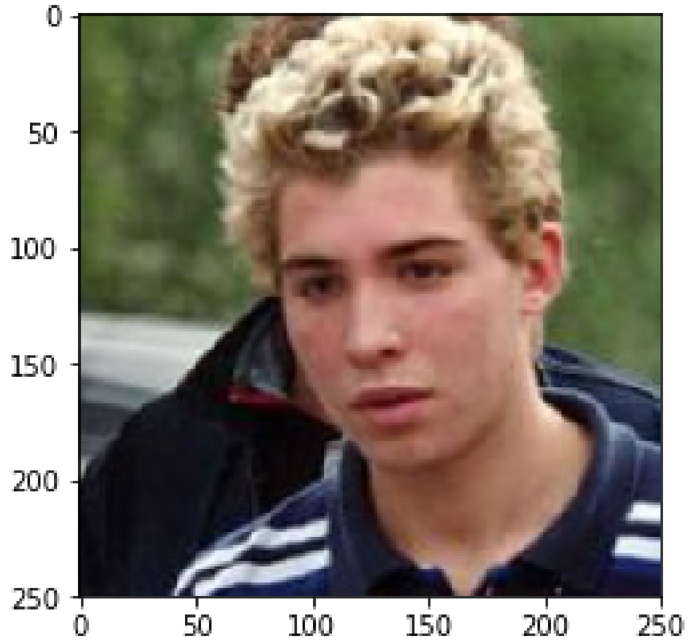
```
---------------
Testing the model for file:  /data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg
```



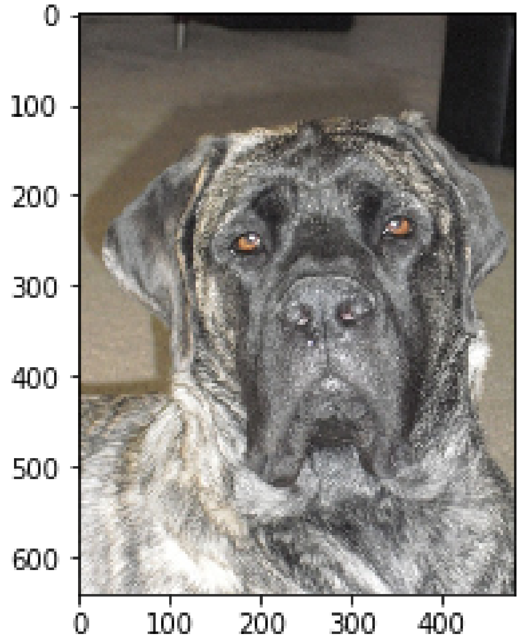

```
Hi, human!
In a dog's world you look like a  Chihuahua
---------------
Testing the model for file:  /data/lfw/Alex_Corretja/Alex_Corretja_0001.jpg
```



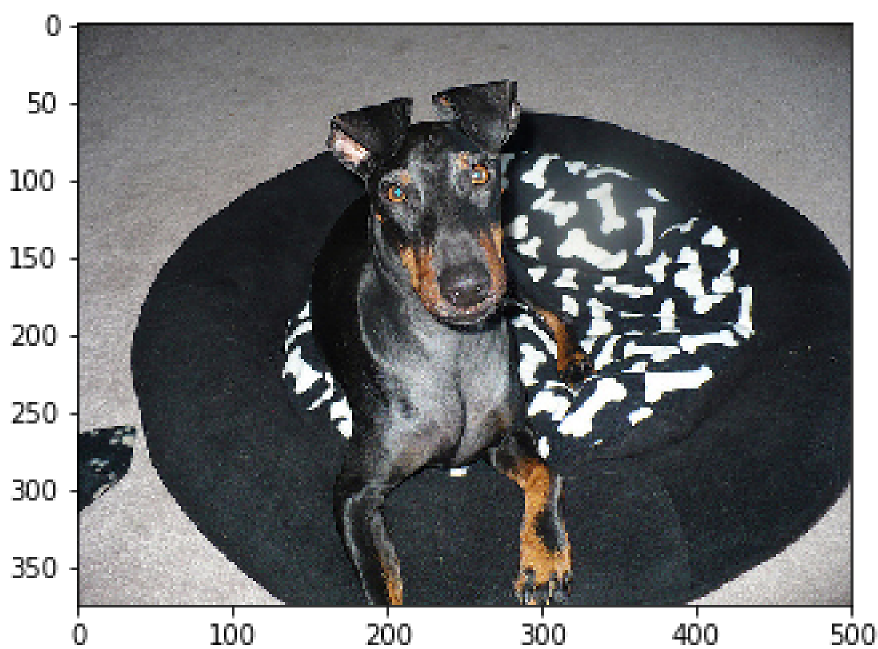
```
Hi, human!
In a dog's world you look like a  Australian shepherd
---------------
Testing the model for file:  /data/lfw/Daniele_Bergamin/Daniele_Bergamin_0001.jpg
```



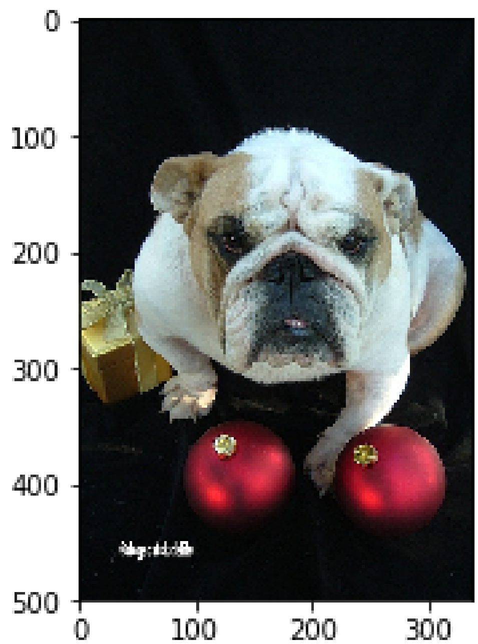

```
Hi, human!
In a dog's world you look like a  Afghan hound
---------------
Testing the model for file:  /data/dog_images/train/103.Mastiff/Mastiff_06845.jpg
```




```
Found a dog!
Seems like a  Mastiff
---------------
Testing the model for file:  /data/dog_images/train/102.Manchester_terrier/Manchester_terrier_06
```



```
Found a dog!
Seems like a  Manchester terrier
---------------
Testing the model for file:  /data/dog_images/train/040.Bulldog/Bulldog_02823.jpg
```



```
Found a dog!
Seems like a  Bulldog
```

In [ ]: