dog_app

May 5, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
## Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets:
Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

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

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

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

```
In [2]: 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.

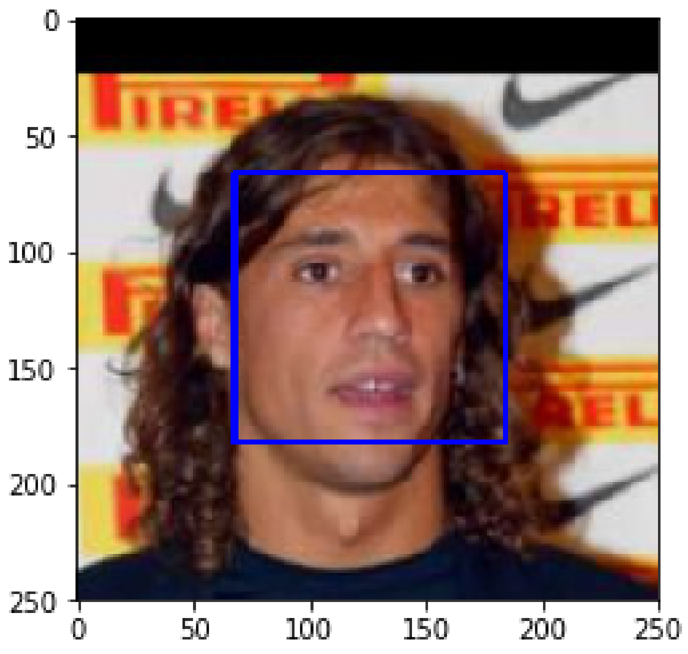
```
In [3]: pip install matplotlib==2.0.2
```

```
The following command must be run outside of the IPython shell:

    $ pip install matplotlib==2.0.2

The Python package manager (pip) can only be used from outside of IPython.
Please reissue the `pip` command in a separate terminal or command prompt.

See the Python documentation for more information on how to install packages:

    https://docs.python.org/3/installing/
```

```
In [4]: 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[18])

# 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)

- Percentage of humans = 98%
- Percentage of dogs = 17%

```
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 = []
        for f1 in tqdm(human_files_short):
            img = face_detector(f1)
            human.append(img)

        print ("Percentage of humans = " + str(human.count(1)) + "%" )

        dog = []
        for f1 in tqdm(dog_files_short):
            img = face_detector(f1)
            dog.append(img)

        print ("Percentage of dogs = " + str(dog.count(1)) + "%" )
```

```
100%|| 100/100 [00:02<00:00, 34.39it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

Percentage of humans = 98%


100%|| 100/100 [00:29<00:00,  7.31it/s]

Percentage of 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.

        # extract pre-trained face detector
        face_cascade2 = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt2.xml')

        # 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_cascade2.detectMultiScale(gray)
            return len(faces) > 0

        human = []
        for f1 in tqdm(human_files_short):
            img = face_detector(f1)
            human.append(img)

        print ("Percentage of humans = " + str(human.count(1)) + "%" )

        dog = []
        for f1 in tqdm(dog_files_short):
            img = face_detector(f1)
            dog.append(img)

        print ("Percentage of dogs = " + str(dog.count(1)) + "%" )
```

```
100%|| 100/100 [00:02<00:00, 39.18it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

Percentage of humans = 100%


100%|| 100/100 [00:27<00:00,  7.64it/s]

Percentage of dogs = 21%
```

## 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()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:07<00:00, 70664680.07it/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 [9]: 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

    image = Image.open(img_path).convert('RGB')
    # convert data to a normalized torch.FloatTensor
    transform = 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])
        ])

    #convert image to tensor
    image =  transform(image)[:3,:,:].unsqueeze(0)

    if use_cuda:
        image = image.cuda()

    predict = VGG16(image)
    _, predicted = torch.max(predict, 1)
    #predicted = np.argmax(predict, axis=-1)

    return predicted.item() # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             predicted = VGG16_predict(img_path)

             return (151 < predicted <= 268) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?
Answer:

- Percentage of dogs in human_files_short = 0%
- Percentage of dogs in dog_files_short = 100%

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human = []
         for f1 in tqdm(human_files_short):
             img = dog_detector(f1)
             human.append(img)

         print ("Percentage of dogs = " + str(human.count(1)) + "%" )

         dog = []
         for f1 in tqdm(dog_files_short):
             img = dog_detector(f1)
             dog.append(img)

         print ("Percentage of dogs = " + str(dog.count(1)) + "%" )
```

```
100%|| 100/100 [00:03<00:00, 30.04it/s]
  3%|         | 3/100 [00:00<00:03, 26.62it/s]

