dog_app

April 25, 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]: !pip install 'pillow==5.4.1'
```

```
Collecting pillow==5.4.1
  Downloading https://files.pythonhosted.org/packages/85/5e/e91792f198bbc5a0d7d3055ad552bc406294
    100% || 2.0MB 269kB/s eta 0:00:01
Installing collected packages: pillow
  Found existing installation: Pillow 6.0.0
    Uninstalling Pillow-6.0.0:
      Successfully uninstalled Pillow-6.0.0
Successfully installed pillow-5.4.1
You are using pip version 9.0.1, however version 19.1 is available.You should consider upgrading
```

```
In [2]: import PIL
        PIL.__version__
```

```
Out[2]: '5.4.1'
```

```
In [3]: 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

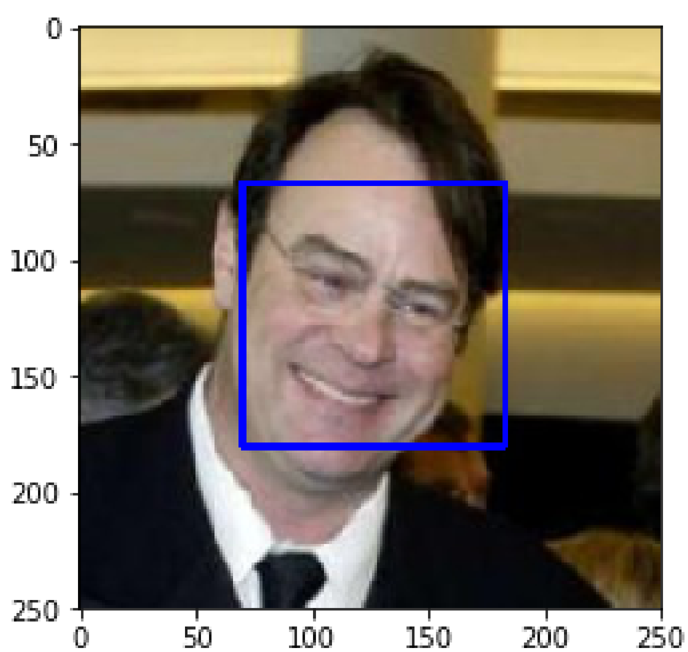
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [4]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
```

```python
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 [5]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

```
In [6]: 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_detected = 0.0
        dog_detected = 0.0

        number_files = len(human_files_short)

        for i in range (0, number_files):
            human_path = human_files_short[i]
```

```
        dog_path = dog_files_short[i]
        if face_detector(human_path) == True:
            human_detected += 1
        if face_detector(dog_path) == True:
            dog_detected += 1

    print('The percentage of the detected face - Humans:{0:.0%}'.format(human_detected / num
    print('The percentage of the detected face - Dogs:{0:.0%}'.format(dog_detected / number_
```

```
The percentage of the detected face - Humans:98%
The percentage of the detected face - Dogs: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 [7]:
```
### (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 [8]:
```
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()
VGG16
print("cuda:",use_cuda)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 109053125.90it/s]
```

```
cuda: True
```

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

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

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

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

In [9]:
```python
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
    img = Image.open(img_path)




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

```
        # Insert the new axis at index 0 i.e. in front of the other axes/dims.
    img = img.unsqueeze(0)

    # Now that we have preprocessed our img, we need to convert it into a
    # Variable
    #img = Variable(img)
    if use_cuda:
        img = img.cuda()


    VGG16.eval()
    # Returns a Tensor of shape
    prediction = VGG16(img)
    prediction = prediction.data.argmax().cpu().numpy()

    ## Return the *index* of the predicted class for that image




    return prediction  # 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 [10]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             class_index = VGG16_predict(img_path)
             if class_index >= 151 and class_index <= 268:
                 return True
             else:
                 return 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 [11]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_detected = 0.0
         dog_detected = 0.0

         num_files = len(human_files_short)

         for i in range(0, num_files):
             human_path = human_files_short[i]
             dog_path = dog_files_short[i]

             if dog_detector(human_path) == True:
                 human_detected += 1
             if dog_detector(dog_path) == True:
                 dog_detected += 1

         print('VGG-16 Prediction')
         print('The percentage of the detected dog - Humans: {0:.0%}'.format(human_detected / nu
         print('The percentage of the detected dog - Dogs: {0:.0%}'.format(dog_detected / num_fi
```

