dog_app

October 8, 2023

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 [1]: import numpy as np
        from glob import glob

        # load filenames for human and dog images
        human_files = np.array(glob("/data/lfw/*/*"))
        dog_files = np.array(glob("/data/dog_images/*/*/*"))

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

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

Step 1: Detect Humans

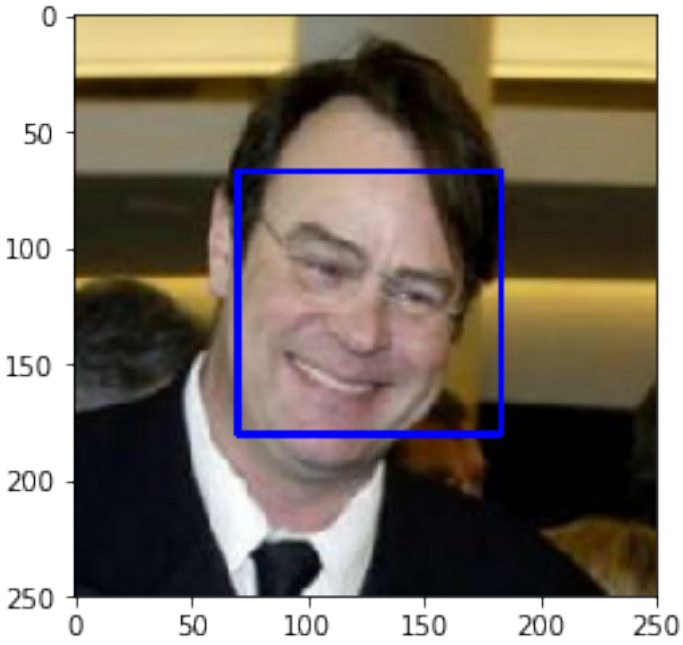
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: * Humans Classified correctly 98.0% * Dogs missclassified 17.0%

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

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

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

        ## DONE: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        human_face = [face_detector(h_path) for h_path in tqdm(human_files_short, desc='Humans')
        missclassified_faces = [face_detector(d_path) for d_path in tqdm(dog_files_short, desc='

        print(f'{sum(human_face) / len(human_files_short) * 100:4.1f}% of humans classified corr
        print(f'{sum(missclassified_faces) / len(dog_files_short) * 100:4.1f}% of dogs missclass
```

```
Humans: 100%|| 100/100 [00:14<00:00,  6.44it/s]
Dogs: 100%|| 100/100 [02:13<00:00,  2.16it/s]

98.0% of humans classified correctly
17.0% of dogs missclassified
```

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

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

Step 2: Detect Dogs

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models
        import torchvision.transforms as transforms
        from torch.autograd import Variable
        import torch.optim as optim
        from torchvision import datasets
        import torch.nn as nn
        import torch.nn.functional as F

        # 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 [00:06<00:00, 90806023.95it/s]
```

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

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

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

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

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

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

            Args:
                img_path: path to an image

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

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

            #Import the img_path
            img = Image.open(img_path)

            #Define normalization for image
            normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.22

            #Define transformations of the image
            preprocess = transforms.Compose([transforms.Resize(256),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             normalize])
            #preprocess the image
            img_tensor = preprocess(img).unsqueeze_(0)

            #Move tensor to GPU
            if use_cuda:
                img_tensor = img_tensor.cuda()

            #turn on eval mode
            VGG16.eval()
```

```
    #get predicted categort for the image
    with torch.no_grad():
        output = VGG16(img_tensor)
        prediction = torch.argmax(output).item()

    VGG16.train()

    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 [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## DONE: Complete the function.
            prediction = VGG16_predict(img_path)
            return True if 151 <= prediction <= 268 else False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: * Dogs detected in human_files_short: 0.0% * Dogs detected in dog_files_short: 100.0%

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

        #Human _files short
        dogs_in_human_files_VGG16 = 0
        for path in tqdm(human_files_short, desc='Human files'):
            if dog_detector(path):
                dogs_in_human_files_VGG16 += 1

        #Dog files short
        dogs_in_dog_files_VGG16 = 0
        for path in tqdm(dog_files_short, desc='Dog files'):
            if dog_detector(path):
                dogs_in_dog_files_VGG16 += 1
```

```
        print('#### VGG16 ####')
        print(f'Dogs detected in "human_files_short": {dogs_in_human_files_VGG16 / len(human_fil
        print(f'Dogs detected in "dog_files_short": {dogs_in_dog_files_VGG16 / len(dog_files_sho
```

```
Human files: 100%|| 100/100 [00:05<00:00, 19.88it/s]
Dog files: 100%|| 100/100 [00:06<00:00, 15.76it/s]

#### VGG16 ####
Dogs detected in "human_files_short": 0.0%
Dogs detected in "dog_files_short": 100.0%
```

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

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

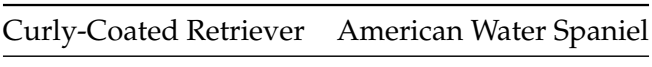
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany Welsh Springer Spaniel

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



Curly-Coated Retriever American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these

different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

In [11]:
```
import os

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

#Train and valid test transforms
train_transforms = transforms.Compose([transforms.Resize(size=258),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomRotation(10),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])])

validTest_transforms = transforms.Compose([transforms.Resize(size=258),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize([0.5, 0.5, 0.5],
                                                                [0.5, 0.5, 0.5])])
#Train, valid and test datasets
train_dataset = datasets.ImageFolder("../../../data/dog_images/train", transform=train_
valid_dataset = datasets.ImageFolder("../../../data/dog_images/valid", transform=validT
test_dataset = datasets.ImageFolder("../../../data/dog_images/test", transform=validTes

#Train, Valid and test loaders with batch sizes
trainLoader = torch.utils.data.DataLoader(train_dataset,
                                          batch_size=50,
                                          shuffle=True,
                                          num_workers=0)
```

