dog_app

February 13, 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 [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

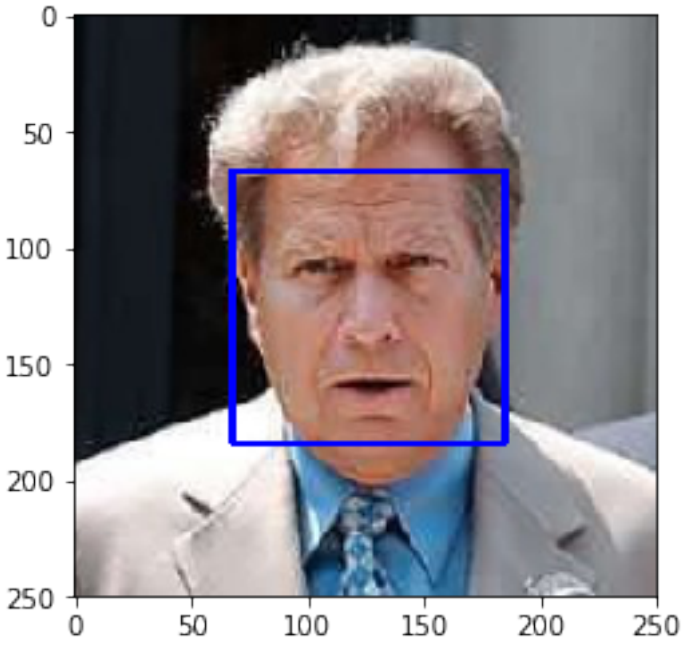
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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[9846])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

98% Percentage of human face have been detected in human_files
17% Percentage of the Human faces have been detected in Dog_files

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

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

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

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

        # Human Faces!!
        for faces in human_files_short:
            human_face_percent.append(face_detector(faces))
            percent = np.sum(human_face_percent)
        print ('{}% Percentage of human face have been detected in human_files   '.format(percen

        # For dogs_files
        for dogs in dog_files_short:
            dog_found.append(face_detector(dogs))
            d_percent = np.sum(dog_found)
```

```
        print ('{}% Percentage of the Human faces have been detected in Dog_files  '.format(d_pe
```

```
98% Percentage of human face have been detected in human_files
17% Percentage of the Human faces have been detected in Dog_files
```

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 [ ]: ### (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 [5]: 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()
        # Print the model
        print (VGG16)
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:30<00:00, 17887801.99it/s]
```

```
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=1000, bias=True)
  )
)
```

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

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

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

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

```
In [6]: from PIL import Image
        import torchvision.transforms as transforms
        from torchvision import datasets
        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

            transform = transforms.Compose([transforms.Resize(224),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor()
                                            ])
            def image_load(img_path):
                """ Load Image __ Return Tensor"""
                #Open the Image "Using Image.open"
                image = Image.open(img_path)
                #Transform the image "Define above"
                image = transform(image).float()
                image = image.unsqueeze(0)
                if torch.cuda.is_available():
                    image = image.cuda()
                return image

            # Move the images to cuda


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

```
    out = VGG16(image_load(img_path))
    pred = torch.argmax(out)

    ''' Hello Mr. Reviewer can you teach me the difference between torch.argmax and tour

   # _, preds = torch.max(out, 1)
    #print (_)

    return pred
# predicted class index

#VGG16_predict('/data/dog_images/train/001.Affenpinscher/Affenpinscher_00001.jpg')
#
```

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 [7]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    pred = VGG16_predict(img_path)

    #Compare if dog the return true else false

    def compare(output):
         if output >= 151 and output < 269:
            return True
         else:
            return False


    return compare(pred) # true/false
#
```

In [8]:
```
# Tests for Dogs
print (dog_detector('/data/dog_images/train/005.Alaskan_malamute/Alaskan_malamute_00300.
print (dog_detector('/data/dog_images/train/084.Icelandic_sheepdog/Icelandic_sheepdog_05
print (dog_detector('/data/dog_images/train/004.Akita/Akita_00269.jpg'))
print (dog_detector('/data/dog_images/train/026.Black_russian_terrier/Black_russian_terr
print (dog_detector('/data/dog_images/train/003.Airedale_terrier/Airedale_terrier_00154.
```

```
True
True
True
True
True
```

```
In [9]: ### Test for Humans
        print(dog_detector('/data/lfw/Adam_Sandler/Adam_Sandler_0002.jpg')) #LOL :D
        print (dog_detector('/data/lfw/Anil_Ramsook/Anil_Ramsook_0001.jpg')) # No idea :P
        print (dog_detector('/data/lfw/Bernadette_Peters/Bernadette_Peters_0001.jpg'))
```

```
False
False
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:
0% percentage of the images in human_files_short have a detected dog
95% percentage of the images in dog_files_short have a detected dog

```
In [10]: ### TODO: Test the performance of the dog_detector function!! Let's set the Bar high!!