```
VGG-16 Prediction
The percentage of the detected dog - Humans: 0%
The percentage of the detected dog - Dogs: 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 [12]: ### (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 [13]:
```
from PIL import ImageFile

ImageFile.LOAD_TRUNCATED_IMAGES = True
%matplotlib inline

# Check if CUDA is available
use_cuda = torch.cuda.is_available()
print ('use_cuda:', use_cuda)
```

```
use_cuda: True
```

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

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
# Declare the transforms for train, valid and test sets.
```

```
# Resize images
# Convert to Tensor
# Normalize images
transforms = {
    'train' : transforms.Compose([transforms.RandomResizedCrop(224),
                                  transforms.RandomHorizontalFlip(),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.485, 0.456, 0.406],
                                                       [0.229, 0.224, 0.225])]),

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

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

# Number of subprocesses
num_workers = 0
# How many samples will be loaded for one batch?
batch_size = 20



# Create image datasets (train, valid, test)
image_datasets = {x: datasets.ImageFolder(os.path.join('/data/dog_images/', x), transfo
                  for x in ['train', 'valid', 'test']}

# Create data loaders (train, valid, test)
data_loaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size
                                               shuffle=True, num_workers=num_workers)
                for x in ['train', 'valid', 'test']}

# Decrease batch size because of the out of memory in the GPU Instance
test_loader = torch.utils.data.DataLoader(image_datasets['test'], shuffle=True,
                                          batch_size=15)
```

In [15]:
```
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'valid', 'test']}

print('Number of records of training dataset: {}'.format(dataset_sizes['train']))
print('Number of records of validation dataset: {}'.format(dataset_sizes['valid']))
print('Number of records of test dataset: {}'.format(dataset_sizes['test']))
```

```
Number of records of training dataset: 6680
Number of records of validation dataset: 835
Number of records of test dataset: 836
```

```
In [16]: class_names = image_datasets['train'].classes

         print(class_names)
```

```
['001.Affenpinscher', '002.Afghan_hound', '003.Airedale_terrier', '004.Akita', '005.Alaskan_mala
```

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

Answer: I loaded in the training, test and validation data, then created data_loaders for each of these sets of data.

I resized all image to 224, center cropped and randomly flipping and rotating the given image data. Most of the pretrained models require the input to be 224x224 images. Also, we'll need to match the normalization used when the models were trained.the means are [0.485, 0.456, 0.406] and the standard deviations are [0.229, 0.224, 0.225].

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [17]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 16, 3, padding=1)
                 # convolutional layer
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

                 # convolutional layer
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 # linear layer (64 * 28 * 28 -> 500)
                 self.fc1 = nn.Linear(64 * 28 * 28, 500)
                 # linear layer (500 -> 133)
                 self.fc2 = nn.Linear(500, 133)
```

```
        # dropout layer (p=0.25)
        self.dropout = nn.Dropout(0.20)
        self.batch_norm = nn.BatchNorm1d(num_features=500)


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

        # add dropout layer
        x = self.dropout(x)

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

        # add dropout layer
        x = self.dropout(x)

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

        # add dropout layer
        x = self.dropout(x)

        # flatten image input
        # 64 * 28 * 28
#         x = x.view(-1, 64 * 28 * 28)
        x = x.view(x.size(0), -1)

        # add 1st hidden layer, with relu activation function
        x = F.relu(self.batch_norm(self.fc1(x)))

        # add dropout layer
        x = self.dropout(x)

        # add 2nd hidden layer, with relu activation function
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()
# check if CUDA is available
use_cuda = torch.cuda.is_available()
# 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: First layer has input shape of (224, 224, 3) and last layer should output 133 classes.

I started adding Convolutional layers (stack of filtered images) and Maxpooling layers(reduce the x-y size of an input, keeping only the most active pixels from the previous layer) as well as the usual Linear + Dropout layers to avoid overfitting and produce a 133-dim output.

MaxPooling2D seems to be a common choice to downsample in these type of classification problems and that is why I chose it.

The more convolutional layers we include, the more complex patterns in color and shape a model can detect.

The first layer in my CNN is a convolutional layer that takes (224, 224, 3) inpute shap.

I'd like my new layer to have 16 filters, each with a height and width of 3. When performing the convolution, I'd like the filter to jump 1 pixel at a time.

_nn.Conv2d(in_channels, out_channels, kernelsize, stride=1, padding=0)

I want this layer to have the same width and height as the input layer, so I will pad accordingly; Then, to construct this convolutional layer, I would use the following line of code: self.conv2 = nn.Conv2d(3, 32, 3, padding=1)

I am adding a pool layer that takes in a kernel_size and a stride after every convolution layer. This will down-sample the input's x-y dimensions, by a factor of 2:

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

I am adding a fully connected Linear Layer to produce a 133-dim output. As well as a Dropout layer to avoid overfitting.

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 [18]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.9)
```

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 [19]: from PIL import ImageFile

         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import PIL
         PIL.__version__

Out[19]: '5.4.1'
```

```
In [ ]: def train(n_epochs, train_loaders, valid_loaders, model, optimizer, criterion, use_cuda,
            """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(data_loaders['train']):
                    # move to GPU
                    if use_cuda:
                        data, target,model = data.to('cuda'), target.to('cuda'),model.to('cuda')
                    ## find the loss and update the model parameters accordingly
                    ## record the average training loss, using something like
                    ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_los
                    # clear the gradients of all optimized variables
                    optimizer.zero_grad()
                    # forward pass
                    output = model(data)
                    # calculate the batch loss
                    loss = criterion(output, target)
                    # backward pass
                    loss.backward()
                    # perform a single optimization step
                    optimizer.step()
                    # update training loss
                    train_loss += loss.item()*data.size(0)
                ######################
                # validate the model #
                ######################
                model.eval()
                for batch_idx, (data, target) in enumerate(data_loaders['valid']):
                    # move to GPU
                    if use_cuda:
                        data, target,model = data.to('cuda'), target.to('cuda'),model.to('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 += loss.item()*data.size(0)
```

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



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

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

        # return trained model
        return model


    # train the model
    model_scratch = train(50, data_loaders['train'], data_loaders['valid'], model_scratch, o
                          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: 4.820621         Validation Loss: 4.844724
Validation loss decreased (inf --> 4.844724).  Saving model ...
Epoch: 2         Training Loss: 4.686676         Validation Loss: 4.788158
Validation loss decreased (4.844724 --> 4.788158).  Saving model ...
Epoch: 3         Training Loss: 4.598934         Validation Loss: 4.764840
Validation loss decreased (4.788158 --> 4.764840).  Saving model ...
Epoch: 4         Training Loss: 4.533178         Validation Loss: 4.645814
Validation loss decreased (4.764840 --> 4.645814).  Saving model ...
Epoch: 5         Training Loss: 4.472453         Validation Loss: 4.648795
Epoch: 6         Training Loss: 4.419758         Validation Loss: 4.588765
Validation loss decreased (4.645814 --> 4.588765).  Saving model ...
Epoch: 7         Training Loss: 4.367096         Validation Loss: 4.536900
Validation loss decreased (4.588765 --> 4.536900).  Saving model ...
Epoch: 8         Training Loss: 4.324759         Validation Loss: 4.384063
Validation loss decreased (4.536900 --> 4.384063).  Saving model ...
```

```
Epoch: 9        Training Loss: 4.289678        Validation Loss: 4.437479
Epoch: 10        Training Loss: 4.244667        Validation Loss: 4.355126
Validation loss decreased (4.384063 --> 4.355126).  Saving model ...
Epoch: 11        Training Loss: 4.213155        Validation Loss: 4.285368
Validation loss decreased (4.355126 --> 4.285368).  Saving model ...
```

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 [ ]: 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,model = data.to('cuda'), target.to('cuda'),model.to('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(data_loaders, model_scratch, criterion_scratch, use_cuda)
```

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 [25]:
```
## TODO: Specify data loaders
loader_transfer = data_loaders
print(loader_transfer)
```

```
{'train': <torch.utils.data.dataloader.DataLoader object at 0x7f625f3d2550>, 'valid': <torch.uti
```

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 [26]:
```
import torchvision.models as models
import torch.nn as nn

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

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

Out[26]:
```
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=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (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
      (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
      )
    )
```

```
    (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=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (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
      (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=
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
      (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
      (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
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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
      (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
      )
    )
    (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
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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
      (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
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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
      (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
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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
      (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
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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_stat
      )
    )
    (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
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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_stat
      )
    )
    (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
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
      (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_stat
      (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
              (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
              (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
              (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_stat
              (relu): ReLU(inplace)
            )
          )
          (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
          (fc): Linear(in_features=2048, out_features=1000, bias=True)
        )
```

```
In [27]: for param in model_transfer.parameters():
             param.requires_grad = False
         model_transfer.fc = nn.Linear(2048 ,133)
```

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 picked ResNet as a transfer model because it performed outstanding on Image Classification. I looked into the structure and functions of ResNet. The core idea of ResNet is introducing a so-called "identity shortcut connection" that skips one or more layers. I guess this prevents overfitting when it's training. I'll use resnet50 trained on ImageNet available from torchvision.

The classifier is a single fully-connected layer:

(fc): Linear(in_features=2048, out_features=1000, bias=True)

This layer was trained on the ImageNet dataset.

we need to replace the classifier (133 classes), but the features will work perfectly on their own.

Choice of criterion: nn.CrossEntropyLoss() This criterion combines :func:nn.LogSoftmax and :func:nn.NLLLoss in one single class. It is useful when training a classification problem with C classes.

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 [28]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.fc.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 [30]: # train the model
         n_epochs = 16
         model_transfer =  train(n_epochs, data_loaders['train'], data_loaders['valid'], model_t
```

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

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

RuntimeError                              Traceback (most recent call last)

<ipython-input-30-43437f1a45d7> in <module>()
      1 # train the model
      2 n_epochs = 16
----> 3 model_transfer =  train(n_epochs, data_loaders['train'], data_loaders['valid'], mode
      4 
      5 # load the model that got the best validation accuracy (uncomment the line below)


<ipython-input-18-edaa0664a09f> in train(n_epochs, train_loaders, vaild_loader, model, o
     22             optimizer.zero_grad()
     23             # forward pass
---> 24             output = model(data)
     25             # calculate the batch loss
     26             loss = criterion(output, target)


/opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)


/opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/re
    149         x = self.avgpool(x)
    150         x = x.view(x.size(0), -1)
--> 151         x = self.fc(x)
    152 
    153         return x


/opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)
```

```
/opt/conda/lib/python3.6/site-packages/torch/nn/modules/linear.py in forward(self, input
     53 
     54     def forward(self, input):
---> 55         return F.linear(input, self.weight, self.bias)
     56 
     57     def extra_repr(self):


/opt/conda/lib/python3.6/site-packages/torch/nn/functional.py in linear(input, weight, b
    990     if input.dim() == 2 and bias is not None:
    991         # fused op is marginally faster
--> 992         return torch.addmm(bias, input, weight.t())
    993 
    994     output = input.matmul(weight.t())


RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.FloatT
```

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 [32]:
```
test(loader_transfer, model_transfer, criterion_transfer, use_cuda)
```

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

RuntimeError                              Traceback (most recent call last)

<ipython-input-32-09f342f34ad1> in <module>()
----> 1 test(loader_transfer, model_transfer, criterion_transfer, use_cuda)


<ipython-input-19-7d2a9540676a> in test(loaders, model, criterion, use_cuda)
     12             data, target = data.cuda(), target.cuda()
     13         # forward pass: compute predicted outputs by passing inputs to the model
---> 14         output = model(data)
     15         # calculate the loss
     16         loss = criterion(output, target)


/opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
```

```
    493             hook_result = hook(self, input, result)


    /opt/conda/lib/python3.6/site-packages/torchvision-0.2.1-py3.6.egg/torchvision/models/re
    149         x = self.avgpool(x)
    150         x = x.view(x.size(0), -1)
--> 151         x = self.fc(x)
    152 
    153         return x