Percentage of dogs = 1%


100%|| 100/100 [00:04<00:00, 25.22it/s]

Percentage of 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 [14]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.

         alexnet = models.alexnet(pretrained=True)
```

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

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

```
Downloading: "https://download.pytorch.org/models/alexnet-owt-4df8aa71.pth" to /root/.torch/mode
100%|| 244418560/244418560 [00:13<00:00, 18623367.16it/s]
```

```
In [15]: def alexnet_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')
             # convert data to a normalized torch.FloatTensor
             transform = 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])
                 ])

             #convert image to tensor
             image =  transform(image)[:3,:,:].unsqueeze(0)

             if use_cuda:
                 image = image.cuda()

             predict = alexnet(image)
             _, predicted = torch.max(predict, 1)
             #predicted = np.argmax(predict, axis=-1)

             return predicted.item() # predicted class index
```

```
In [16]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector2(img_path):
             ## TODO: Complete the function.

             predicted = alexnet_predict(img_path)

             return (151 < predicted <= 268) # true/false

In [17]: human = []
         for f1 in tqdm(human_files_short):
             img = dog_detector2(f1)
             human.append(img)

         print ("Percentage of dogs = " + str(human.count(1)) + "%" )

         dog = []
         for f1 in tqdm(dog_files_short):
             img = dog_detector2(f1)
             dog.append(img)

         print ("Percentage of dogs = " + str(dog.count(1)) + "%" )
```

```
100%|| 100/100 [00:00<00:00, 107.27it/s]
  0%|          | 0/100 [00:00<?, ?it/s]

Percentage of dogs = 0%


100%|| 100/100 [00:02<00:00, 41.63it/s]

Percentage of dogs = 97%
```

## 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 [18]: import os
         from torchvision import datasets
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

         # define dataloader parameters
         batch_size = 20
         num_workers=0
         num_classes = 133

         train = transforms.Compose([
                 transforms.Resize(256),
                 transforms.RandomHorizontalFlip(p=0.5),
                transforms.ColorJitter(brightness=0.70, contrast=0.45, saturation=0, hue=0.25),
                transforms.RandomRotation(270),
                transforms.CenterCrop(224),
                transforms.ToTensor(),
                transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                        std=[0.229, 0.224, 0.225])
                ])
        validation = 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])
                ])
        test = 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])
                ])

        data_dir = '/data/dog_images'
        train_dir = os.path.join(data_dir, 'train/')
        valid_dir = os.path.join(data_dir, 'valid/')
        test_dir = os.path.join(data_dir, 'test/')

        train_data = datasets.ImageFolder(train_dir, transform=train)
        valid_data = datasets.ImageFolder(valid_dir, transform=validation)
        test_data = datasets.ImageFolder(test_dir, transform=test)

        # prepare data loaders
        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
        valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=False)
        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                  num_workers=num_workers, shuffle=False)

        loaders_scratch = {'train': train_loader, 'valid': valid_loader, 'test': test_loader}
        # print out some data stats
        print('Num training images: ', len(train_data))
        print('Num validation images: ', len(valid_data))
        print('Num test images: ', len(test_data))
```

```
Num training images:  6680
Num validation images:  835
Num test images:  836
```

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:

- Code resizes the image by cropping at tensor of 224, so that all the images are of the same size
- Yes, by flip, rotations and playing with colors(contrast, brightness, hue)

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [56]: 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, 28, 3, padding=1)
                 self.conv2 = nn.Conv2d(28, 56, 3, padding=1)
                 self.conv3 = nn.Conv2d(56, 112, 3, stride=3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 self.fc1 = nn.Linear(112 * 9 * 9, 5120)
                 self.fc2 = nn.Linear(5120, 1024)
                 self.fc3 = nn.Linear(1024, 133)
                 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)))
                 # flatten image input
                 #print(x.shape)
                 x = x.view(-1, 112 * 9 * 9)
                 # 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
            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()
```