         # For Humans!! Let's Do this!!
         percent_h =[]
         for human in human_files_short:
             percent_h.append(dog_detector(human))
         print ('{}% percentage of the images in human_files_short have a detected dog'.format(n



         ### on the images in human_files_short and dog_files_short.


         # For Dogs!! Who let the dogs out!!

         percent_d = []
         for dog_ in dog_files_short:
             percent_d.append(dog_detector(dog_))
         print('{}% percentage of the images in dog_files_short have a detected dog'.format(np.s
```

```
0% percentage of the images in human_files_short have a detected dog
94% percentage of the images in dog_files_short have a detected dog
```

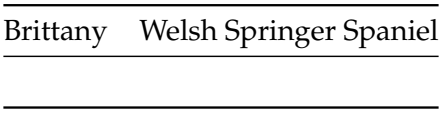
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

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

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

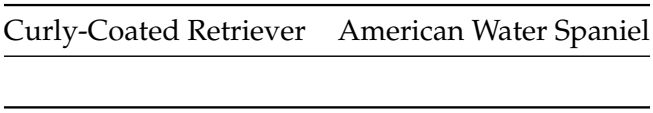
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

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



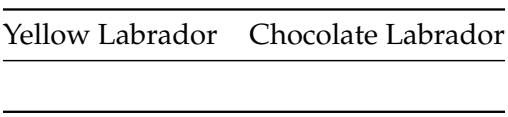
Brittany Welsh Springer Spaniel

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



Curly-Coated Retriever American Water Spaniel

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



Yellow Labrador Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

In [11]:
```
import os
from torchvision import datasets
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
# Variables

batch_sizes = 32
num_worker = 0

transform = transforms.Compose([transforms.Resize(224),
                                transforms.CenterCrop(224),
                                transforms.RandomRotation(30),
                                transforms.RandomHorizontalFlip(),
                                transforms.ToTensor(),
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))
                               ])
#Training Image set
train_set = datasets.ImageFolder('/data/dog_images/train/', transform = transform)

# Validation Image Set
vaild_set = datasets.ImageFolder('/data/dog_images/valid', transform = transform)

# Test Image set
test_set = datasets.ImageFolder('/data/dog_images/test', transform = transform)

#Loaders
#Train_loader
train_loader = torch.utils.data.DataLoader(train_set, batch_size = batch_sizes, shuffle

#valid_loader
valid_loader = torch.utils.data.DataLoader(vaild_set, batch_size= batch_sizes, shuffle

#Test_loader
test_loader = torch.utils.data.DataLoader(test_set, batch_size = batch_sizes, shuffle =
```

In [ ]:
```
#Vizulize the data
#Test for Loader
```

```
images, labels = next(iter(train_loader))
images = images.numpy()
#Plot the figures
fig = plt.figure(figsize=(25, 4))
for i in np.arange(20):
    ax =fig.add_subplot(2, 20/2, i+1, xticks=[], yticks=[])
   # images = images / 2 + 0.5 # unnormalize
    plt.imshow(np.transpose(images[i], (1, 2, 0)))
    #set the title
  #  ax.set_title(labels[i])# Classes not defined
```

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:

- How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

---> I used to transforms.Resize(224), transforms.CenterCrop(224) to crop & shape the image in the size of 224 x 224.

- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

---> **Yes!!** By using transforms.RandomRotation(30),transforms.RandomHorizontalFlip(). It helps your model learn better.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 #Convolutional Layers
                 #Sees 224x224x3 Input Image
                 self.conv1 = nn.Conv2d(3, 32, 3, stride = 2, padding = 1)
                 #Sees 112x112x32
                 self.conv2 = nn.Conv2d(32, 64, 3,stride = 2, padding = 1)
                 #Sees 56x56x64
```

```
            self.conv3 = nn.Conv2d(64, 128, 3, padding = 1)
            #Sees 28x28x128
          #Pooling layers
            self.pool = nn.MaxPool2d(2, 2)
            breeds = 133
          #Fully Connected Layers
            self.fc1 = nn.Linear(128 * 7 * 7 , 1024)
            self.fc2 = nn.Linear(1024, breeds )


        #Droup probability of 25%
            self.dropout = nn.Dropout(0.25)
        def forward(self, x):
            ## Define forward behavior
            x = F.relu(self.conv1(x))
            x = self.pool(x)
            x = F.relu(self.conv2(x))
            x = self.pool(x)

            x = F.relu(self.conv3(x))
            x = self.pool(x)
            #Flatten the Image
            x = x.view(-1, 128 *7 * 7)

            #To the Fully Conected layer
            x = self.dropout(x)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = self.fc2(x)

            return x

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

    # instantiate the CNN
    model_scratch = Net()

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

```
In [13]: print (model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
  (fc1): Linear(in_features=6272, out_features=1024, bias=True)
  (fc2): 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:

I decide to use CNN model with three Convolutional layers with Relu activation fucntion and a pooling layer attached to it after every pass.

After passing through the convoultional layer it is flattened and transfered to the fully connected layers. With 'FC1' 128 * 7 * 7 input_layer and 1024 output layers, Relu activation function followed by a dropout layer (p=0.25) and 'FC2' have 1024 input_layer connected to previous layer and 133 output_layer according to the classes.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

In [14]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.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 [15]:
```
def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    valid_loss_min = np.Inf

    for epoch in range(1, n_epochs+1):
        # 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(loader[0]):
```

```
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            optimizer_scratch.zero_grad()
            output = model_scratch(data)
            loss = criterion_scratch(output, target)
            loss.backward()
            optimizer_scratch.step()
            train_loss += ((1 / (batch_idx + 1)) * (loss.data - train_loss))
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loader[1]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            out = model_scratch(data)
            vloss = criterion_scratch(out, target)
            valid_loss +=  ((1 / (batch_idx + 1)) * (vloss.data - valid_loss))
            ## update the average validation loss


        # 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 had decreased {:.6f} -->  {:.6f}. Saving Current Mo
            torch.save(model_scratch.state_dict(), 'model_scratch.pt')
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
loader = [train_loader, valid_loader]
model_scratch = train(15, loader, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.782468         Validation Loss: 4.614245
Validation Loss had decreased inf -->  4.614245. Saving Current Model..
Epoch: 2         Training Loss: 4.415905         Validation Loss: 4.315432
Validation Loss had decreased 4.614245 -->  4.315432. Saving Current Model..
Epoch: 3         Training Loss: 4.159312         Validation Loss: 4.141236
Validation Loss had decreased 4.315432 -->  4.141236. Saving Current Model..
Epoch: 4         Training Loss: 3.967436         Validation Loss: 3.981108
Validation Loss had decreased 4.141236 -->  3.981108. Saving Current Model..
Epoch: 5         Training Loss: 3.788776         Validation Loss: 3.893961
Validation Loss had decreased 3.981108 -->  3.893961. Saving Current Model..
Epoch: 6         Training Loss: 3.631158         Validation Loss: 3.883883
Validation Loss had decreased 3.893961 -->  3.883883. Saving Current Model..
Epoch: 7         Training Loss: 3.494131         Validation Loss: 3.937231
Epoch: 8         Training Loss: 3.359942         Validation Loss: 3.780471
Validation Loss had decreased 3.883883 -->  3.780471. Saving Current Model..
Epoch: 9         Training Loss: 3.208518         Validation Loss: 3.774253
Validation Loss had decreased 3.780471 -->  3.774253. Saving Current Model..
Epoch: 10         Training Loss: 3.066980         Validation Loss: 3.747632
Validation Loss had decreased 3.774253 -->  3.747632. Saving Current Model..
Epoch: 11         Training Loss: 2.950228         Validation Loss: 3.829823
Epoch: 12         Training Loss: 2.808523         Validation Loss: 3.814773
Epoch: 13         Training Loss: 2.683963         Validation Loss: 3.845926
Epoch: 14         Training Loss: 2.562822         Validation Loss: 3.841945
Epoch: 15         Training Loss: 2.459389         Validation Loss: 3.930493
```

```
In [16]: # 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 [17]: def test(loaders, model, criterion, use_cuda):

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

             model.eval()
             for batch_idx, (data, target) in enumerate(loaders[0]):
                 # 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
    loader = [test_loader]
    test(loader, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.719241


Test Accuracy: 14% (121/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

```
In [18]: ## TODO: Specify data loaders
         batch_sizes = 32
         num_worker = 0

         transform = transforms.Compose([transforms.Resize(224),
                                         transforms.CenterCrop(224),
                                         transforms.RandomRotation(30),

                                         transforms.RandomHorizontalFlip(p=0.2),
                                         transforms.ToTensor(),
```

```
                                transforms.Normalize((0.485, 0.456, 0.406),
                                                     (0.229, 0.224, 0.225))
                                ])
        #Training Image set
        train_set_tran = datasets.ImageFolder('/data/dog_images/train/', transform = transform)

        # Validation Image Set
        vaild_set_tran = datasets.ImageFolder('/data/dog_images/valid', transform = transform)

        # Test Image set
        test_set_tran = datasets.ImageFolder('/data/dog_images/test', transform = transform)

        #Loaders
        #Train_loader
        train_loader_tran = torch.utils.data.DataLoader(train_set_tran, batch_size = batch_size

        #valid_loader
        valid_loader_tran = torch.utils.data.DataLoader(vaild_set_tran, batch_size= batch_sizes

        #Test_loader
        test_loader_tran = torch.utils.data.DataLoader(test_set_tran, batch_size = batch_sizes,
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

In [19]:
```
import torchvision.models as models
import torch.nn as nn

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

input_layer = model_transfer.fc.in_features
output_layer = 133




#Freeze the parameters in Resnet!! Okay!!

for prams in model_transfer.parameters():
    prams.requires_grad = False

#define the custom layer for resnet
fc_custom = nn.Linear(input_layer, output_layer)
```

```
        #Apply the custom layer

        model_transfer.fc = fc_custom

        #move model to GPU!!
        if use_cuda:
            model_transfer = model_transfer.cuda()
```

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 71796408.28it/s]
```

In [20]:

```
        #Unfreeze the custom layer to adapt weights!!
        for prams in model_transfer.fc.parameters():
            prams.requires_grad = True
        print (model_transfer)
```

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

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

```
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

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

```
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

```
In [21]: print (model_transfer.fc)
```

```
Linear(in_features=2048, out_features=133, bias=True)
```

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

Answer:

It used ResNet50 as my model and replaced the fc layer with my custom made layer suitted for the given task. Freezed all the parameters and unfreezed the custom parameters.

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 [22]: criterion_transfer = nn.CrossEntropyLoss()

         ResNetParms = model_transfer.fc.parameters()

         optimizer_transfer = optim.Adam(ResNetParms, 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 [23]: # train the model
         loaders = [train_loader_tran, valid_loader_tran]

         def train_transfer(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[0]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## find the loss and update the model parameters accordingly
            optimizer_transfer.zero_grad()
            output = model_transfer(data)
            loss = criterion_transfer(output, target)
            loss.backward()
            optimizer_transfer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
            ## record the average training loss, using something like
            ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders[1]):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            out = model_transfer(data)
            vloss = criterion_transfer(out, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (vloss.data - valid_loss
            ## update the average validation loss


        # 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 had decreased {:.6f} -->  {:.6f}. Saving Current Mo
            torch.save(model_transfer.state_dict(), 'model_trasfer.pt')
            valid_loss_min = valid_loss

    # return trained model
    return model
```

```
        model_transfer =  train_transfer(25, loaders, model_transfer, optimizer_transfer, crite

        # 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: 2.328061         Validation Loss: 1.025582
Validation Loss had decreased inf -->  1.025582. Saving Current Model..
Epoch: 2         Training Loss: 0.879667         Validation Loss: 0.761514
Validation Loss had decreased 1.025582 -->  0.761514. Saving Current Model..
Epoch: 3         Training Loss: 0.642100         Validation Loss: 0.655831
Validation Loss had decreased 0.761514 -->  0.655831. Saving Current Model..
Epoch: 4         Training Loss: 0.561082         Validation Loss: 0.660429
Epoch: 5         Training Loss: 0.490589         Validation Loss: 0.633445
Validation Loss had decreased 0.655831 -->  0.633445. Saving Current Model..
Epoch: 6         Training Loss: 0.443461         Validation Loss: 0.597901
Validation Loss had decreased 0.633445 -->  0.597901. Saving Current Model..
Epoch: 7         Training Loss: 0.423007         Validation Loss: 0.604735
Epoch: 8         Training Loss: 0.377476         Validation Loss: 0.620995
Epoch: 9         Training Loss: 0.360276         Validation Loss: 0.602939
Epoch: 10         Training Loss: 0.335009         Validation Loss: 0.614755
Epoch: 11         Training Loss: 0.338916         Validation Loss: 0.583603
Validation Loss had decreased 0.597901 -->  0.583603. Saving Current Model..
Epoch: 12         Training Loss: 0.323460         Validation Loss: 0.631725
Epoch: 13         Training Loss: 0.291297         Validation Loss: 0.617680
Epoch: 14         Training Loss: 0.295719         Validation Loss: 0.771904
Epoch: 15         Training Loss: 0.282835         Validation Loss: 0.684667
Epoch: 16         Training Loss: 0.261980         Validation Loss: 0.643550
Epoch: 17         Training Loss: 0.268791         Validation Loss: 0.644148
Epoch: 18         Training Loss: 0.256880         Validation Loss: 0.640594
Epoch: 19         Training Loss: 0.251191         Validation Loss: 0.667173
Epoch: 20         Training Loss: 0.235856         Validation Loss: 0.648816
Epoch: 21         Training Loss: 0.241209         Validation Loss: 0.598467
Epoch: 22         Training Loss: 0.237662         Validation Loss: 0.662421
Epoch: 23         Training Loss: 0.224308         Validation Loss: 0.702431
Epoch: 24         Training Loss: 0.213824         Validation Loss: 0.704913
Epoch: 25         Training Loss: 0.211493         Validation Loss: 0.671364
```

```
In [24]: # load the model that got the best validation accuracy
         model_transfer.load_state_dict(torch.load('model_trasfer.pt'))
```

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 [25]: loaders_transfer = [train_loader_tran]
         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.204092


Test Accuracy: 93% (6248/6680)
```

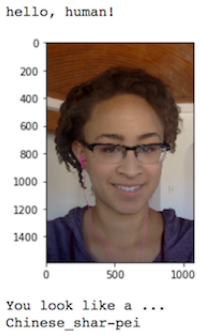
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

```
In [26]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.

         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in train_set_tran.classes]
         # Read Images
         def image_read(img_path):
                 image = Image.open(img_path)
                 image = transform(image)
                 image = image.unsqueeze(0)
                 if use_cuda:
                     image = image.cuda()
                 return image

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

             model_transfer.eval()
             out = model_transfer(image_read(img_path))
             preds = torch.argmax(out)
             return class_names[preds]
```

```
In [27]: print (predict_breed_transfer('/data/dog_images/train/001.Affenpinscher/Affenpinscher_0
         print (predict_breed_transfer('/data/dog_images/train/039.Bull_terrier/Bull_terrier_027
         print (predict_breed_transfer('/data/dog_images/valid/113.Old_english_sheepdog/Old_engl
         print (predict_breed_transfer('/data/dog_images/test/068.Flat-coated_retriever/Flat-coa
```

```
Affenpinscher
Bull terrier
Old english sheepdog
Flat-coated retriever
```



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.

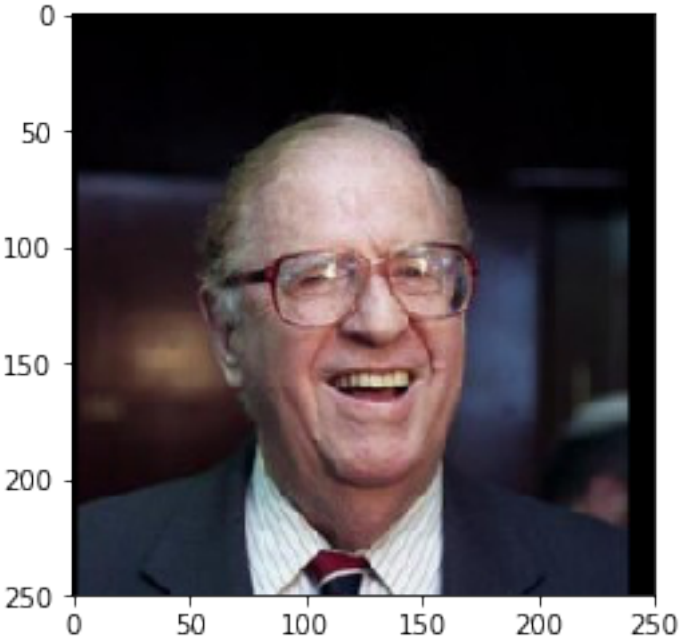
You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

Some sample output for our algorithm is provided below, but feel free to design your own user experience!

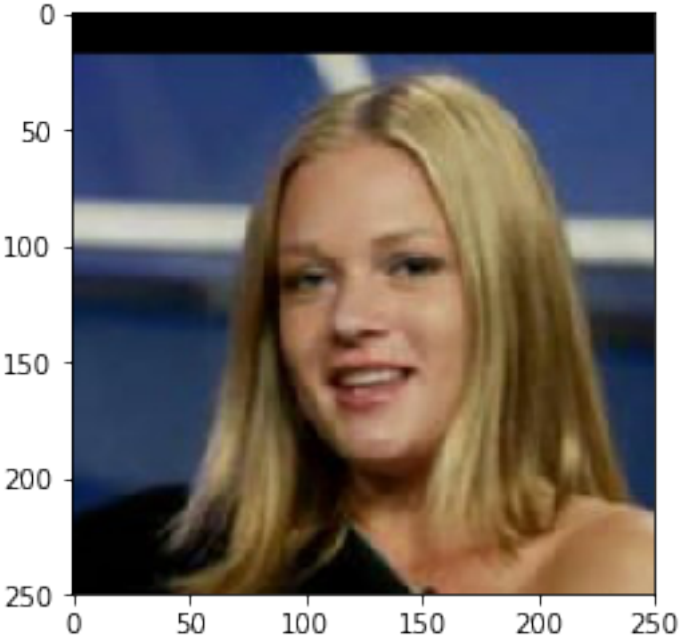
1.1.18 (IMPLEMENTATION) Write your Algorithm

In [32]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):

    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path) == True:
        pred = predict_breed_transfer (img_path)
        print ('It is a dog.  \nIts predicted breed is... {}'.format (pred))
    elif face_detector(img_path) > 0:
        pred = predict_breed_transfer (img_path)
        print ('Hi, Human \nYou look like... \n{}'.format(pred))
    else:
        print ('Sorry!! I dont know this.')
    ## handle cases for a human face, dog, and neither
```

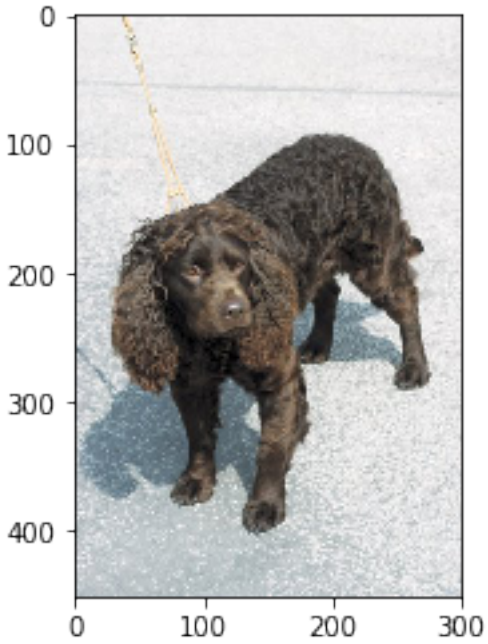
In [33]:
```
print (run_app('/data/lfw/Abba_Eban/Abba_Eban_0001.jpg'))
print (run_app('/data/lfw/AJ_Cook/AJ_Cook_0001.jpg'))
print (run_app('/data/dog_images/test/009.American_water_spaniel/American_water_spaniel
```




```
Hi, Human
You look like...
American water spaniel
None
```



```
Hi, Human
You look like...
Silky terrier
None
```




```
It is a dog.
Its predicted breed is... American water spaniel
None
```

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.

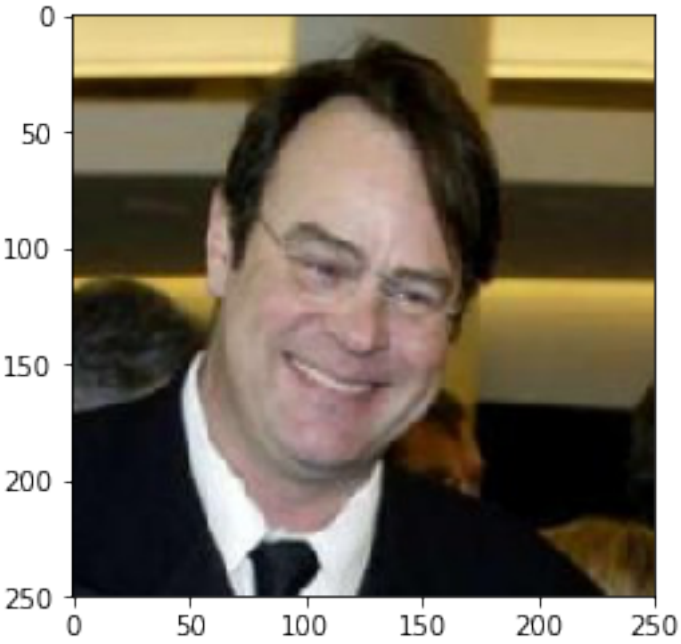
Answer: (Three possible points for improvement)

1) Can augment the data, and more data can definately help improve the accuracy.

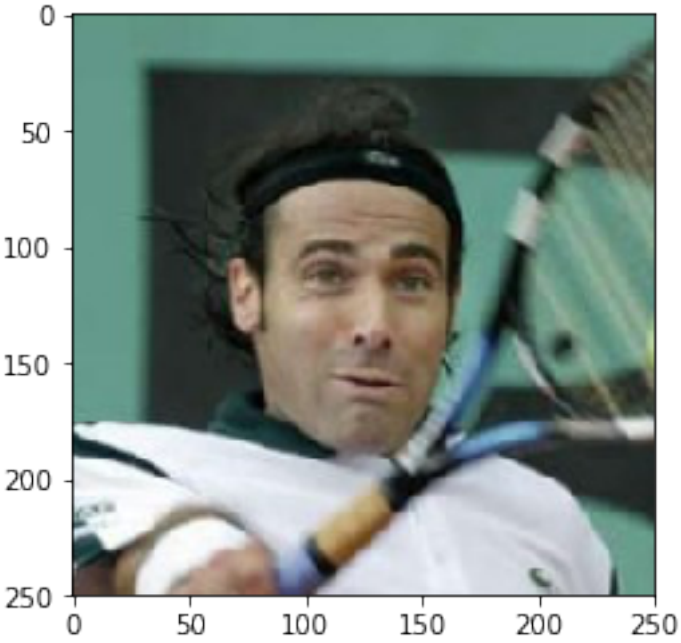
2) Choosing different hyperparameters: different optimizer, dropout, weight initialization.

3) Choosing a different model can all surely help in improving model. :)

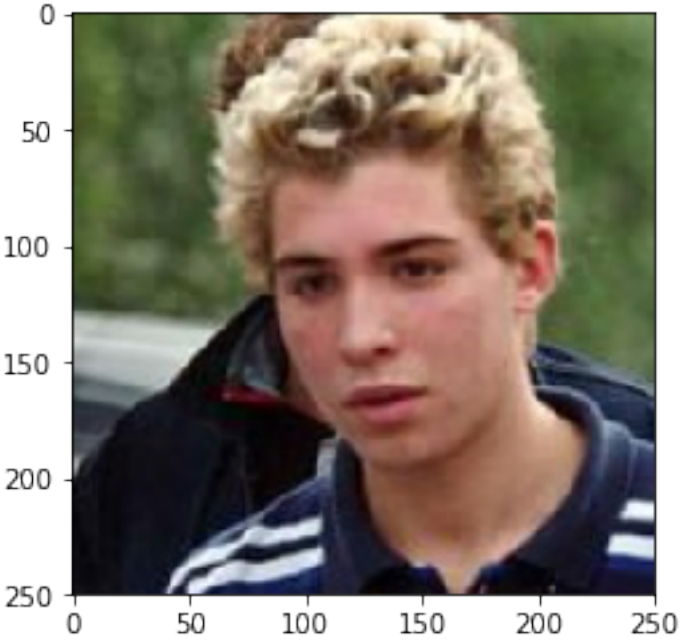
```
In [34]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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



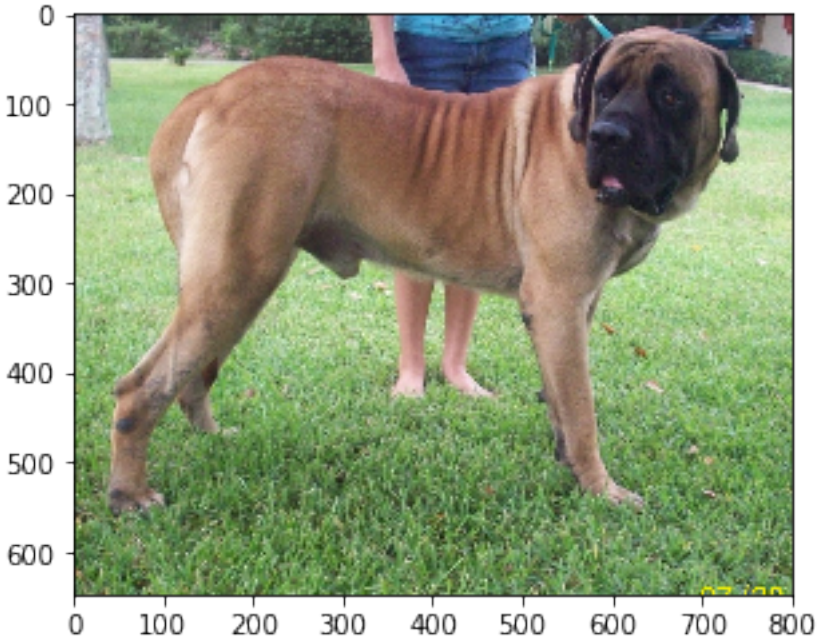

```
Hi, Human
You look like...
American staffordshire terrier
```



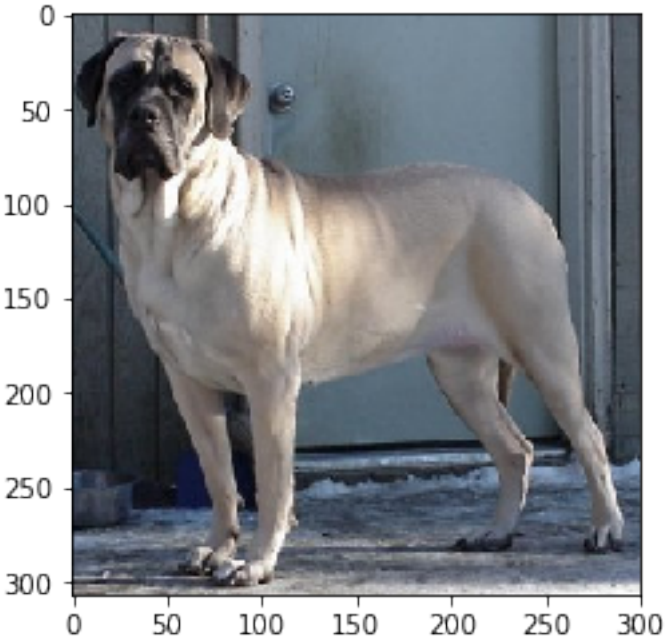

```
Hi, Human
You look like...
Australian shepherd
```



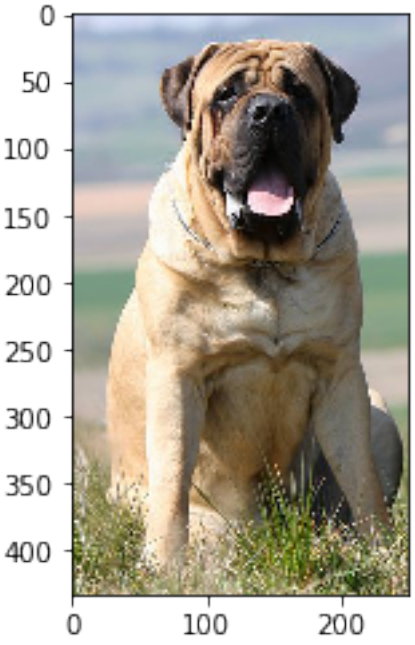
```
Hi, Human
You look like...
American water spaniel
```




```
It is a dog.
Its predicted breed is... Mastiff
```

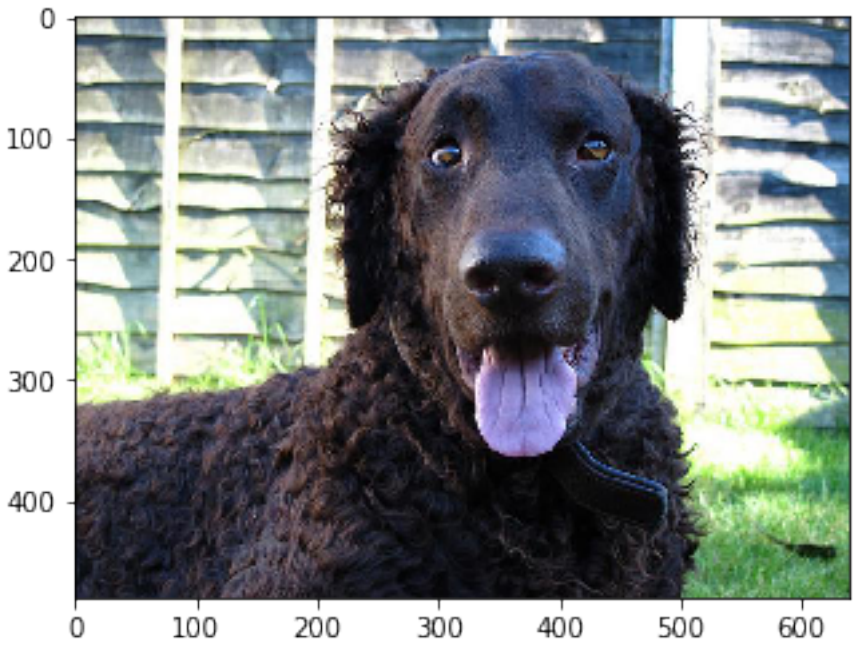



```
It is a dog.
Its predicted breed is... Bullmastiff
```



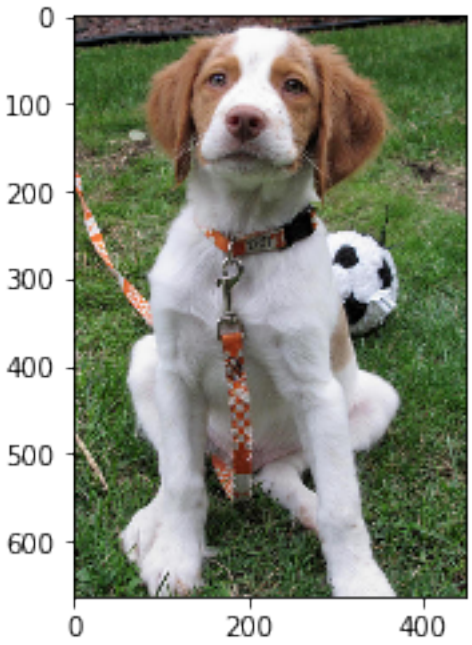
```
It is a dog.
Its predicted breed is... Bullmastiff
```

```
In [35]: run_app ('images/Curly-coated_retriever_03896.jpg')
```




```
It is a dog.
Its predicted breed is... Curly-coated retriever
```

```
In [37]: run_app('images/Brittany_02625.jpg')
```




```
It is a dog.
Its predicted breed is... Brittany
```

```
In [ ]:
```