    /opt/conda/lib/python3.6/site-packages/torch/nn/modules/module.py in __call__(self, *inp
    489             result = self._slow_forward(*input, **kwargs)
    490         else:
--> 491             result = self.forward(*input, **kwargs)
    492         for hook in self._forward_hooks.values():
    493             hook_result = hook(self, input, result)


    /opt/conda/lib/python3.6/site-packages/torch/nn/modules/linear.py in forward(self, input
     53 
     54     def forward(self, input):
---> 55         return F.linear(input, self.weight, self.bias)
     56 
     57     def extra_repr(self):


    /opt/conda/lib/python3.6/site-packages/torch/nn/functional.py in linear(input, weight, b
    990     if input.dim() == 2 and bias is not None:
    991         # fused op is marginally faster
--> 992         return torch.addmm(bias, input, weight.t())
    993 
    994     output = input.matmul(weight.t())


    RuntimeError: Expected object of type torch.FloatTensor but found type torch.cuda.FloatT
```

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 [34]:
```
### 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  image_datasets['train'].classes]
```

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

AttributeError                            Traceback (most recent call last)

<ipython-input-34-8533990c53ce> in <module>()
      3
      4 # list of class names by index, i.e. a name can be accessed like class_names[0]
----> 5 class_names = [item[4:].replace("_", " ") for item in loader_transfer['train'].class


AttributeError: 'DataLoader' object has no attribute 'classes'
```

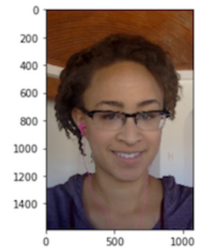
In [35]:
```python
from PIL import Image
import torchvision.transforms as transforms

def load_input_image(img_path):
    image = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                     transforms.ToTensor(),
                                          transforms.Normalize(mean=[0.485, 0.456,
                                             std=[0.229, 0.224, 0.225])])


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

In [36]:
```python
def predict_breed_transfer(model, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(img))
    return class_names[idx]
```

In [37]:
```python
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    predition = predict_breed_transfer(model_transfer, class_names, img_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

```
image_file_name: ./images/sample_human_output.png,          predition breed: 114.Otterhound
image_file_name: ./images/Curly-coated_retriever_03896.jpg,          predition breed: 131.Wireha
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition breed: 051.Chow_chow
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition breed: 052.Clumber_sp
image_file_name: ./images/sample_cnn.png,          predition breed: 114.Otterhound
image_file_name: ./images/American_water_spaniel_00648.jpg,          predition breed: 033.Bouvie
image_file_name: ./images/sample_dog_output.png,          predition breed: 051.Chow_chow
```

```
hello, human!
```




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

Sample Human Output

```
image_file_name: ./images/Brittany_02625.jpg,          predition breed: 104.Miniature_schnauzer
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,          predition breed: 056.Dachsh
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition breed: 052.Clumber_sp
```

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

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             if dog_detector(img_path) is True:
                 prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                 print("Dogs Detected!\nIt looks like a {0}".format(prediction))
             elif face_detector(img_path) > 0:
                 prediction = predict_breed_transfer(model_transfer, class_names, img_path)
                 print("Hello, human!\nYou may look 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?

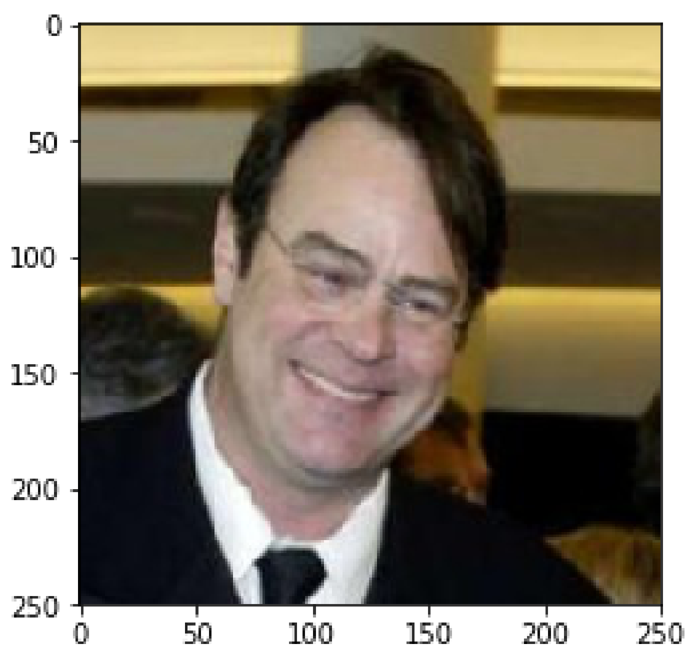
1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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.