```
In [58]: model_scratch
```

```
Out[58]: Net(
           (conv1): Conv2d(3, 28, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv2): Conv2d(28, 56, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv3): Conv2d(56, 112, kernel_size=(3, 3), stride=(3, 3), padding=(1, 1))
           (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
           (fc1): Linear(in_features=9072, out_features=5120, bias=True)
           (fc2): Linear(in_features=5120, out_features=1024, bias=True)
           (fc3): Linear(in_features=1024, out_features=133, bias=True)
           (dropout): Dropout(p=0.25)
         )
```

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

- With the input image of 224x224, and maintaining the using a 3x3 kernel size, with padding of 1 and stride of 1. We produce three convolutional layers from these, the image is resized to a depth of 28.
- I apply a max pooling layer
- I create three linear layers
- The image is flattened, a dropout layer is applied and relu activation is applied on the 1st and 2nd linear layer with a last linear layer

```
Net(
     (conv1): Conv2d(3, 28, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
     (conv2): Conv2d(28, 56, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
```

```
    (conv3): Conv2d(56, 112, kernel_size=(3, 3), stride=(3, 3), padding=(1, 1))
    (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (fc1): Linear(in_features=9072, out_features=5120, bias=True)
    (fc2): Linear(in_features=5120, out_features=1024, bias=True)
    (fc3): Linear(in_features=1024, out_features=133, bias=True)
    (dropout): Dropout(p=0.25)
  )
```

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 [60]: 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 [61]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     # clear the gradients of all optimized variables
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
```

```
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            ## record the average training loss, using something like
            #train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss
            train_loss += loss.item()*data.size(0)

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            #valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss
            valid_loss += loss.item()*data.size(0)

        train_loss = train_loss/len(train_data)
        valid_loss = valid_loss/len(valid_data)

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

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

    # return trained model
    return model
```

```
# train the model
model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.882762         Validation Loss: 4.864083
Validation loss decreased (inf --> 4.864083).  Saving model ...
Epoch: 2         Training Loss: 4.854634         Validation Loss: 4.803589
Validation loss decreased (4.864083 --> 4.803589).  Saving model ...
Epoch: 3         Training Loss: 4.781064         Validation Loss: 4.705515
Validation loss decreased (4.803589 --> 4.705515).  Saving model ...
Epoch: 4         Training Loss: 4.729337         Validation Loss: 4.643237
Validation loss decreased (4.705515 --> 4.643237).  Saving model ...
Epoch: 5         Training Loss: 4.663815         Validation Loss: 4.548336
Validation loss decreased (4.643237 --> 4.548336).  Saving model ...
Epoch: 6         Training Loss: 4.573821         Validation Loss: 4.491800
Validation loss decreased (4.548336 --> 4.491800).  Saving model ...
Epoch: 7         Training Loss: 4.504210         Validation Loss: 4.441577
Validation loss decreased (4.491800 --> 4.441577).  Saving model ...
Epoch: 8         Training Loss: 4.460880         Validation Loss: 4.442853
Epoch: 9         Training Loss: 4.417995         Validation Loss: 4.386418
Validation loss decreased (4.441577 --> 4.386418).  Saving model ...
Epoch: 10         Training Loss: 4.390779         Validation Loss: 4.374327
Validation loss decreased (4.386418 --> 4.374327).  Saving model ...
Epoch: 11         Training Loss: 4.344354         Validation Loss: 4.400632
Epoch: 12         Training Loss: 4.322942         Validation Loss: 4.307265
Validation loss decreased (4.374327 --> 4.307265).  Saving model ...
Epoch: 13         Training Loss: 4.283368         Validation Loss: 4.266696
Validation loss decreased (4.307265 --> 4.266696).  Saving model ...
Epoch: 14         Training Loss: 4.259424         Validation Loss: 4.293739
Epoch: 15         Training Loss: 4.210891         Validation Loss: 4.206538
Validation loss decreased (4.266696 --> 4.206538).  Saving model ...
Epoch: 16         Training Loss: 4.181589         Validation Loss: 4.236521
Epoch: 17         Training Loss: 4.154693         Validation Loss: 4.168289
Validation loss decreased (4.206538 --> 4.168289).  Saving model ...
Epoch: 18         Training Loss: 4.111999         Validation Loss: 4.138176
Validation loss decreased (4.168289 --> 4.138176).  Saving model ...
Epoch: 19         Training Loss: 4.093443         Validation Loss: 4.160531
Epoch: 20         Training Loss: 4.064591         Validation Loss: 4.126647
Validation loss decreased (4.138176 --> 4.126647).  Saving model ...
Epoch: 21         Training Loss: 4.032932         Validation Loss: 4.067719
Validation loss decreased (4.126647 --> 4.067719).  Saving model ...
Epoch: 22         Training Loss: 4.005106         Validation Loss: 4.157981
Epoch: 23         Training Loss: 3.979077         Validation Loss: 4.079214
Epoch: 24         Training Loss: 3.955450         Validation Loss: 4.077508
```

```
Epoch: 25         Training Loss: 3.903855         Validation Loss: 4.020667
Validation loss decreased (4.067719 --> 4.020667).  Saving model ...
Epoch: 26         Training Loss: 3.897173         Validation Loss: 4.053967
Epoch: 27         Training Loss: 3.855594         Validation Loss: 4.078908
Epoch: 28         Training Loss: 3.832076         Validation Loss: 3.994348
Validation loss decreased (4.020667 --> 3.994348).  Saving model ...
Epoch: 29         Training Loss: 3.798742         Validation Loss: 3.999785
Epoch: 30         Training Loss: 3.794712         Validation Loss: 4.113721
Epoch: 31         Training Loss: 3.740271         Validation Loss: 4.020485
Epoch: 32         Training Loss: 3.723316         Validation Loss: 3.900496
Validation loss decreased (3.994348 --> 3.900496).  Saving model ...
Epoch: 33         Training Loss: 3.683839         Validation Loss: 3.987203
Epoch: 34         Training Loss: 3.674013         Validation Loss: 3.887048
Validation loss decreased (3.900496 --> 3.887048).  Saving model ...
Epoch: 35         Training Loss: 3.633167         Validation Loss: 3.901886
Epoch: 36         Training Loss: 3.615865         Validation Loss: 3.991924
Epoch: 37         Training Loss: 3.591522         Validation Loss: 3.864630
Validation loss decreased (3.887048 --> 3.864630).  Saving model ...
Epoch: 38         Training Loss: 3.549038         Validation Loss: 3.869951
Epoch: 39         Training Loss: 3.543606         Validation Loss: 3.838102
Validation loss decreased (3.864630 --> 3.838102).  Saving model ...
Epoch: 40         Training Loss: 3.512233         Validation Loss: 3.937259
Epoch: 41         Training Loss: 3.474342         Validation Loss: 3.821909
Validation loss decreased (3.838102 --> 3.821909).  Saving model ...
Epoch: 42         Training Loss: 3.459136         Validation Loss: 3.850329
Epoch: 43         Training Loss: 3.440789         Validation Loss: 3.924095
Epoch: 44         Training Loss: 3.450480         Validation Loss: 3.852177
Epoch: 45         Training Loss: 3.385861         Validation Loss: 3.870952
Epoch: 46         Training Loss: 3.338431         Validation Loss: 3.925692
Epoch: 47         Training Loss: 3.328619         Validation Loss: 3.902689
Epoch: 48         Training Loss: 3.321177         Validation Loss: 3.923892
Epoch: 49         Training Loss: 3.277735         Validation Loss: 3.885371
Epoch: 50         Training Loss: 3.253857         Validation Loss: 3.973604
```

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

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [62]: 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: 3.799080


Test Accuracy: 11% (94/836)
```

## 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 [19]: 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 [20]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)
         #model_transfer = models.alexnet(pretrained=True)

         # Freeze training for all "features" layers
         for param in model_transfer.features.parameters():
             param.requires_grad = False

         import torch.nn as nn

         n_inputs = model_transfer.classifier[6].in_features

         # add last linear layer (n_inputs -> 133 classes)
         num_classes = 133
         # new layers automatically have requires_grad = True
         last_layer = nn.Linear(n_inputs, num_classes)

         model_transfer.classifier[6] = last_layer

         print(model_transfer)

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

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

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 chose the VGG16.

Then freeze the parameter of the feature layer, then changed the out_features of the last linear layer of the classifier from 1000 to 133 (the number of our classes for this problem).

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function and optimizer. Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

```
In [21]: import torch.optim as optim

         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 [84]: # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 4.826039         Validation Loss: 4.348639
Validation loss decreased (inf --> 4.348639).  Saving model ...
Epoch: 2         Training Loss: 4.466857         Validation Loss: 3.713335
Validation loss decreased (4.348639 --> 3.713335).  Saving model ...
Epoch: 3         Training Loss: 4.071286         Validation Loss: 2.973053
Validation loss decreased (3.713335 --> 2.973053).  Saving model ...
Epoch: 4         Training Loss: 3.680192         Validation Loss: 2.296467
Validation loss decreased (2.973053 --> 2.296467).  Saving model ...
Epoch: 5         Training Loss: 3.364021         Validation Loss: 1.810371
Validation loss decreased (2.296467 --> 1.810371).  Saving model ...
Epoch: 6         Training Loss: 3.174762         Validation Loss: 1.524000
Validation loss decreased (1.810371 --> 1.524000).  Saving model ...
Epoch: 7         Training Loss: 3.017953         Validation Loss: 1.320561
Validation loss decreased (1.524000 --> 1.320561).  Saving model ...
Epoch: 8         Training Loss: 2.877753         Validation Loss: 1.186723
Validation loss decreased (1.320561 --> 1.186723).  Saving model ...
Epoch: 9         Training Loss: 2.797505         Validation Loss: 1.097868
Validation loss decreased (1.186723 --> 1.097868).  Saving model ...
Epoch: 10         Training Loss: 2.672190         Validation Loss: 1.031746
Validation loss decreased (1.097868 --> 1.031746).  Saving model ...
Epoch: 11         Training Loss: 2.638044         Validation Loss: 0.958858
Validation loss decreased (1.031746 --> 0.958858).  Saving model ...
Epoch: 12         Training Loss: 2.572917         Validation Loss: 0.925159
Validation loss decreased (0.958858 --> 0.925159).  Saving model ...
Epoch: 13         Training Loss: 2.530705         Validation Loss: 0.896418
Validation loss decreased (0.925159 --> 0.896418).  Saving model ...
Epoch: 14         Training Loss: 2.492792         Validation Loss: 0.850495
Validation loss decreased (0.896418 --> 0.850495).  Saving model ...
Epoch: 15         Training Loss: 2.443943         Validation Loss: 0.834106
Validation loss decreased (0.850495 --> 0.834106).  Saving model ...
Epoch: 16         Training Loss: 2.413645         Validation Loss: 0.795826
Validation loss decreased (0.834106 --> 0.795826).  Saving model ...
Epoch: 17         Training Loss: 2.377221         Validation Loss: 0.789291
Validation loss decreased (0.795826 --> 0.789291).  Saving model ...
Epoch: 18         Training Loss: 2.356160         Validation Loss: 0.777228
Validation loss decreased (0.789291 --> 0.777228).  Saving model ...
Epoch: 19         Training Loss: 2.295306         Validation Loss: 0.758155
```

```
Validation loss decreased (0.777228 --> 0.758155).  Saving model ...
Epoch: 20         Training Loss: 2.320753         Validation Loss: 0.754201
Validation loss decreased (0.758155 --> 0.754201).  Saving model ...
```

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

```
Test Loss: 0.807640


Test Accuracy: 75% (627/836)
```

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

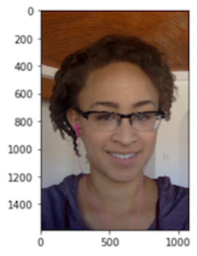
```
In [24]: class_names = train_data.classes
```

```
In [25]: class_names = [s[4:] for s in class_names]
         class_names
```

```
Out[25]: ['Affenpinscher',
          'Afghan_hound',
          'Airedale_terrier',
          'Akita',
          'Alaskan_malamute',
          'American_eskimo_dog',
          'American_foxhound',
          'American_staffordshire_terrier',
          'American_water_spaniel',
          'Anatolian_shepherd_dog',
          'Australian_cattle_dog',
          'Australian_shepherd',
          'Australian_terrier',
          'Basenji',
          'Basset_hound',
          'Beagle',
          'Bearded_collie',
          'Beauceron',
'Bedlington_terrier',
'Belgian_malinois',
'Belgian_sheepdog',
'Belgian_tervuren',
'Bernese_mountain_dog',
'Bichon_frise',
'Black_and_tan_coonhound',
'Black_russian_terrier',
'Bloodhound',
'Bluetick_coonhound',
'Border_collie',
'Border_terrier',
'Borzoi',
'Boston_terrier',
'Bouvier_des_flandres',
'Boxer',
'Boykin_spaniel',
'Briard',
'Brittany',
'Brussels_griffon',
'Bull_terrier',
'Bulldog',
'Bullmastiff',
'Cairn_terrier',
'Canaan_dog',
'Cane_corso',
'Cardigan_welsh_corgi',
'Cavalier_king_charles_spaniel',
'Chesapeake_bay_retriever',
'Chihuahua',
'Chinese_crested',
'Chinese_shar-pei',
'Chow_chow',
'Clumber_spaniel',
'Cocker_spaniel',
'Collie',
'Curly-coated_retriever',
'Dachshund',
'Dalmatian',
'Dandie_dinmont_terrier',
'Doberman_pinscher',
'Dogue_de_bordeaux',
'English_cocker_spaniel',
'English_setter',
'English_springer_spaniel',
'English_toy_spaniel',
'Entlebucher_mountain_dog',
'Field_spaniel',
'Finnish_spitz',
'Flat-coated_retriever',
'French_bulldog',
'German_pinscher',
'German_shepherd_dog',
'German_shorthaired_pointer',
'German_wirehaired_pointer',
'Giant_schnauzer',
'Glen_of_imaal_terrier',
'Golden_retriever',
'Gordon_setter',
'Great_dane',
'Great_pyrenees',
'Greater_swiss_mountain_dog',
'Greyhound',
'Havanese',
'Ibizan_hound',
'Icelandic_sheepdog',
'Irish_red_and_white_setter',
'Irish_setter',
'Irish_terrier',
'Irish_water_spaniel',
'Irish_wolfhound',
'Italian_greyhound',
'Japanese_chin',
'Keeshond',
'Kerry_blue_terrier',
'Komondor',
'Kuvasz',
'Labrador_retriever',
'Lakeland_terrier',
'Leonberger',
'Lhasa_apso',
'Lowchen',
'Maltese',
'Manchester_terrier',
'Mastiff',
'Miniature_schnauzer',
'Neapolitan_mastiff',
'Newfoundland',
'Norfolk_terrier',
'Norwegian_buhund',
'Norwegian_elkhound',
'Norwegian_lundehund',
'Norwich_terrier',
'Nova_scotia_duck_tolling_retriever',
'Old_english_sheepdog',
'Otterhound',
 'Papillon',
 'Parson_russell_terrier',
 'Pekingese',
 'Pembroke_welsh_corgi',
 'Petit_basset_griffon_vendeen',
 'Pharaoh_hound',
 'Plott',
 'Pointer',
 'Pomeranian',
 'Poodle',
 'Portuguese_water_dog',
 'Saint_bernard',
 'Silky_terrier',
 'Smooth_fox_terrier',
 'Tibetan_mastiff',
 'Welsh_springer_spaniel',
 'Wirehaired_pointing_griffon',
 'Xoloitzcuintli',
 'Yorkshire_terrier']
```

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.


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

    image = Image.open(img_path).convert('RGB')
    # convert data to a normalized torch.FloatTensor
    transform = 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])
        ])

    #convert image to tensor
    image =  transform(image)[:3,:,:].unsqueeze(0)

    if use_cuda:
        image = image.cuda()

    predict = model_transfer(image)
    _, predicted = torch.max(predict, 1)
    #predicted = np.argmax(predict, axis=-1)

    return class_names[predicted.item()] # predicted class name
```

```
hello, human!
```



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

Sample Human Output

## 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 [33]: ### 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
             image = Image.open(img_path)

             if dog_detector(img_path):
                 print ('Image uploaded is that of a dog')
                 plt.imshow(image)
                 plt.show()
                 prediction = predict_breed_transfer(img_path)
                 print("It looks like a {0}".format(prediction))
                 print ("")

             elif face_detector(img_path):
                 print ('Image uploaded is that of a human')
                 plt.imshow(image)
                 plt.show()
                 prediction = predict_breed_transfer(img_path)
                 print("If human were a dog, he/she would look like a {0}".format(prediction))
```

```
            print ("")

        else:
            print ("Image can't be detected")
            plt.imshow(image)
            plt.show()
```

## 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)

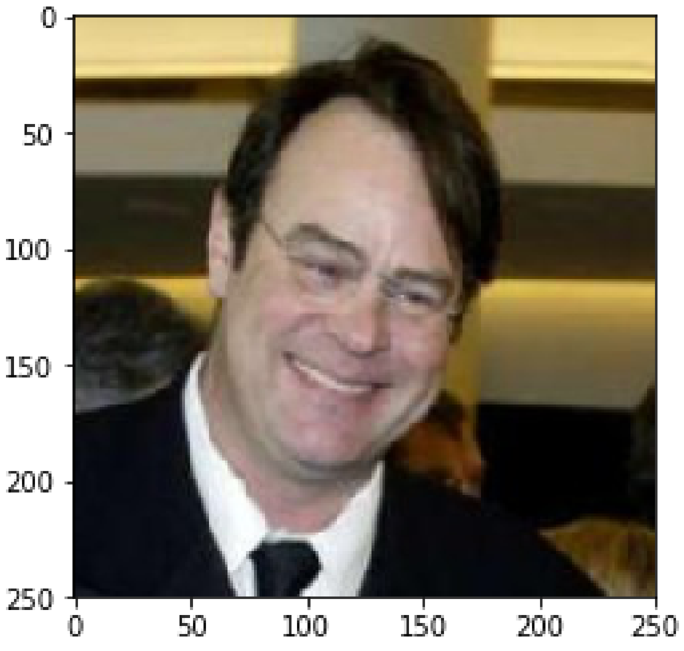
Output is very good.

Some points for improvement would be;

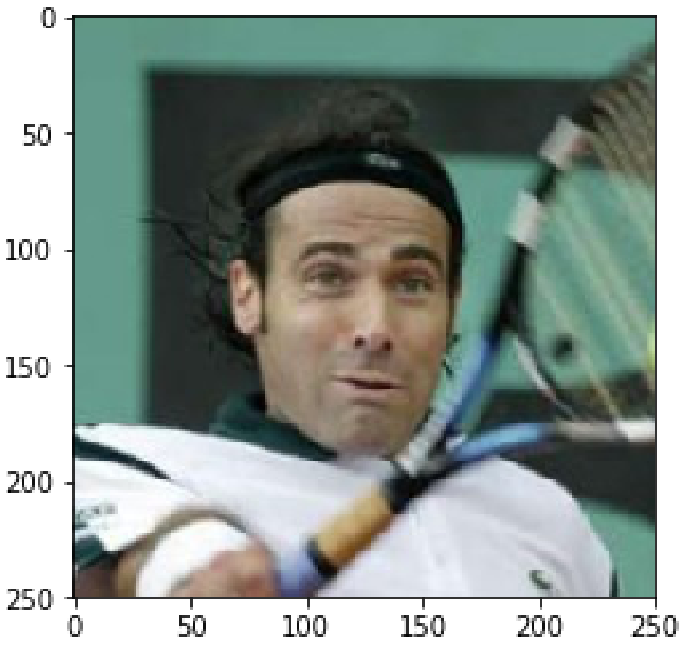
```
- Parameter tuning - learning rate and optimizer
- More training images for the model
- Using a model with more linear layers
```

```
In [34]: ## 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)
```

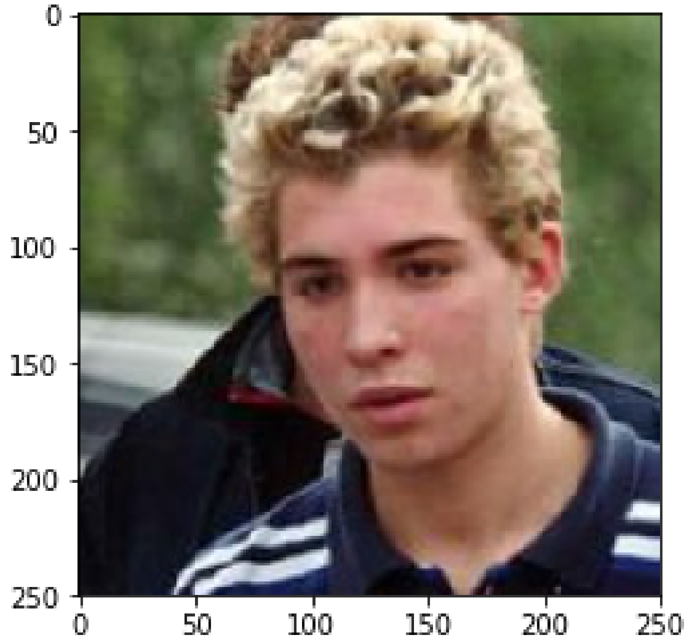
```
Image uploaded is that of a human
```



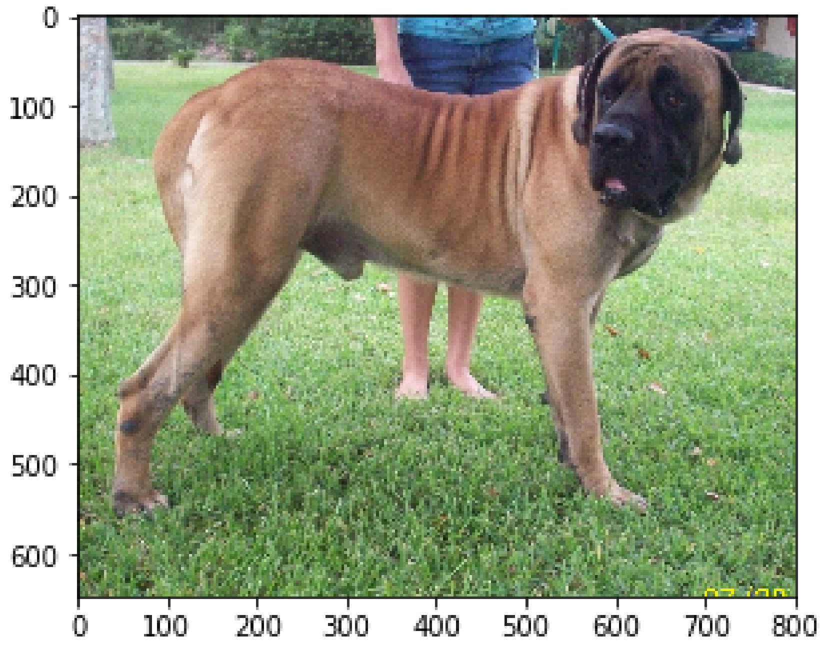
```
If human were a dog, he/she would look like a English_cocker_spaniel