```
validLoader = torch.utils.data.DataLoader(valid_dataset,
                                          batch_size=50,
                                          shuffle=True,
                                          num_workers=0)

testLoader = torch.utils.data.DataLoader(test_dataset,
                                          batch_size=25,
                                          shuffle=False,
                                          num_workers=0)
```

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: ##### 1) Image resizing In the original VGG16 paper by Simonyan and Zisserman (2015), the authors used a 224x224 pixel image as input, randomly cropped from a rescaled version of the original image. The exact reason for this choice is unclear, but one possible explanation is that it results in a 7x7 image after 5 maxpool layers, which has a center point.

To ensure that the cropped image contains the features we are interested in, we must rescale the original image, as cropping a 224x224 image out of a much larger original image is unlikely to do so. Following the Simonyan et al. paper, I rescaled the original images to 256x256 pixels before cropping.

2) Data Augmentation To improve the model's generalization ability, I augmented the image data by randomly rotating images up to 10 degrees and flipping them horizontally. Data augmentation is a simple way to increase the size and diversity of a dataset, which can lead to better model performance.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: # define the CNN architecture
         class Net(nn.Module):
             ### DONE: 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, kernel_size=3, padding=1)
                 self.pool1 = nn.MaxPool2d(2, 2)

                 self.conv2 = nn.Conv2d(16, 32, kernel_size=3, padding=1)
                 self.pool2 = nn.MaxPool2d(2, 2)

                 self.conv3 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
                 self.pool3 = nn.MaxPool2d(2, 2)
```

```
        self.conv4 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.pool4 = nn.MaxPool2d(2, 2)

        self.conv5 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.pool5 = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(256 * 7 * 7, 500)
        self.fc2 = nn.Linear(500, 133)
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = self.pool4(F.relu(self.conv4(x)))
        x = self.pool5(F.relu(self.conv5(x)))
        x = self.dropout(x)

        x = x.view(-1, 256 * 7 * 7)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)
        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: I decided to follow the original VGG16 paper and implemented the simplest model from Table 1, Column A. My model consists of 5 convolutional layers with a kernel size of 3x3, padding of 1, and a gradual increase in the number of feature maps. In between each convolutional layer is a maxpool layer with a 2x2 kernel and stride of 2, which halves the size of all feature maps. After 5 convolutions and maxpool layers, we end up with 512 7x7 feature maps.

The feature maps are then flattened to a vector of length 25088 and fed into 3 fully connected (FC) layers for classification. I reduced the number of nodes per layer in the FC layers compared to the VGG16 paper, as we only have 133 classes instead of 1000.

Finally, instead of a softmax layer as the last layer in my network, I used an FC layer. This is because I used PyTorch's CrossEntropyLoss() class for training, which combines a log-softmax output layer activation and a negative log-likelihood loss function. When testing the neural network, the output of the network is fed into a softmax function to obtain class probabilities.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

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

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

         ### DONE: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

```
In [14]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

                     # Clear the gradients
                     optimizer.zero_grad()
                     # forwarward pass
                     output = model(data)
                     # calculate batch loss
                     loss = criterion(output, target)
                     # backward pass
                     loss.backward()
                     # perform optimization step
```

```
                optimizer.step()
                # update training loss
                train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                with torch.no_grad():
                    output = model(data)
                loss = criterion(output, target)
                valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))


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

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

        # return trained model
        return model

    # define loaders_scratch
    loaders_scratch = {'train': trainLoader,
                       'valid': validLoader,
                       'test': testLoader}
    # train the model
    model_scratch = train(25, 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: 4.881396         Validation Loss: 4.794728
Validation loss decreased (inf --> 4.794728). Saving model...
```

```
Epoch: 2         Training Loss: 4.694956         Validation Loss: 4.540327
Validation loss decreased (4.794728 --> 4.540327). Saving model...
Epoch: 3         Training Loss: 4.459741         Validation Loss: 4.341259
Validation loss decreased (4.540327 --> 4.341259). Saving model...
Epoch: 4         Training Loss: 4.283921         Validation Loss: 4.202704
Validation loss decreased (4.341259 --> 4.202704). Saving model...
Epoch: 5         Training Loss: 4.146477         Validation Loss: 4.071009
Validation loss decreased (4.202704 --> 4.071009). Saving model...
Epoch: 6         Training Loss: 4.010734         Validation Loss: 3.956155
Validation loss decreased (4.071009 --> 3.956155). Saving model...
Epoch: 7         Training Loss: 3.893957         Validation Loss: 3.888801
Validation loss decreased (3.956155 --> 3.888801). Saving model...
Epoch: 8         Training Loss: 3.786501         Validation Loss: 3.801845
Validation loss decreased (3.888801 --> 3.801845). Saving model...
Epoch: 9         Training Loss: 3.678520         Validation Loss: 3.773283
Validation loss decreased (3.801845 --> 3.773283). Saving model...
Epoch: 10         Training Loss: 3.584975         Validation Loss: 3.744219
Validation loss decreased (3.773283 --> 3.744219). Saving model...
Epoch: 11         Training Loss: 3.498338         Validation Loss: 3.675176
Validation loss decreased (3.744219 --> 3.675176). Saving model...
Epoch: 12         Training Loss: 3.422376         Validation Loss: 3.645039
Validation loss decreased (3.675176 --> 3.645039). Saving model...
Epoch: 13         Training Loss: 3.337576         Validation Loss: 3.634536
Validation loss decreased (3.645039 --> 3.634536). Saving model...
Epoch: 14         Training Loss: 3.262336         Validation Loss: 3.622879
Validation loss decreased (3.634536 --> 3.622879). Saving model...
Epoch: 15         Training Loss: 3.174767         Validation Loss: 3.616030
Validation loss decreased (3.622879 --> 3.616030). Saving model...
Epoch: 16         Training Loss: 3.103293         Validation Loss: 3.565116
Validation loss decreased (3.616030 --> 3.565116). Saving model...
Epoch: 17         Training Loss: 3.061470         Validation Loss: 3.512646
Validation loss decreased (3.565116 --> 3.512646). Saving model...
Epoch: 18         Training Loss: 2.971161         Validation Loss: 3.457260
Validation loss decreased (3.512646 --> 3.457260). Saving model...
Epoch: 19         Training Loss: 2.894192         Validation Loss: 3.461219
Epoch: 20         Training Loss: 2.824546         Validation Loss: 3.461739
Epoch: 21         Training Loss: 2.748347         Validation Loss: 3.436605
Validation loss decreased (3.457260 --> 3.436605). Saving model...
Epoch: 22         Training Loss: 2.694246         Validation Loss: 3.376447
Validation loss decreased (3.436605 --> 3.376447). Saving model...
Epoch: 23         Training Loss: 2.644095         Validation Loss: 3.457834
Epoch: 24         Training Loss: 2.588599         Validation Loss: 3.420929
Epoch: 25         Training Loss: 2.513115         Validation Loss: 3.489680
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [15]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 3.446848


Test Accuracy: 17% (148/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

In [16]:
```
## DONE: Specify data loaders

        data_dir = 'dogImages'

        train_transforms = transforms.Compose([transforms.Resize(size=258),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.RandomRotation(10),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],
                                                                    [0.229, 0.224, 0.225])])

        validTest_transforms = transforms.Compose([transforms.Resize(size=258),
                                                   transforms.CenterCrop(224),
                                                   transforms.ToTensor(),
                                                   transforms.Normalize([0.485, 0.456, 0.406],
                                                                        [0.229, 0.224, 0.225])])

        train_dataset = datasets.ImageFolder("../../../data/dog_images/train", transform=train_
        valid_dataset = datasets.ImageFolder("../../../data/dog_images/valid", transform=validT
        test_dataset = datasets.ImageFolder("../../../data/dog_images/test", transform=validTes

        #Train, Valid and test loaders with batch sizes
        trainLoader = torch.utils.data.DataLoader(train_dataset,
                                                  batch_size=50,
                                                  shuffle=True,
                                                  num_workers=0)

        validLoader = torch.utils.data.DataLoader(valid_dataset,
                                                  batch_size=50,
                                                  shuffle=True,
                                                  num_workers=0)

        testLoader = torch.utils.data.DataLoader(test_dataset,
                                                 batch_size=25,
                                                 shuffle=False,
                                                 num_workers=0)
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

```
In [17]: ## DONE: Specify model architecture
         # check if CUDA is available
         use_cuda = torch.cuda.is_available()

         # download VGG16 pretrained model
         model_transfer = models.vgg16(pretrained=True)

         # Freeze parameters of the model to avoid brackpropagation
         for param in model_transfer.parameters():
             param.requires_grad = False

         # get the number of dog classes from the train_dataset
         number_of_dog_classes = len(train_dataset.classes)

         # Define dog breed classifier part of model_transfer
         classifier = nn.Sequential(nn.Linear(25088, 4096),
                                    nn.ReLU(),
                                    nn.Dropout(0.5),
                                    nn.Linear(4096, 512),
                                    nn.ReLU(),
                                    nn.Dropout(0.5),
                                    nn.Linear(512, number_of_dog_classes))

         # Rplace the original classifier with the dog breed classifier from above
         model_transfer.classifier = classifier

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

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

Answer: I used the entire feature extractor part of the pre-trained VGG16 model, keeping the weights constant, to avoid overfitting when training on a small new dataset of dog images. This is because the VGG16 network was trained on dogs, among other things, and therefore contains useful high-level feature information in the later convolutional layers.

I then replaced the classifier part of the VGG16 model with my own dog breed classifier. My classifier is modeled along the original VGG16 classifier, with 3 fully connected layers, 2 dropout layers to reduce the number of parameters, and 2 ReLU activations. I changed the number of nodes per fully connected layer to match the number of dog classes we have, i.e. 133.

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [18]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.Adam(model_transfer.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 [19]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             valid_loss_min = np.Inf

             print(f"Batch Size: {loaders['train'].batch_size}\n")

             for epoch in range(1, n_epochs+1):
                 train_loss = 0.0
                 valid_loss = 0.0

                 # train the model
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     optimizer.zero_grad()
                     output = model(data)
                     loss = criterion(output, target)
                     loss.backward()
                     optimizer.step()
                     train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))

                     if (batch_idx + 1) % 5 == 0:
                         print(f'Epoch:{epoch}/{n_epochs} \tBatch:{batch_idx + 1}')
                         print(f'Train Loss: {train_loss}\n')

                 # validate the model
                 model.eval()
                 for batch_idx, (data, target) in enumerate(loaders['valid']):
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     with torch.no_grad():
                         output = model(data)
                     loss = criterion(output, target)
                     valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

```
                    valid_loss
                    ))

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

            # return trained model
            return model

        # define loaders_transfer
        loaders_transfer = {'train': trainLoader,
                            'valid': validLoader,
                            'test': testLoader}
        # train the model
        model_transfer = train(7, loaders_transfer, model_transfer, optimizer_transfer,
                               criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
Batch Size: 50

Epoch:1/7         Batch:5
Train Loss: 7.473165035247803

Epoch:1/7         Batch:10
Train Loss: 6.427736282348633

Epoch:1/7         Batch:15
Train Loss: 5.920994758605957

Epoch:1/7         Batch:20
Train Loss: 5.651684761047363

Epoch:1/7         Batch:25
Train Loss: 5.491879463195801

Epoch:1/7         Batch:30
Train Loss: 5.379464149475098

Epoch:1/7         Batch:35
Train Loss: 5.284078598022461

Epoch:1/7         Batch:40
Train Loss: 5.222708702087402
```

```
Epoch:1/7         Batch:45
Train Loss: 5.168618679046631

Epoch:1/7         Batch:50
Train Loss: 5.124896049499512

Epoch:1/7         Batch:55
Train Loss: 5.073019027709961

Epoch:1/7         Batch:60
Train Loss: 5.047085762023926

Epoch:1/7         Batch:65
Train Loss: 5.018572807312012

Epoch:1/7         Batch:70
Train Loss: 4.994725227355957

Epoch:1/7         Batch:75
Train Loss: 4.966402530670166

Epoch:1/7         Batch:80
Train Loss: 4.926339626312256

Epoch:1/7         Batch:85
Train Loss: 4.89259672164917

Epoch:1/7         Batch:90
Train Loss: 4.852309226989746

Epoch:1/7         Batch:95
Train Loss: 4.82546329498291

Epoch:1/7         Batch:100
Train Loss: 4.791667938232422

Epoch:1/7         Batch:105
Train Loss: 4.757153511047363

Epoch:1/7         Batch:110
Train Loss: 4.721999168395996

Epoch:1/7         Batch:115
Train Loss: 4.690231800079346

Epoch:1/7         Batch:120
Train Loss: 4.652533531188965
```

```
Epoch:1/7         Batch:125
Train Loss: 4.610992431640625

Epoch:1/7         Batch:130
Train Loss: 4.584768295288086

Epoch: 1         Training Loss: 4.564701         Validation Loss: 2.890014
Validation loss decreased (inf --> 2.890014). Saving model...
Epoch:2/7         Batch:5
Train Loss: 3.3415722846984863

Epoch:2/7         Batch:10
Train Loss: 3.417816162109375

Epoch:2/7         Batch:15
Train Loss: 3.492551565170288

Epoch:2/7         Batch:20
Train Loss: 3.5124974250793457

Epoch:2/7         Batch:25
Train Loss: 3.515665054321289

Epoch:2/7         Batch:30
Train Loss: 3.508145809173584

Epoch:2/7         Batch:35
Train Loss: 3.4839823246002197

Epoch:2/7         Batch:40
Train Loss: 3.4687867164611816

Epoch:2/7         Batch:45
Train Loss: 3.439481735229492

Epoch:2/7         Batch:50
Train Loss: 3.3987064361572266

Epoch:2/7         Batch:55
Train Loss: 3.39288592338562

Epoch:2/7         Batch:60
Train Loss: 3.371910572052002

Epoch:2/7         Batch:65
Train Loss: 3.373260498046875
```

```
Epoch:2/7         Batch:70
Train Loss: 3.359954833984375

Epoch:2/7         Batch:75
Train Loss: 3.3467297554016113

Epoch:2/7         Batch:80
Train Loss: 3.3292386531829834

Epoch:2/7         Batch:85
Train Loss: 3.3126864433288574

Epoch:2/7         Batch:90
Train Loss: 3.288783311843872

Epoch:2/7         Batch:95
Train Loss: 3.285830020904541

Epoch:2/7         Batch:100
Train Loss: 3.271442174911499

Epoch:2/7         Batch:105
Train Loss: 3.2624149322509766

Epoch:2/7         Batch:110
Train Loss: 3.249950885772705

Epoch:2/7         Batch:115
Train Loss: 3.2297940254211426

Epoch:2/7         Batch:120
Train Loss: 3.2082014083862305

Epoch:2/7         Batch:125
Train Loss: 3.193521738052368

Epoch:2/7         Batch:130
Train Loss: 3.1779568195343018

Epoch: 2         Training Loss: 3.167957         Validation Loss: 1.706382
Validation loss decreased (2.890014 --> 1.706382). Saving model...
Epoch:3/7         Batch:5
Train Loss: 2.7675814628601074

Epoch:3/7         Batch:10
Train Loss: 2.8597941398620605

Epoch:3/7         Batch:15
```

```
Train Loss: 2.8299574851989746

Epoch:3/7         Batch:20
Train Loss: 2.7647910118103027

Epoch:3/7         Batch:25
Train Loss: 2.789820671081543

Epoch:3/7         Batch:30
Train Loss: 2.7619073390960693

Epoch:3/7         Batch:35
Train Loss: 2.7546546459198

Epoch:3/7         Batch:40
Train Loss: 2.712700605392456

Epoch:3/7         Batch:45
Train Loss: 2.732764959335327

Epoch:3/7         Batch:50
Train Loss: 2.724297523498535

Epoch:3/7         Batch:55
Train Loss: 2.7147977352142334

Epoch:3/7         Batch:60
Train Loss: 2.7189438343048096

Epoch:3/7         Batch:65
Train Loss: 2.6954081058502197

Epoch:3/7         Batch:70
Train Loss: 2.6884374618530273

Epoch:3/7         Batch:75
Train Loss: 2.6907784938812256

Epoch:3/7         Batch:80
Train Loss: 2.6885316371917725

Epoch:3/7         Batch:85
Train Loss: 2.6854262351989746

Epoch:3/7         Batch:90
Train Loss: 2.6782190799713135

Epoch:3/7         Batch:95
```

```
Train Loss: 2.671383857727051

Epoch:3/7         Batch:100
Train Loss: 2.6675028800964355

Epoch:3/7         Batch:105
Train Loss: 2.6593399047851562

Epoch:3/7         Batch:110
Train Loss: 2.6507017612457275

Epoch:3/7         Batch:115
Train Loss: 2.641123056411743

Epoch:3/7         Batch:120
Train Loss: 2.6378297805786133

Epoch:3/7         Batch:125
Train Loss: 2.6392085552215576

Epoch:3/7         Batch:130
Train Loss: 2.636779308319092

Epoch: 3         Training Loss: 2.635740         Validation Loss: 1.530334
Validation loss decreased (1.706382 --> 1.530334). Saving model...
Epoch:4/7         Batch:5
Train Loss: 2.49190092086792

Epoch:4/7         Batch:10
Train Loss: 2.4266843795776367

Epoch:4/7         Batch:15
Train Loss: 2.3798322677612305

Epoch:4/7         Batch:20
Train Loss: 2.382491111755371

Epoch:4/7         Batch:25
Train Loss: 2.3802330493927

Epoch:4/7         Batch:30
Train Loss: 2.415888786315918

Epoch:4/7         Batch:35
Train Loss: 2.4167251586914062

Epoch:4/7         Batch:40
Train Loss: 2.4100680351257324
```

```
Epoch:4/7         Batch:45
Train Loss: 2.4154460430145264

Epoch:4/7         Batch:50
Train Loss: 2.4182653427124023

Epoch:4/7         Batch:55
Train Loss: 2.410555839538574

Epoch:4/7         Batch:60
Train Loss: 2.401998996734619

Epoch:4/7         Batch:65
Train Loss: 2.402604818344116

Epoch:4/7         Batch:70
Train Loss: 2.4022269248962402

Epoch:4/7         Batch:75
Train Loss: 2.398056983947754

Epoch:4/7         Batch:80
Train Loss: 2.385394334793091

Epoch:4/7         Batch:85
Train Loss: 2.3765223026275635

Epoch:4/7         Batch:90
Train Loss: 2.376295804977417

Epoch:4/7         Batch:95
Train Loss: 2.3816099166870117

Epoch:4/7         Batch:100
Train Loss: 2.377159357070923

Epoch:4/7         Batch:105
Train Loss: 2.3785629272460938

Epoch:4/7         Batch:110
Train Loss: 2.376589298248291

Epoch:4/7         Batch:115
Train Loss: 2.373541831970215

Epoch:4/7         Batch:120
Train Loss: 2.3774352073669434
```

```
Epoch:4/7         Batch:125
Train Loss: 2.375803232192993

Epoch:4/7         Batch:130
Train Loss: 2.3734242916107178

Epoch: 4         Training Loss: 2.368410         Validation Loss: 1.359561
Validation loss decreased (1.530334 --> 1.359561). Saving model...
Epoch:5/7         Batch:5
Train Loss: 2.152315378189087

Epoch:5/7         Batch:10
Train Loss: 2.1770079135894775

Epoch:5/7         Batch:15
Train Loss: 2.180626392364502

Epoch:5/7         Batch:20
Train Loss: 2.2095515727996826

Epoch:5/7         Batch:25
Train Loss: 2.240825653076172

Epoch:5/7         Batch:30
Train Loss: 2.2436330318450928

Epoch:5/7         Batch:35
Train Loss: 2.231349229812622

Epoch:5/7         Batch:40
Train Loss: 2.2068512439727783

Epoch:5/7         Batch:45
Train Loss: 2.2228007316589355

Epoch:5/7         Batch:50
Train Loss: 2.2011702060699463

Epoch:5/7         Batch:55
Train Loss: 2.2030136585235596

Epoch:5/7         Batch:60
Train Loss: 2.223008394241333

Epoch:5/7         Batch:65
Train Loss: 2.2180533409118652
```

```
Epoch:5/7         Batch:70
Train Loss: 2.222964286804199

Epoch:5/7         Batch:75
Train Loss: 2.2249808311462402

Epoch:5/7         Batch:80
Train Loss: 2.2310478687286377

Epoch:5/7         Batch:85
Train Loss: 2.2247910499572754

Epoch:5/7         Batch:90
Train Loss: 2.233812093734741

Epoch:5/7         Batch:95
Train Loss: 2.228811740875244

Epoch:5/7         Batch:100
Train Loss: 2.2243778705596924

Epoch:5/7         Batch:105
Train Loss: 2.2138712406158447

Epoch:5/7         Batch:110
Train Loss: 2.2144253253936768

Epoch:5/7         Batch:115
Train Loss: 2.2139172554016113

Epoch:5/7         Batch:120
Train Loss: 2.2128326892852783

Epoch:5/7         Batch:125
Train Loss: 2.216679811477661

Epoch:5/7         Batch:130
Train Loss: 2.215333938598633

Epoch: 5         Training Loss: 2.216769         Validation Loss: 1.240154
Validation loss decreased (1.359561 --> 1.240154). Saving model...
Epoch:6/7         Batch:5
Train Loss: 1.8776848316192627

Epoch:6/7         Batch:10
Train Loss: 2.056091785430908

Epoch:6/7         Batch:15
```

```
Train Loss: 2.0701558589935303

Epoch:6/7         Batch:20
Train Loss: 2.137758731842041

Epoch:6/7         Batch:25
Train Loss: 2.1217424869537354

Epoch:6/7         Batch:30
Train Loss: 2.1479642391204834

Epoch:6/7         Batch:35
Train Loss: 2.123241424560547

Epoch:6/7         Batch:40
Train Loss: 2.097996234893799

Epoch:6/7         Batch:45
Train Loss: 2.1038591861724854

Epoch:6/7         Batch:50
Train Loss: 2.1048331260681152

Epoch:6/7         Batch:55
Train Loss: 2.0737812519073486

Epoch:6/7         Batch:60
Train Loss: 2.0631468296051025

Epoch:6/7         Batch:65
Train Loss: 2.05387806892395

Epoch:6/7         Batch:70
Train Loss: 2.058842420578003

Epoch:6/7         Batch:75
Train Loss: 2.059363842010498

Epoch:6/7         Batch:80
Train Loss: 2.0745582580566406

Epoch:6/7         Batch:85
Train Loss: 2.0764663219451904

Epoch:6/7         Batch:90
Train Loss: 2.0859768390655518

Epoch:6/7         Batch:95
```

```
Train Loss: 2.0837745666503906

Epoch:6/7         Batch:100
Train Loss: 2.092078685760498

Epoch:6/7         Batch:105
Train Loss: 2.1014389991760254

Epoch:6/7         Batch:110
Train Loss: 2.1026179790496826

Epoch:6/7         Batch:115
Train Loss: 2.0982892513275146

Epoch:6/7         Batch:120
Train Loss: 2.0999858379364014

Epoch:6/7         Batch:125
Train Loss: 2.1047916412353516

Epoch:6/7         Batch:130
Train Loss: 2.1089160442352295

Epoch: 6         Training Loss: 2.114678         Validation Loss: 1.158162
Validation loss decreased (1.240154 --> 1.158162). Saving model...
Epoch:7/7         Batch:5
Train Loss: 2.021800994873047

Epoch:7/7         Batch:10
Train Loss: 1.8951289653778076

Epoch:7/7         Batch:15
Train Loss: 1.8996362686157227

Epoch:7/7         Batch:20
Train Loss: 1.8472721576690674

Epoch:7/7         Batch:25
Train Loss: 1.8523635864257812

Epoch:7/7         Batch:30
Train Loss: 1.8583946228027344

Epoch:7/7         Batch:35
Train Loss: 1.8631305694580078

Epoch:7/7         Batch:40
Train Loss: 1.9357380867004395
```

```
Epoch:7/7         Batch:45
Train Loss: 1.9273287057876587

Epoch:7/7         Batch:50
Train Loss: 1.9523977041244507

Epoch:7/7         Batch:55
Train Loss: 1.974427342414856

Epoch:7/7         Batch:60
Train Loss: 1.984613299369812

Epoch:7/7         Batch:65
Train Loss: 1.9787983894348145

Epoch:7/7         Batch:70
Train Loss: 1.9689295291900635

Epoch:7/7         Batch:75
Train Loss: 1.9635522365570068

Epoch:7/7         Batch:80
Train Loss: 1.9627892971038818

Epoch:7/7         Batch:85
Train Loss: 1.9725959300994873

Epoch:7/7         Batch:90
Train Loss: 1.9717820882797241

Epoch:7/7         Batch:95
Train Loss: 1.9723604917526245

Epoch:7/7         Batch:100
Train Loss: 1.9758775234222412

Epoch:7/7         Batch:105
Train Loss: 1.9651567935943604

Epoch:7/7         Batch:110
Train Loss: 1.9529215097427368

Epoch:7/7         Batch:115
Train Loss: 1.9623535871505737

Epoch:7/7         Batch:120
Train Loss: 1.963469386100769
```

```
Epoch:7/7         Batch:125
Train Loss: 1.9697654247283936

Epoch:7/7         Batch:130
Train Loss: 1.9767730236053467

Epoch: 7         Training Loss: 1.970680         Validation Loss: 1.072272
Validation loss decreased (1.158162 --> 1.072272). 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 [20]: 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()
                 # 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
                 output = F.softmax(output, dim=1)
                 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))
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

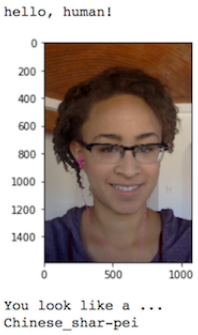
```
Test Loss: 1.125920
```

```
Test Accuracy: 66% (552/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [21]: ### DONE: 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 train_dataset.classes]
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))

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

             # Define normalization step for image
             normalize = transforms.Normalize(mean=(0.485, 0.456, 0.406),
                                              std=(0.229, 0.224, 0.225))

             # Define transformations of image
             preprocess = transforms.Compose([transforms.Resize(258),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              normalize])

             # Preprocess image to 4D Tensor (.unsqueeze(0) adds a dimension)
             img_tensor = preprocess(img).unsqueeze_(0)

             # Move tensor to GPU if available
             if use_cuda:
                 img_tensor = img_tensor.cuda()

             ## Inference
             # Turn on evaluation mode
             model_transfer.eval()

             # Get predicted category for image
             with torch.no_grad():
                 output = model_transfer(img_tensor)
                 prediction = torch.argmax(output).item()

             # Turn off evaluation mode
             model_transfer.train()
```



Sample Human Output

```
    # Use prediction to get dog breed
    breed = class_names[prediction]

    return breed
```

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 [24]:
```
### DONE: 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
    if face_detector(img_path):
        print('Hello Human!')
        plt.imshow(Image.open(img_path))
        plt.show()
        print(f'You look like a {predict_breed_transfer(img_path)}')
        print('\n=====================================\n')
    elif dog_detector(img_path):
        plt.imshow(Image.open(img_path))
        plt.show()
        print(f'This is a picture of a ... {predict_breed_transfer(img_path)}')
```