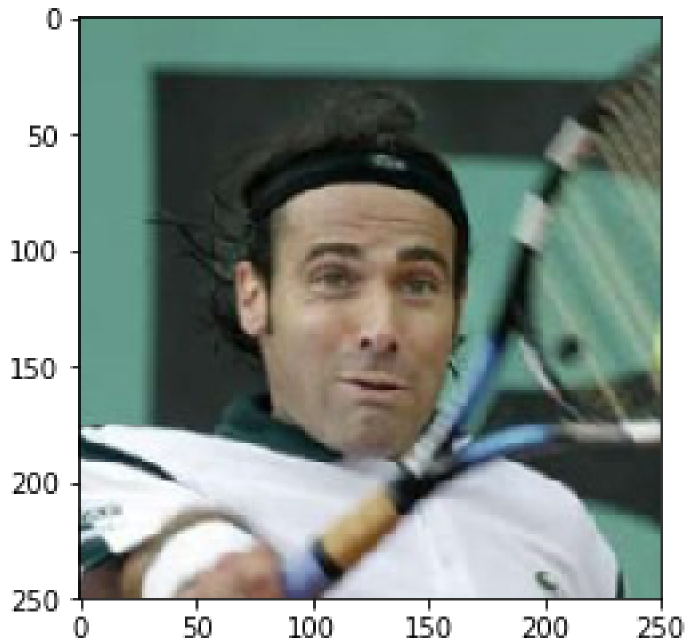
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement) * Fine tune the model to give a better accuracy * Handle better the case when there are multiple dogs/humans or dogs and humans in an image * Clean up code and make it more modular

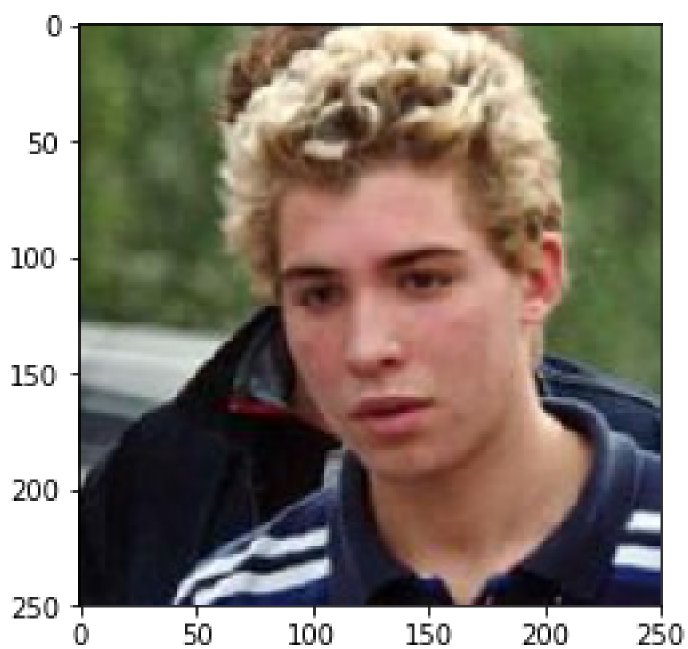
In [43]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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