Image uploaded is that of a human
```



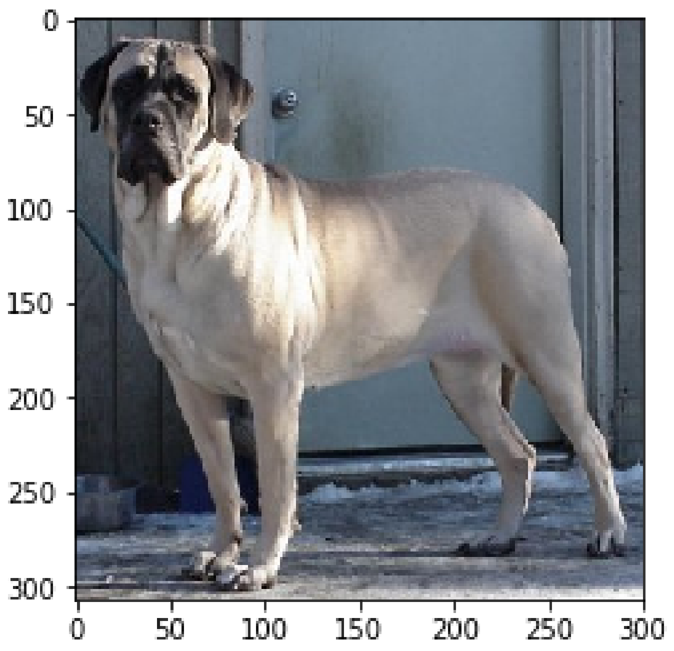
```
If human were a dog, he/she would look like a Chinese_crested

Image uploaded is that of a human
```



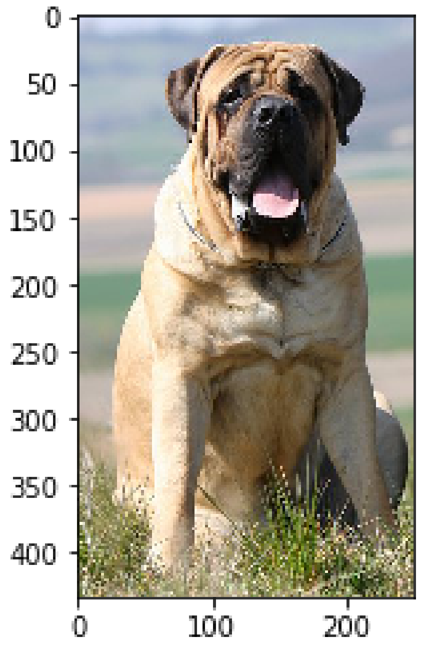
```
If human were a dog, he/she would look like a Afghan_hound

Image uploaded is that of a dog
```



```
It looks like a Bullmastiff

Image uploaded is that of a dog
```



```
It looks like a Mastiff

Image uploaded is that of a dog
```



```
It looks like a Bullmastiff
```

```
In [ ]:
```

```
In [ ]:
```