```
        print('\n=======================================\n')
    else:
        plt.imshow(Image.open(img_path))
        plt.show()
        print('The picture is neither a dog or a human')
        print('\n=======================================\n')
```

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!

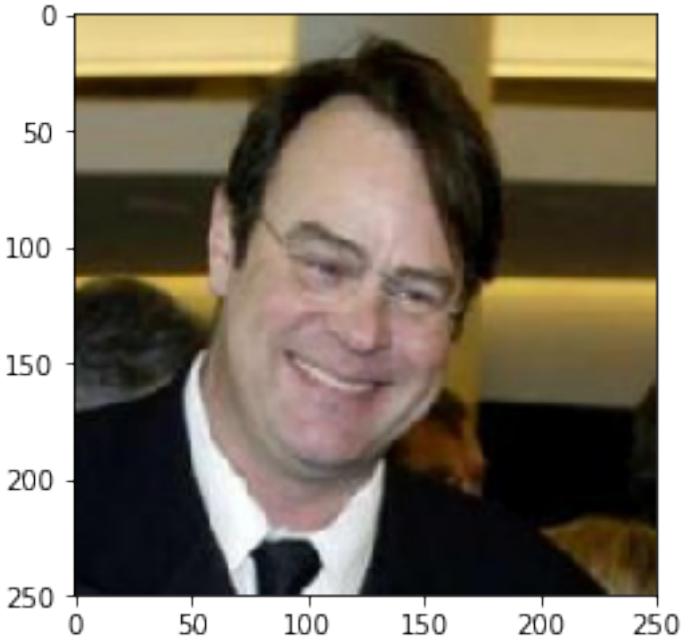
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

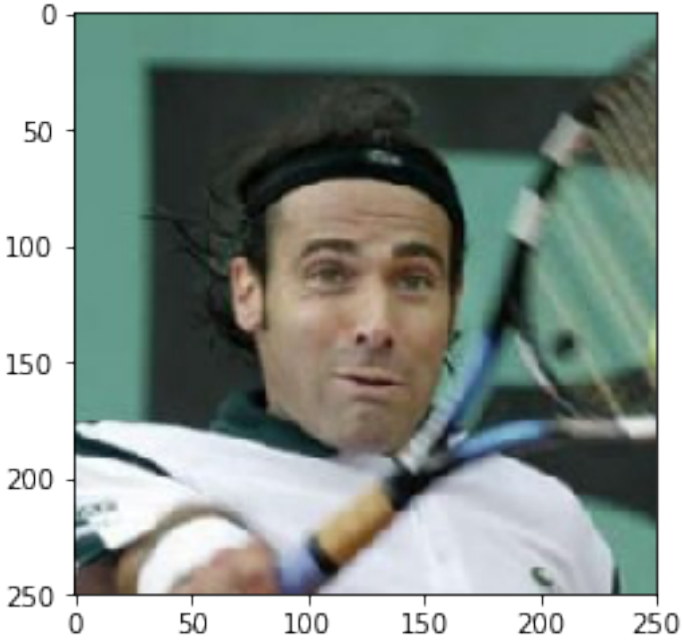
Answer: (Three possible points for improvement) * Improve the face detector algorithm by building a new neural network using transfer learning and the VGG16 network. * Use transfer learning on the dog detector to increase accuracy. * Increase the accuracy of the dog breed predictor by increasing the number of training episodes and possibly using a better feature detector, such as the one from the InceptionV3 network.

In [25]:
```
## DONE: 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)
```

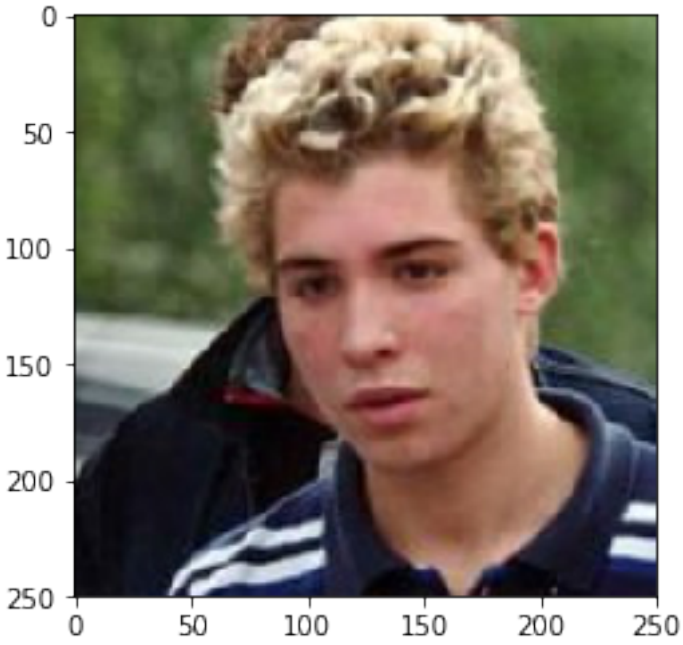
```
Hello Human!
```




```
You look like a Nova scotia duck tolling retriever

=======================================

Hello Human!
```

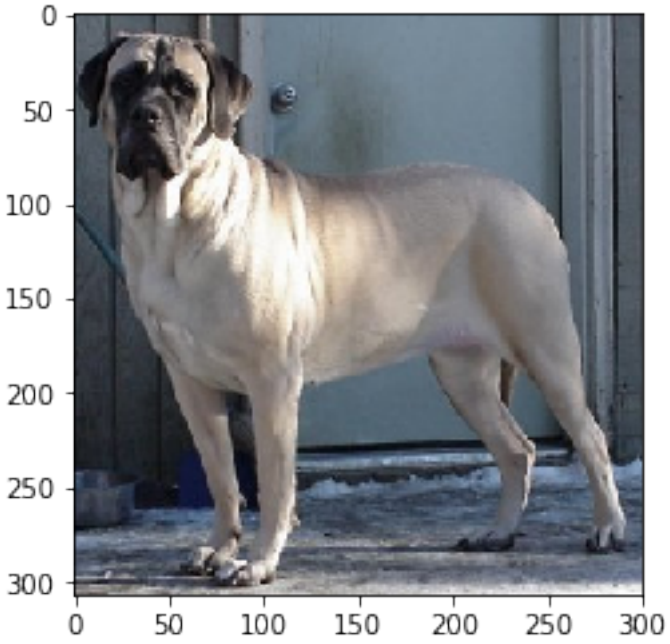


```
You look like a Nova scotia duck tolling retriever

=======================================

Hello Human!
```

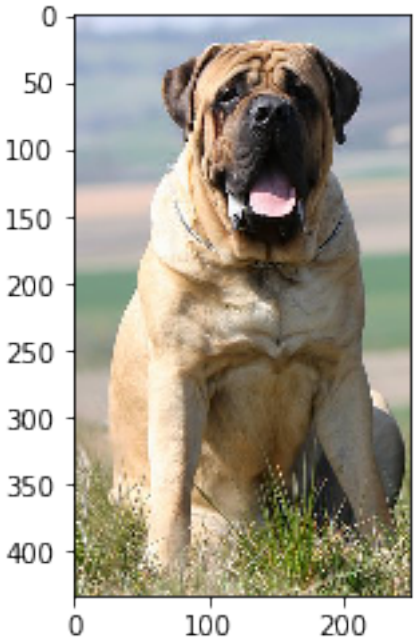



```
You look like a Kerry blue terrier

=======================================
```


```
This is a picture of a ... Bullmastiff

========================================
```

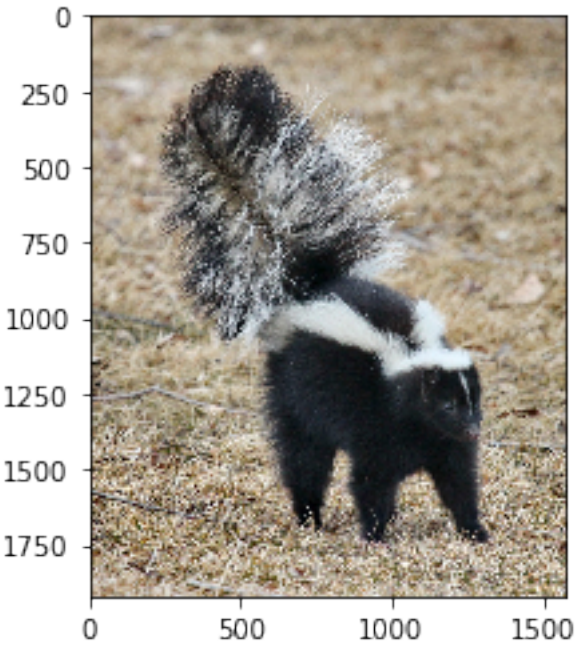