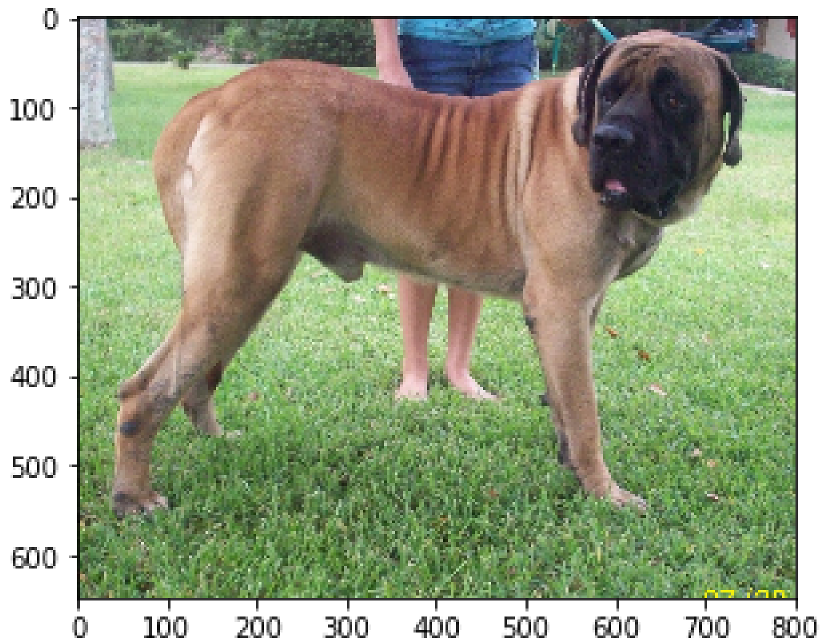
```
Hello, human!
You may look like a 003.Airedale_terrier
```



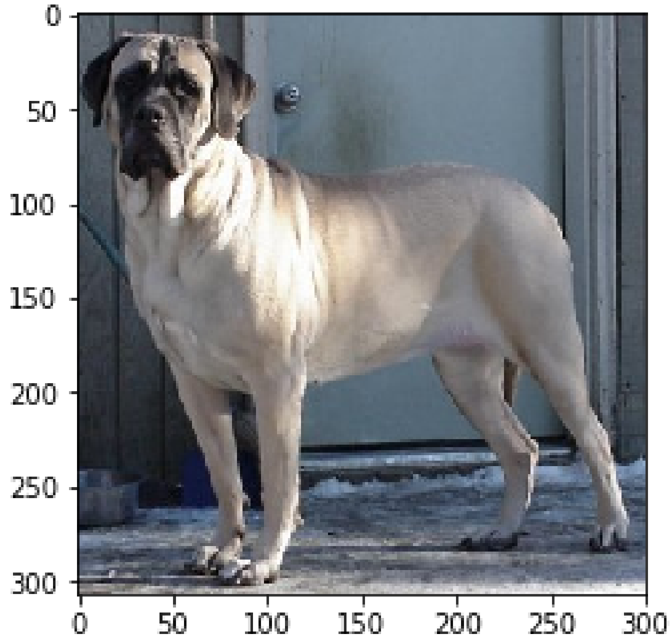

```
Hello, human!
You may look like a 073.German_wirehaired_pointer
```



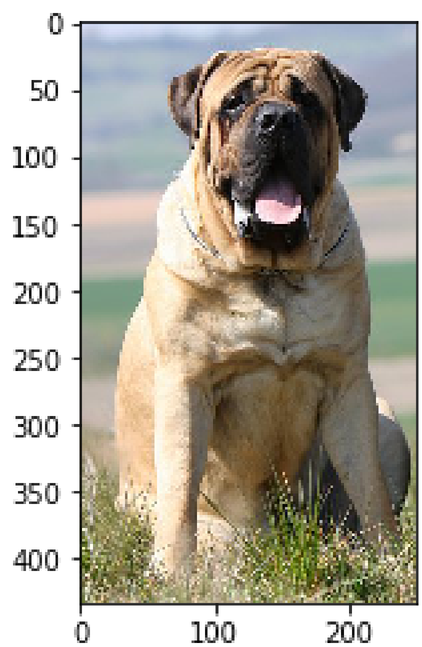
Hello, human!
You may look like a 107.Norfolk_terrier



Dogs Detected!
It looks like a 029.Border_collie




```
Dogs Detected!
It looks like a 056.Dachshund
```



```
Dogs Detected!
It looks like a 052.Clumber_spaniel
```

```
In [ ]: for img_file in os.listdir('./my_images'):
            img_path = os.path.join('./my_images', img_file)
            run_app(img_path)
```