```
This is a picture of a ... Mastiff

========================================
```

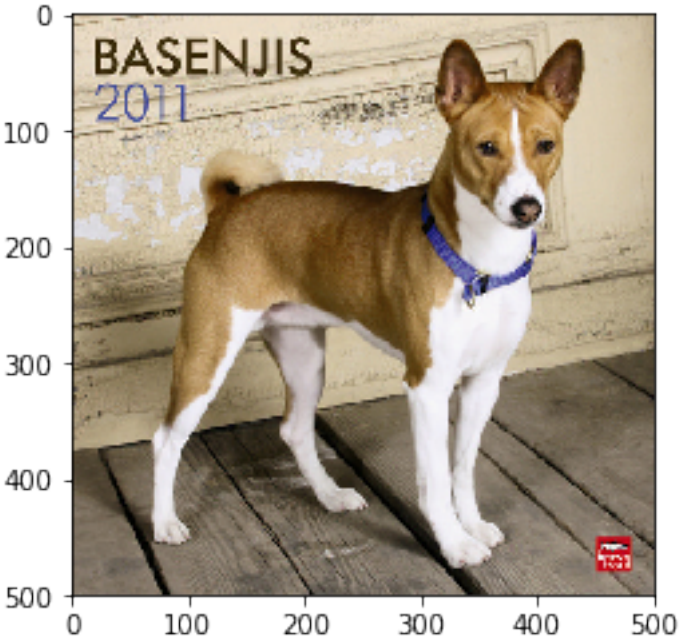



```
This is a picture of a ... Mastiff

========================================
```

```
In [27]: ## Execute algorithm with 6 test images
         files = np.array(glob("test_images/*"))
         for file_path in files:
             run_app(file_path)
```

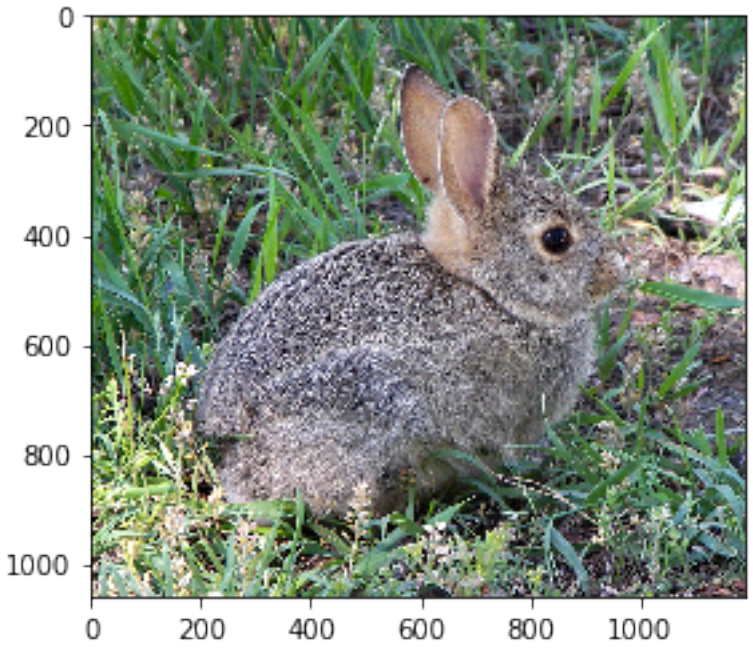



```
The picture is neither a dog or a human

=======================================
```

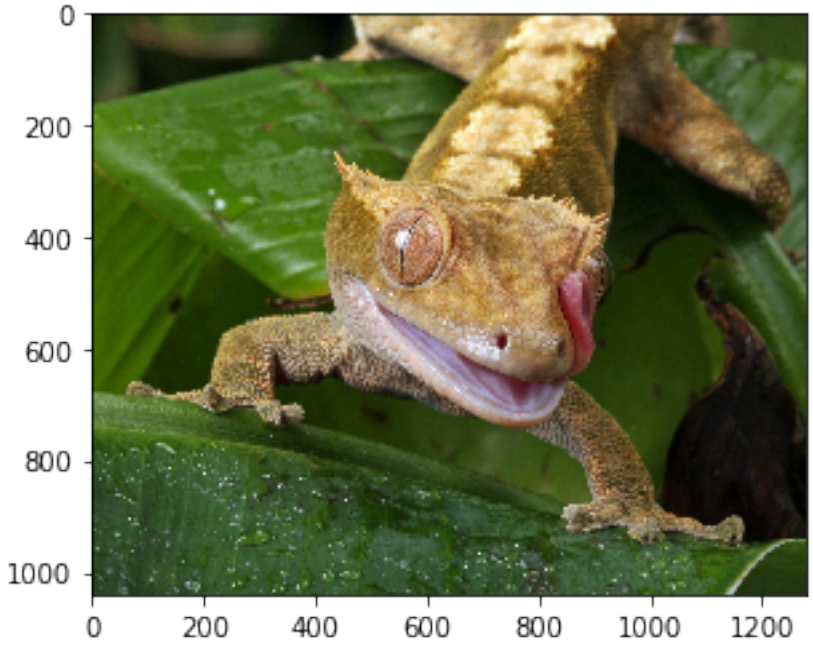


```
This is a picture of a ... Basenji

========================================
```


```
This is a picture of a ... Keeshond

========================================
```




```
The picture is neither a dog or a human

========================================
```



```
The picture is neither a dog or a human

=======================================
```


```
The picture is neither a dog or a human

=======================================
```

In [ ]: