dog_app

June 6, 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

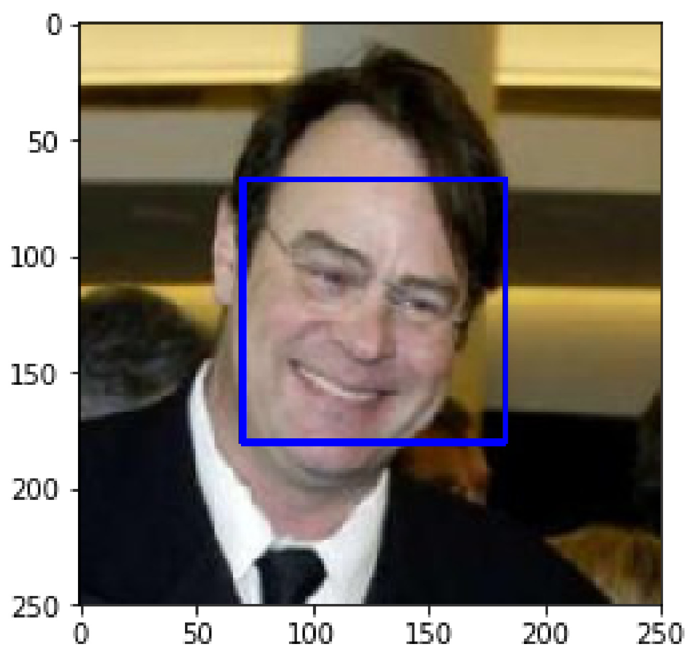
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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. #-#-#

count_detection = 0;
total_cnt = len(human_files_short)
for file in human_files_short:
    ans = face_detector(file)
    count_detection += ans
print("Human Face Detection in Human Files",count_detection,"/",total_cnt)
## TODO: Test the performance of the face_detector algorithm
## on the images in human_files_short and dog_files_short.
count_detection = 0;
total_cnt = len(dog_files_short)
for file in dog_files_short:
    ans = face_detector(file)
    count_detection += ans
print("Human Face Detected in Dog Files",count_detection,"/",total_cnt)
```

```
Human Face Detection in Human Files 98 / 100
Human Face Detected in Dog Files 17 / 100
```

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

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

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [6]: import torch
        import torchvision.models as models

        # define VGG16 model
        VGG16 = models.vgg16(pretrained=True)

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

        # move model to GPU if CUDA is available
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:04<00:00, 112184347.41it/s]
```

```
In [7]: if use_cuda:
            VGG16 = VGG16.cuda()
```

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 [17]: from PIL import Image
         import torchvision.transforms as transforms

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

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''
             image = Image.open(img_path).convert("RGB")
             transform = transforms.Compose([transforms.Resize(size=(244,244)),transforms.Random
             # normalizaiton parameters from pytorch doc.

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

             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image
             if use_cuda:
                 prediction = VGG16(image.cuda())
                 prediction = prediction.cpu().data.numpy().argmax()
             else :
                 prediction = VGG16(image)
                 return torch.max(prediction,1)[1].item()
             return prediction # predicted class index
```

```
In [18]: VGG16_predict(dog_files_short[0])
```

```
Out[18]: 243
```

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

```
        index = VGG16_predict(img_path)
        return index <= 268 and index >= 151# 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:

```
In [22]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         count = 0
         for file  in human_files_short:
             count+=dog_detector(file)
         print("Dog detected in human files",count,"/",len(human_files_short))
         count = 0
         for file  in dog_files_short:
             count+=dog_detector(file)
         print("Dog Detected in Dog Files",count,"/",len(dog_files_short))
```

```
Dog detected in human files 0 / 100
Dog Detected in Dog Files 97 / 100
```

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

```
In [60]: ### (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 [61]:
```
import os
from torchvision import datasets
import torchvision.transforms as transforms
import torch

import numpy as np

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


data_dir = '/data/dog_images/'
train_dir = os.path.join(data_dir, 'train/')
valid_dir = os.path.join(data_dir, 'valid/')
test_dir = os.path.join(data_dir, 'test/')
std_norm = transforms.Normalize(mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.25
```

```
data_transforms ={'train': transforms.Compose([transforms.RandomResizedCrop(224),transf
                  'val': transforms.Compose([transforms.Resize(226), transforms.CenterC
                  'test': transforms.Compose([transforms.Resize(size=(224,224)), transf
train_data = datasets.ImageFolder(train_dir, transform = data_transforms['train'])
valid_data = datasets.ImageFolder(valid_dir,transform = data_transforms['val'])
test_data = datasets.ImageFolder(test_dir,transform = data_transforms['test'])

batch_size = 20
num_workers = 0
trainloader = torch.utils.data.DataLoader(train_data,batch_size = batch_size, num_worke
valiloader = torch.utils.data.DataLoader(valid_data, batch_size = batch_size, num_worke
testloader = torch.utils.data.DataLoader(test_data, batch_size = batch_size, num_worker
Loaders = {'train':trainloader,'test':testloader,'valid':valiloader}
```

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:

To train data, I've applied RandomSizedCrop and Horizontal Flip , both resizing and augmentating train_data. By augmenting I expect better results on train_data. Also, preventing overfitting For valid_data, I have applied Resize(226) and CenterCrop(224) = 224 X 224. The validation data is for validating so no image Augmentation. No augmentation is used for test_data also, For test only Resizing is used.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [11]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         num_classes = 133
```

```
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__()
                 self.conv1 = nn.Conv2d(3, 32, 3, stride = 2, padding = 1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride = 2, padding = 1)
                 self.conv3 = nn.Conv2d(64, 128, 3,  padding = 1)
         #         self.conv4 = nn.Conv2d(128, 256, 3, stride = 1,padding = 1)

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

                 self.fc1 = nn.Linear(7*7*128, 500)
```

```
        self.fc2 = nn.Linear(500,133)
#         self.fc3 = nn.Linear(133,133)


        self.dropout = nn.Dropout(0.3)
#         self.dropout2 = nn.Dropout(0.2)
        ## Define layers of a CNN

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
#         x= self.pool(F.relu(self.conv4(x)))
#         print(x.shape)

        x = x.view(-1, 7*7*128)
#         print(x.shape)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
#         x = self.fc3(x)""


        return x

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

# instantiate the CNN
model_scratch = Net()

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

In [13]:
```
model_scratch
```

Out[13]:
```
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=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.3)
)
```

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

Answer:

My Network is as following: 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)) (conv4): Conv2d(128, 256, 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=12544, out_features=500, bias=True) (fc2): Linear(in_features=500, out_features=133, bias=True) (dropout1): Dropout(p=0.3))

Explanation: I am using 4 convolution layers and two fully connected. conv1 and conv2 have kernel size (3,3) and stride (2,2), downsizing of image by 2. In conv3 and conv4, stride is one signifing no change in dimensions of image. pool layer is reducing the image by 64. After all conv layers, we'll have 4 X 4 X 256. And Appling dropout to this will avoid overfitting. Fully connected layer fc1, will work on the flattened images and will provide with 500 inputs to the fc2. fc2, will provide the final predictions

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

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

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.09)
```

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(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
                optimizer.zero_grad()
                output = model(data)
                loss = criterion(output,target)
                loss.backward()
                optimizer.step()
#                 train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_los
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)
                if batch_idx % 100 == 0:
                    print('Epoch %d, Batch %d loss: %.6f' % (epoch, batch_idx + 1, train_lo
            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(Loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output,target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)
            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(epoc

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

In [16]:
```
model_scratch = train(20, Loaders, model_scratch, optimizer_scratch, criterion_scratch,
```

```
Epoch 1, Batch 1 loss: 4.904297
Epoch 1, Batch 101 loss: 4.883441
Epoch 1, Batch 201 loss: 4.872369
Epoch 1, Batch 301 loss: 4.857360
Epoch: 1 	Training Loss: 4.852027 	Validation Loss: 4.724347
Validation loss decreased (inf --> 4.724347).  Saving model ...
Epoch 2, Batch 1 loss: 4.740420
Epoch 2, Batch 101 loss: 4.746137
Epoch 2, Batch 201 loss: 4.713694
```

```
Epoch 2, Batch 301 loss: 4.697197
Epoch: 2        Training Loss: 4.691772        Validation Loss: 4.477993
Validation loss decreased (4.724347 --> 4.477993).  Saving model ...
Epoch 3, Batch 1 loss: 4.541754
Epoch 3, Batch 101 loss: 4.613983
Epoch 3, Batch 201 loss: 4.601125
Epoch 3, Batch 301 loss: 4.586032
Epoch: 3        Training Loss: 4.580477        Validation Loss: 4.502907
Epoch 4, Batch 1 loss: 4.987620
Epoch 4, Batch 101 loss: 4.531548
Epoch 4, Batch 201 loss: 4.518010
Epoch 4, Batch 301 loss: 4.511339
Epoch: 4        Training Loss: 4.510870        Validation Loss: 4.270361
Validation loss decreased (4.477993 --> 4.270361).  Saving model ...
Epoch 5, Batch 1 loss: 4.456739
Epoch 5, Batch 101 loss: 4.438235
Epoch 5, Batch 201 loss: 4.416199
Epoch 5, Batch 301 loss: 4.414828
Epoch: 5        Training Loss: 4.412768        Validation Loss: 4.173578
Validation loss decreased (4.270361 --> 4.173578).  Saving model ...
Epoch 6, Batch 1 loss: 4.643089
Epoch 6, Batch 101 loss: 4.353110
Epoch 6, Batch 201 loss: 4.349885
Epoch 6, Batch 301 loss: 4.344499
Epoch: 6        Training Loss: 4.339184        Validation Loss: 4.161581
Validation loss decreased (4.173578 --> 4.161581).  Saving model ...
Epoch 7, Batch 1 loss: 4.278313
Epoch 7, Batch 101 loss: 4.263852
Epoch 7, Batch 201 loss: 4.262385
Epoch 7, Batch 301 loss: 4.262248
Epoch: 7        Training Loss: 4.249466        Validation Loss: 4.051602
Validation loss decreased (4.161581 --> 4.051602).  Saving model ...
Epoch 8, Batch 1 loss: 4.006072
Epoch 8, Batch 101 loss: 4.211720
Epoch 8, Batch 201 loss: 4.195191
Epoch 8, Batch 301 loss: 4.184651
Epoch: 8        Training Loss: 4.191563        Validation Loss: 4.096645
Epoch 9, Batch 1 loss: 4.111743
Epoch 9, Batch 101 loss: 4.106140
Epoch 9, Batch 201 loss: 4.120181
Epoch 9, Batch 301 loss: 4.128221
Epoch: 9        Training Loss: 4.124011        Validation Loss: 3.977837
Validation loss decreased (4.051602 --> 3.977837).  Saving model ...
Epoch 10, Batch 1 loss: 4.411751
Epoch 10, Batch 101 loss: 4.067576
Epoch 10, Batch 201 loss: 4.085449
Epoch 10, Batch 301 loss: 4.056787
Epoch: 10        Training Loss: 4.060209        Validation Loss: 3.996731
```

```
Epoch 11, Batch 1 loss: 3.932485
Epoch 11, Batch 101 loss: 3.955082
Epoch 11, Batch 201 loss: 3.965570
Epoch 11, Batch 301 loss: 3.978037
Epoch: 11         Training Loss: 3.979943         Validation Loss: 3.923017
Validation loss decreased (3.977837 --> 3.923017).  Saving model ...
Epoch 12, Batch 1 loss: 3.532090
Epoch 12, Batch 101 loss: 3.903154
Epoch 12, Batch 201 loss: 3.919204
Epoch 12, Batch 301 loss: 3.920167
Epoch: 12         Training Loss: 3.925153         Validation Loss: 3.962110
Epoch 13, Batch 1 loss: 3.616037
Epoch 13, Batch 101 loss: 3.831761
Epoch 13, Batch 201 loss: 3.851689
Epoch 13, Batch 301 loss: 3.866619
Epoch: 13         Training Loss: 3.872827         Validation Loss: 3.899328
Validation loss decreased (3.923017 --> 3.899328).  Saving model ...
Epoch 14, Batch 1 loss: 3.865402
Epoch 14, Batch 101 loss: 3.819099
Epoch 14, Batch 201 loss: 3.825713
Epoch 14, Batch 301 loss: 3.827878
Epoch: 14         Training Loss: 3.827740         Validation Loss: 3.779043
Validation loss decreased (3.899328 --> 3.779043).  Saving model ...
Epoch 15, Batch 1 loss: 3.703564
Epoch 15, Batch 101 loss: 3.754349
Epoch 15, Batch 201 loss: 3.761976
Epoch 15, Batch 301 loss: 3.758374
Epoch: 15         Training Loss: 3.755891         Validation Loss: 3.705425
Validation loss decreased (3.779043 --> 3.705425).  Saving model ...
Epoch 16, Batch 1 loss: 2.673039
Epoch 16, Batch 101 loss: 3.718846
Epoch 16, Batch 201 loss: 3.733446
Epoch 16, Batch 301 loss: 3.720389
Epoch: 16         Training Loss: 3.722823         Validation Loss: 3.733690
Epoch 17, Batch 1 loss: 2.906736
Epoch 17, Batch 101 loss: 3.677064
Epoch 17, Batch 201 loss: 3.676181
Epoch 17, Batch 301 loss: 3.680780
Epoch: 17         Training Loss: 3.679182         Validation Loss: 3.682679
Validation loss decreased (3.705425 --> 3.682679).  Saving model ...
Epoch 18, Batch 1 loss: 3.672214
Epoch 18, Batch 101 loss: 3.565964
Epoch 18, Batch 201 loss: 3.598293
Epoch 18, Batch 301 loss: 3.612562
Epoch: 18         Training Loss: 3.619976         Validation Loss: 3.867052
Epoch 19, Batch 1 loss: 3.834167
Epoch 19, Batch 101 loss: 3.524819
Epoch 19, Batch 201 loss: 3.532051
```

```
Epoch 19, Batch 301 loss: 3.561999
Epoch: 19         Training Loss: 3.578168         Validation Loss: 3.692367
Epoch 20, Batch 1 loss: 3.837141
Epoch 20, Batch 101 loss: 3.614466
Epoch 20, Batch 201 loss: 3.590082
Epoch 20, Batch 301 loss: 3.552526
Epoch: 20         Training Loss: 3.550200         Validation Loss: 3.539784
Validation loss decreased (3.682679 --> 3.539784).  Saving model ...
```

```
In [17]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [18]: if use_cuda:
             print("Cuda aVAl")
```

```
Cuda aVAl
```

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 [28]: 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, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.644132


Test Accuracy: 14% (125/836)
```

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 [83]: ## TODO: Specify data loaders
         loaders_transfer = Loaders.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [84]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.resnet50(pretrained=True)
         for param in model_transfer.parameters():
             param.required_grad  = False

         model_transfer
```

```
Out[84]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
```

```
(layer1): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    (relu): ReLU(inplace)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
    )
  )
  (1): Bottleneck(
```

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

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

```
            (downsample): Sequential(
              (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
              (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
            )
          )
          (1): Bottleneck(
            (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
            (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
            (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
            (relu): ReLU(inplace)
          )
          (2): Bottleneck(
            (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
            (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
            (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
            (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stat
            (relu): ReLU(inplace)
          )
        )
        (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
        (fc): Linear(in_features=2048, out_features=1000, bias=True)
      )
```

```
In [85]: model_transfer.fc = nn.Linear(2048, 133, bias=True)
```

```
In [86]: for param in model_transfer.fc.parameters():
             param.required_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 [87]: 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:

In theabove CNN architecture, I have used transfer learning and applied ResNet architecture, beacause it gives good result in Image Classification. I have a created a new full connected layer with 113 outputs, allowing it to classify to 113 classes. In the model, I have put required_grad() true only for fully connected layers, the rest of the layers won't require grad, because the rest of the model is already trained.

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 [88]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr = 0.05)
```

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 [34]: # train the model
         model_transfer =  train(20, loaders_transfer, model_transfer, optimizer_transfer, crite
```

```
Epoch 1, Batch 1 loss: 3.016304
Epoch 1, Batch 101 loss: 2.744655
Epoch 1, Batch 201 loss: 2.430051
Epoch 1, Batch 301 loss: 2.201836
Epoch: 1        Training Loss: 2.139226        Validation Loss: 0.932287
Validation loss decreased (inf --> 0.932287).  Saving model ...
Epoch 2, Batch 1 loss: 1.614223
Epoch 2, Batch 101 loss: 1.353407
Epoch 2, Batch 201 loss: 1.352534
Epoch 2, Batch 301 loss: 1.318322
Epoch: 2        Training Loss: 1.297075        Validation Loss: 0.650386
Validation loss decreased (0.932287 --> 0.650386).  Saving model ...
Epoch 3, Batch 1 loss: 1.403655
Epoch 3, Batch 101 loss: 1.131113
Epoch 3, Batch 201 loss: 1.143867
Epoch 3, Batch 301 loss: 1.129119
Epoch: 3        Training Loss: 1.124858        Validation Loss: 0.549246
Validation loss decreased (0.650386 --> 0.549246).  Saving model ...
Epoch 4, Batch 1 loss: 0.738822
Epoch 4, Batch 101 loss: 0.976005
Epoch 4, Batch 201 loss: 0.972623
Epoch 4, Batch 301 loss: 0.979292
Epoch: 4        Training Loss: 0.976941        Validation Loss: 0.500454
Validation loss decreased (0.549246 --> 0.500454).  Saving model ...
Epoch 5, Batch 1 loss: 0.933326
Epoch 5, Batch 101 loss: 0.888348
Epoch 5, Batch 201 loss: 0.891249
Epoch 5, Batch 301 loss: 0.914435
Epoch: 5        Training Loss: 0.913550        Validation Loss: 0.473830
Validation loss decreased (0.500454 --> 0.473830).  Saving model ...
Epoch 6, Batch 1 loss: 0.778854
Epoch 6, Batch 101 loss: 0.910423
Epoch 6, Batch 201 loss: 0.886951
Epoch 6, Batch 301 loss: 0.880525
Epoch: 6        Training Loss: 0.880102        Validation Loss: 0.458532
Validation loss decreased (0.473830 --> 0.458532).  Saving model ...
Epoch 7, Batch 1 loss: 0.820914
Epoch 7, Batch 101 loss: 0.850395
Epoch 7, Batch 201 loss: 0.849584
Epoch 7, Batch 301 loss: 0.857668
Epoch: 7        Training Loss: 0.859922        Validation Loss: 0.444596
Validation loss decreased (0.458532 --> 0.444596).  Saving model ...
Epoch 8, Batch 1 loss: 0.608399
Epoch 8, Batch 101 loss: 0.807510
Epoch 8, Batch 201 loss: 0.800615
Epoch 8, Batch 301 loss: 0.810238
Epoch: 8        Training Loss: 0.803349        Validation Loss: 0.425124
Validation loss decreased (0.444596 --> 0.425124).  Saving model ...
```

```
Epoch 9, Batch 1 loss: 0.454510
Epoch 9, Batch 101 loss: 0.796147
Epoch 9, Batch 201 loss: 0.781630
Epoch 9, Batch 301 loss: 0.772754
Epoch: 9        Training Loss: 0.769858        Validation Loss: 0.390222
Validation loss decreased (0.425124 --> 0.390222).  Saving model ...
Epoch 10, Batch 1 loss: 0.575975
Epoch 10, Batch 101 loss: 0.747352
Epoch 10, Batch 201 loss: 0.715471
Epoch 10, Batch 301 loss: 0.749511
Epoch: 10        Training Loss: 0.753510        Validation Loss: 0.376710
Validation loss decreased (0.390222 --> 0.376710).  Saving model ...
Epoch 11, Batch 1 loss: 0.444356
Epoch 11, Batch 101 loss: 0.688461
Epoch 11, Batch 201 loss: 0.723141
Epoch 11, Batch 301 loss: 0.738630
Epoch: 11        Training Loss: 0.744491        Validation Loss: 0.390005
Epoch 12, Batch 1 loss: 0.445772
Epoch 12, Batch 101 loss: 0.745241
Epoch 12, Batch 201 loss: 0.744728
Epoch 12, Batch 301 loss: 0.747980
Epoch: 12        Training Loss: 0.743573        Validation Loss: 0.398802
Epoch 13, Batch 1 loss: 0.881874
Epoch 13, Batch 101 loss: 0.742158
Epoch 13, Batch 201 loss: 0.734896
Epoch 13, Batch 301 loss: 0.732033
Epoch: 13        Training Loss: 0.729834        Validation Loss: 0.415215
Epoch 14, Batch 1 loss: 0.563947
Epoch 14, Batch 101 loss: 0.697861
Epoch 14, Batch 201 loss: 0.717407
Epoch 14, Batch 301 loss: 0.717669
Epoch: 14        Training Loss: 0.722167        Validation Loss: 0.376382
Validation loss decreased (0.376710 --> 0.376382).  Saving model ...
Epoch 15, Batch 1 loss: 0.971855
Epoch 15, Batch 101 loss: 0.677000
Epoch 15, Batch 201 loss: 0.686369
Epoch 15, Batch 301 loss: 0.694500
Epoch: 15        Training Loss: 0.693446        Validation Loss: 0.387354
Epoch 16, Batch 1 loss: 0.794873
Epoch 16, Batch 101 loss: 0.695291
Epoch 16, Batch 201 loss: 0.694459
Epoch 16, Batch 301 loss: 0.693890
Epoch: 16        Training Loss: 0.704071        Validation Loss: 0.377547
Epoch 17, Batch 1 loss: 0.666065
Epoch 17, Batch 101 loss: 0.657807
Epoch 17, Batch 201 loss: 0.674208
Epoch 17, Batch 301 loss: 0.677045
Epoch: 17        Training Loss: 0.675199        Validation Loss: 0.382637
```

```
Epoch 18, Batch 1 loss: 0.951814
Epoch 18, Batch 101 loss: 0.679490
Epoch 18, Batch 201 loss: 0.656461
Epoch 18, Batch 301 loss: 0.658395
Epoch: 18         Training Loss: 0.666728         Validation Loss: 0.391608
Epoch 19, Batch 1 loss: 0.550489
Epoch 19, Batch 101 loss: 0.674701
Epoch 19, Batch 201 loss: 0.659774
Epoch 19, Batch 301 loss: 0.666206
Epoch: 19         Training Loss: 0.667394         Validation Loss: 0.400225
Epoch 20, Batch 1 loss: 0.607578
Epoch 20, Batch 101 loss: 0.676583
Epoch 20, Batch 201 loss: 0.689026
Epoch 20, Batch 301 loss: 0.685571
Epoch: 20         Training Loss: 0.689461         Validation Loss: 0.381436
```

```
In [89]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.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 [90]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 0.484475


Test Accuracy: 84% (706/836)
```

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

         class_names = [item[4:].replace("_", " ") for item in loaders_transfer['train'].dataset

In [92]: loaders_transfer['train'].dataset.classes[:10]

Out[92]: ['001.Affenpinscher',
          '002.Afghan_hound',
          '003.Airedale_terrier',
          '004.Akita',
```

```
 '005.Alaskan_malamute',
 '006.American_eskimo_dog',
 '007.American_foxhound',
 '008.American_staffordshire_terrier',
 '009.American_water_spaniel',
 '010.Anatolian_shepherd_dog']
```

In [93]:
```
class_names[:10]
```

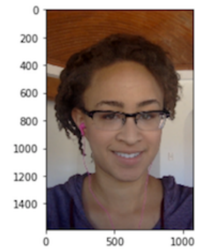
Out[93]:
```
['Affenpinscher',
 'Afghan hound',
 'Airedale terrier',
 'Akita',
 'Alaskan malamute',
 'American eskimo dog',
 'American foxhound',
 'American staffordshire terrier',
 'American water spaniel',
 'Anatolian shepherd dog']
```

In [94]:
```
from PIL import Image
import torchvision.transforms as transforms

def load_input_image(img_path):
    image = Image.open(img_path).convert('RGB')
    prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                     transforms.ToTensor(),
                                     std_norm])

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

In [95]:
```
def predict_breed_transfer(model, class_names, img_path):
    # load the image and return the predicted breed
    img = load_input_image(img_path)
    model = model.cpu()
    model.eval()
    idx = torch.argmax(model(img))
    return class_names[idx]
```

In [96]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    predition = predict_breed_transfer(model_transfer, class_names, img_path)
    print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```

```
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,          predition breed: Irish red
image_file_name: ./images/sample_human_output.png,          predition breed: English toy spaniel
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition breed: Labrador retri
```



Sample Human Output

```
image_file_name: ./images/Curly-coated_retriever_03896.jpg,          predition breed: Curly-coat
image_file_name: ./images/sample_cnn.png,          predition breed: Affenpinscher
image_file_name: ./images/Brittany_02625.jpg,          predition breed: Brittany
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition breed: Labrador retri
image_file_name: ./images/American_water_spaniel_00648.jpg,          predition breed: Curly-coat
image_file_name: ./images/sample_dog_output.png,          predition breed: Entlebucher mountain
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition breed: Labrador retri
```

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!

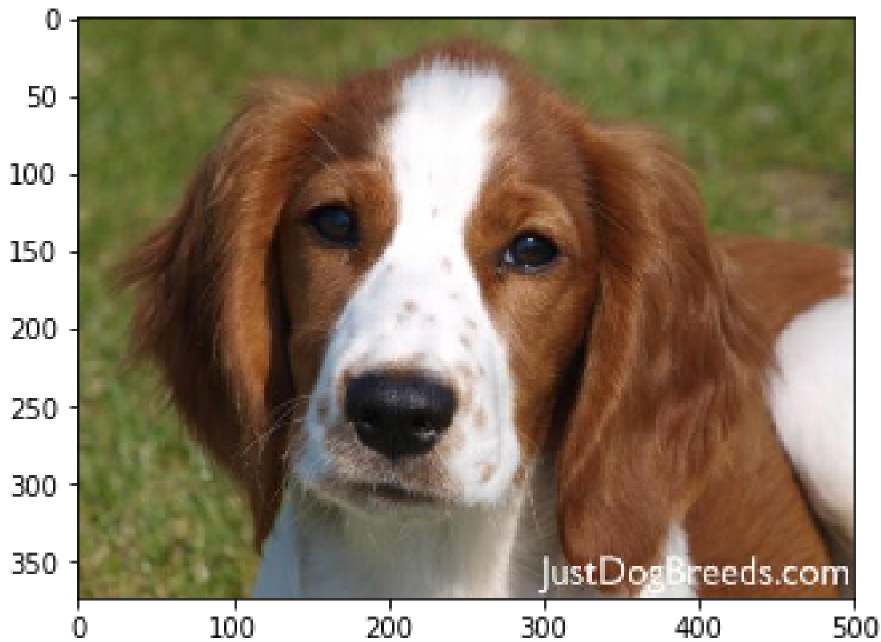
1.1.18 (IMPLEMENTATION) Write your Algorithm

In [103]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    index = VGG16_predict(img_path)
    if index <= 268 and index >= 151:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Dogs Detected!\nIt looks like a {0}".format(prediction))
```

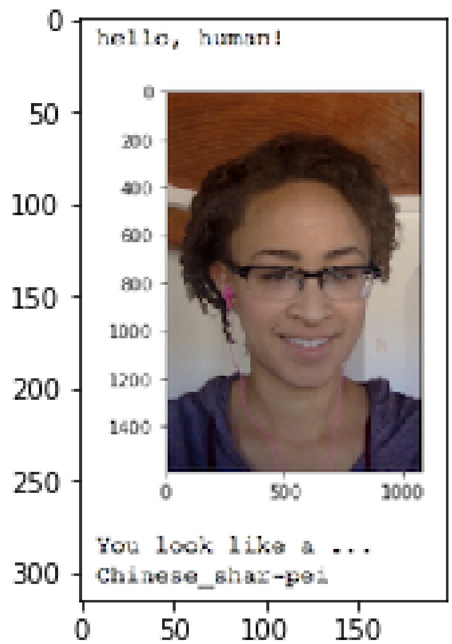
```
        elif face_detector(img_path) > 0:
            prediction = predict_breed_transfer(model_transfer, class_names, img_path)
            print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predi
        else:
            print("Error! Can't detect anything..")
```

```
In [104]: import os
```

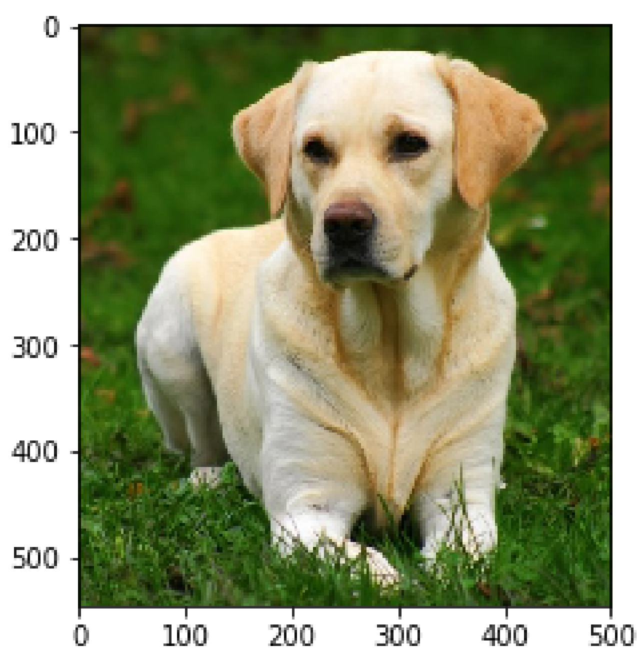
```
In [105]: for img_file in os.listdir('./images'):
              img_path = os.path.join('./images', img_file)
              run_app(img_path)
```



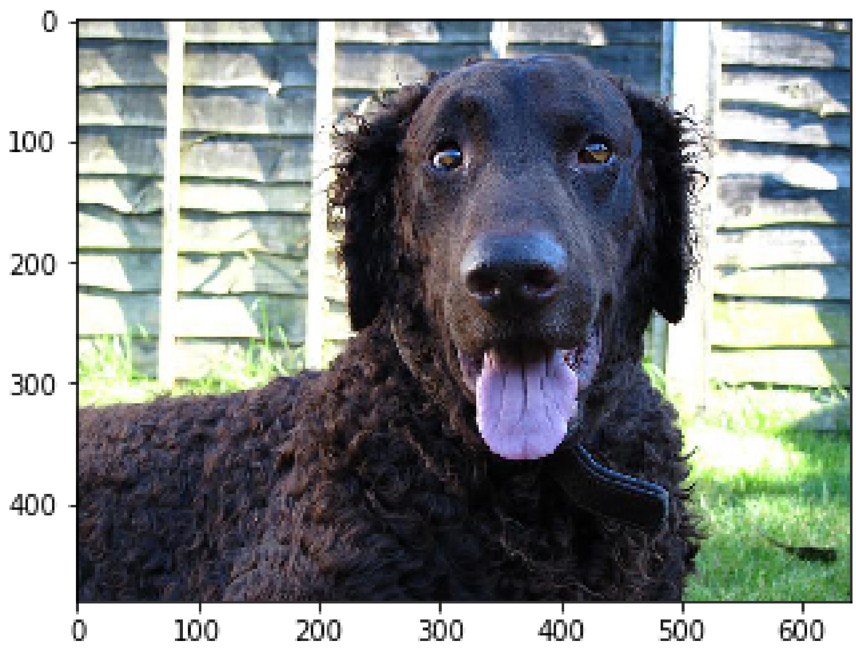

```
Dogs Detected!
It looks like a Irish red and white setter
```



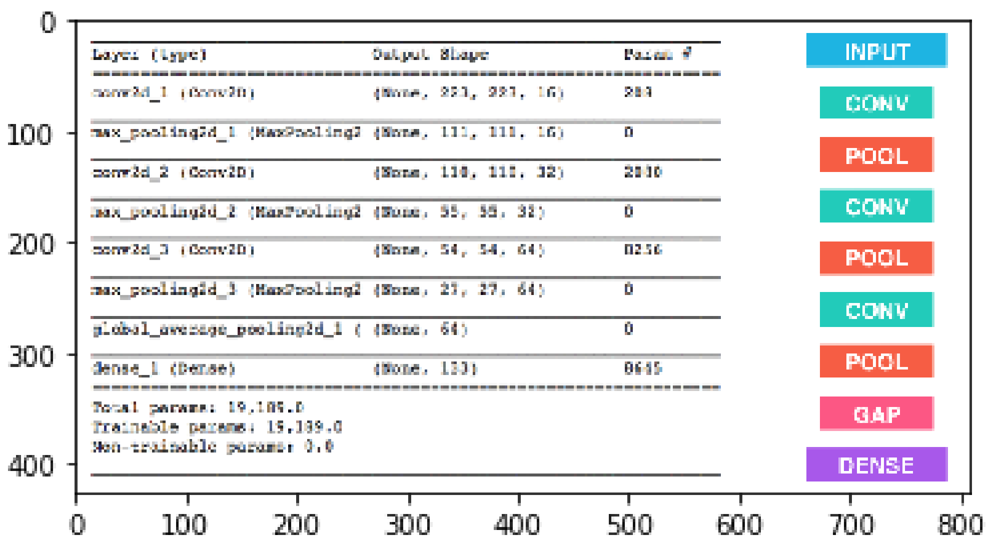

```
Hello, human!
If you were a dog..You may look like a English toy spaniel
```



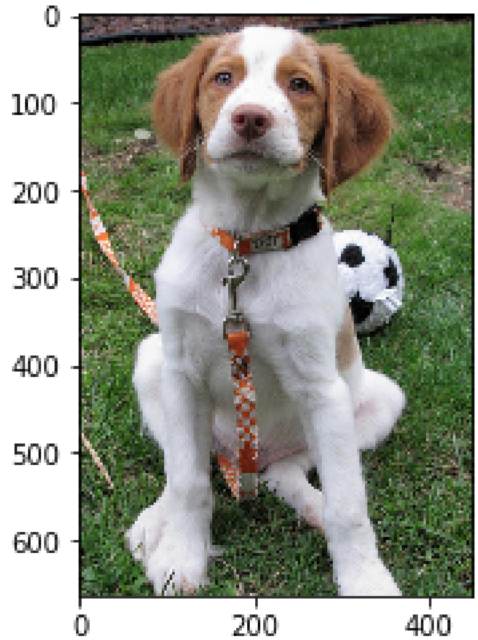
```
Dogs Detected!
It looks like a Labrador retriever
```




```
Dogs Detected!
It looks like a Curly-coated retriever
```



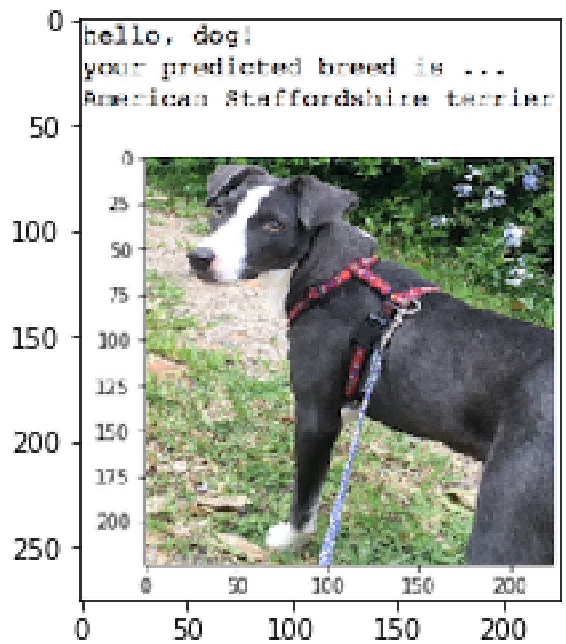
```
Error! Can't detect anything..
```



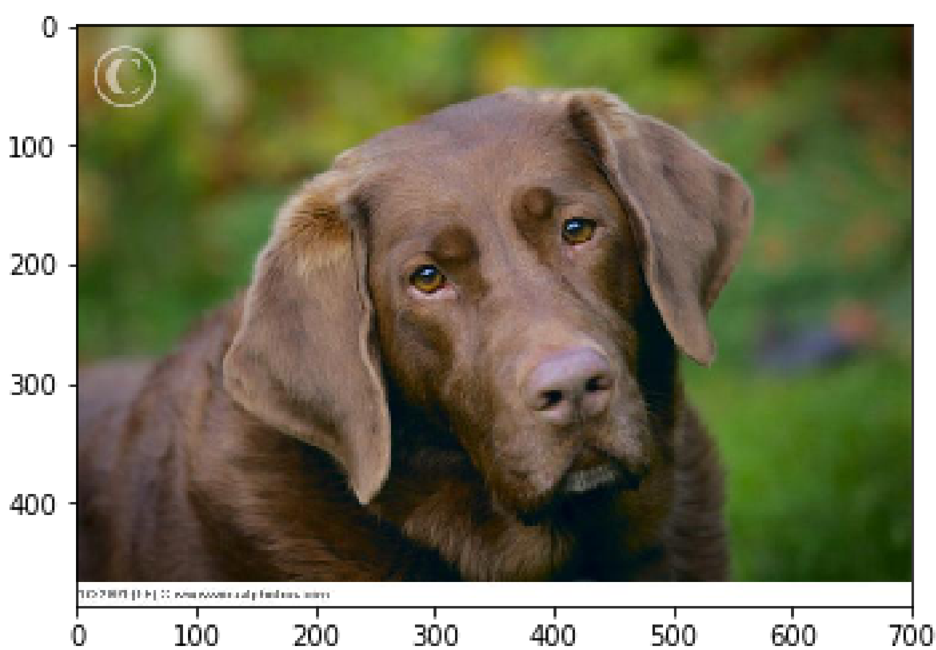

```
Dogs Detected!
It looks like a Brittany
```

```
Dogs Detected!
It looks like a Labrador retriever
```


```
Dogs Detected!
It looks like a Curly-coated retriever
```




```
Dogs Detected!
It looks like a Entlebucher mountain dog
```



```
Dogs Detected!
It looks like a Labrador retriever
```

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.

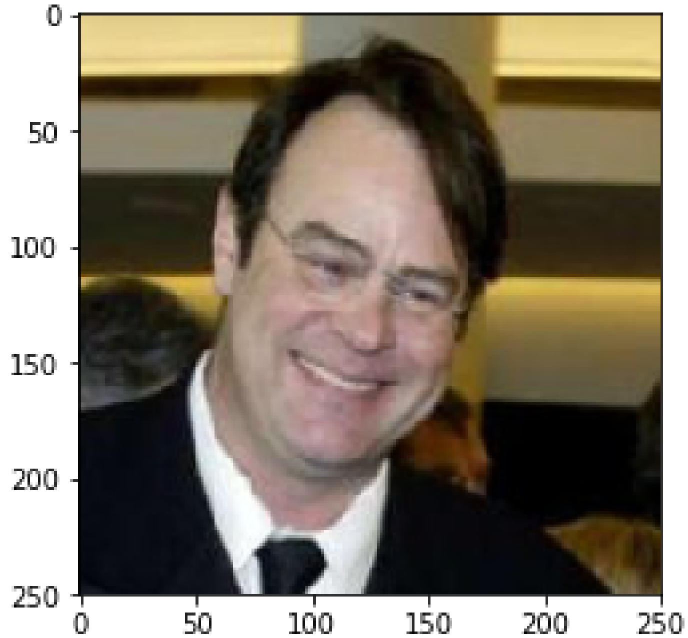
Answer: (Three possible points for improvement)

1. More dataset would diversify the data, and models will have more data to train on. Augmentation can be also can be worked upon like Vertical Flip etc.

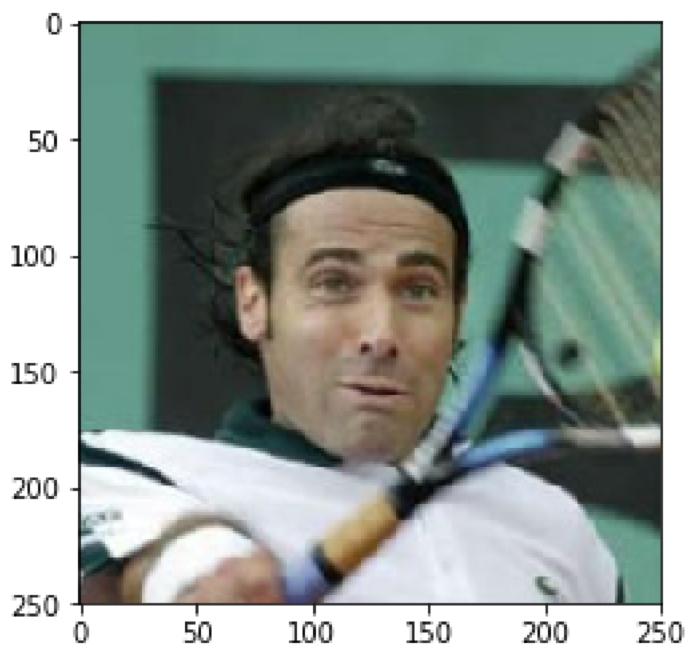
2. Learning rates and drop-out can also be improved. Hyperparameters

3. We can experiment with more models, ensembling the models, might be improve model's performance.

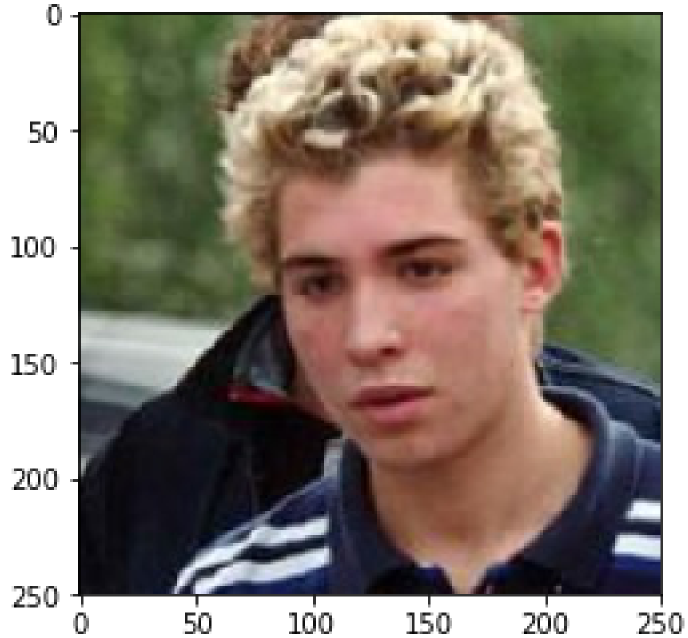
```
In [106]: ## TODO: Execute your algorithm from Step 6 on
          ## at least 6 images on your computer.
          ## Feel free to use as many code cells as needed.

          ## suggested code, below
          for file in np.hstack((human_files[:6], dog_files[:6])):
              run_app(file)
```



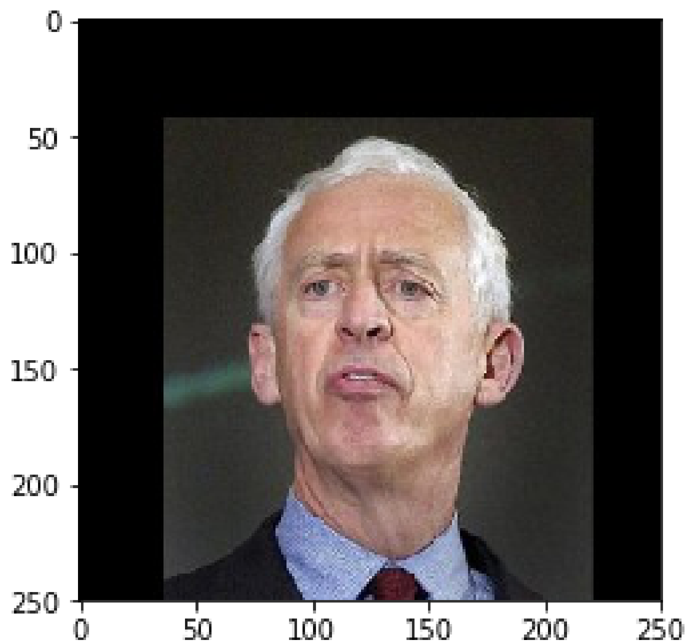

```
Hello, human!
If you were a dog..You may look like a Chihuahua
```



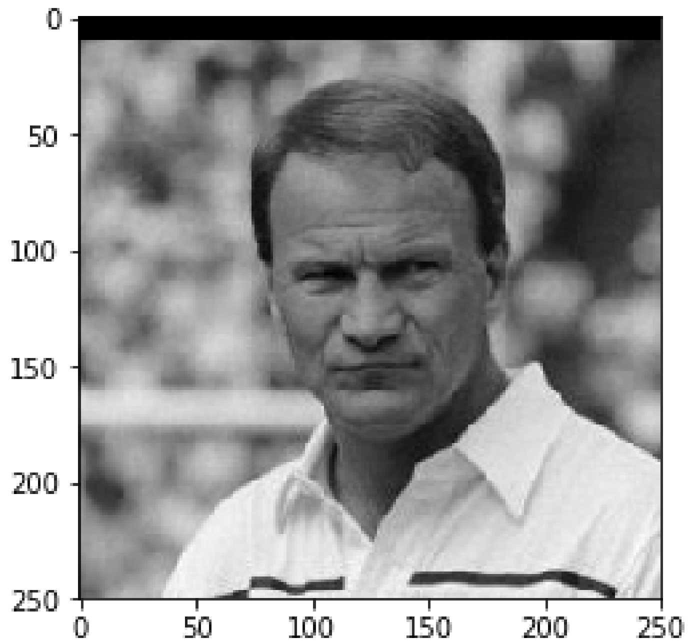
```
Hello, human!
If you were a dog..You may look like a American foxhound
```



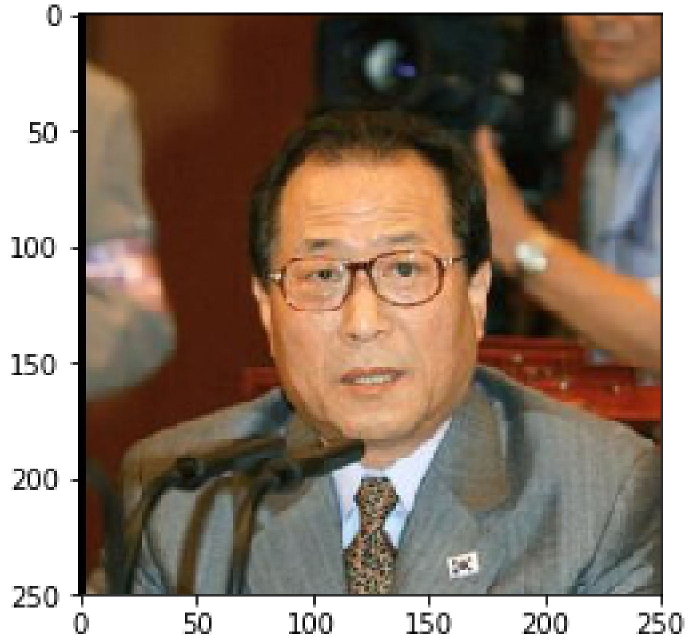

```
Hello, human!
If you were a dog..You may look like a American water spaniel
```



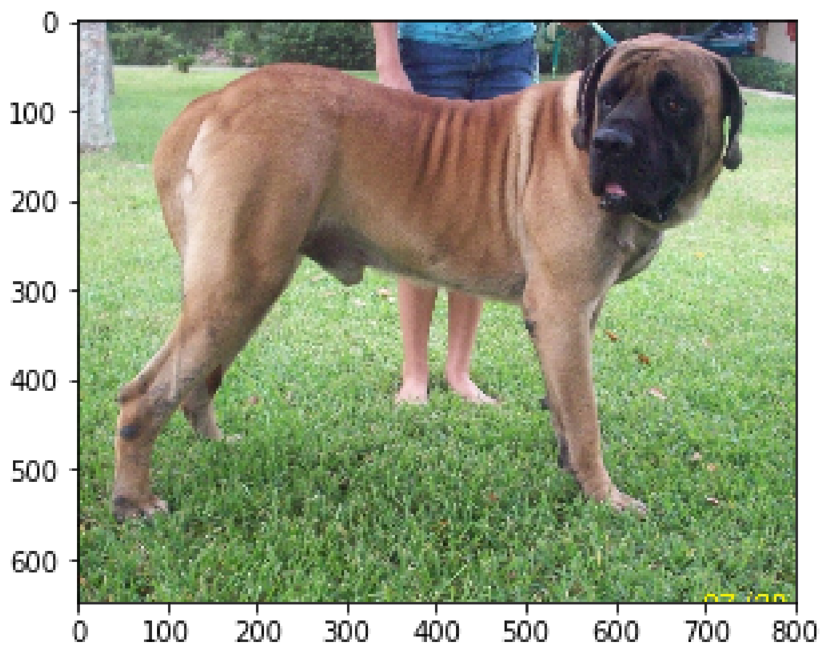
```
Hello, human!
If you were a dog..You may look like a Bull terrier
```



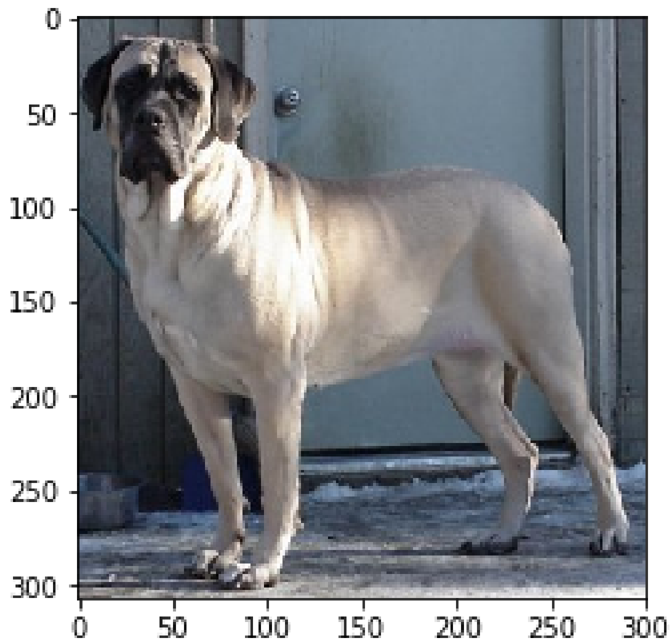

```
Hello, human!
If you were a dog..You may look like a Smooth fox terrier
```



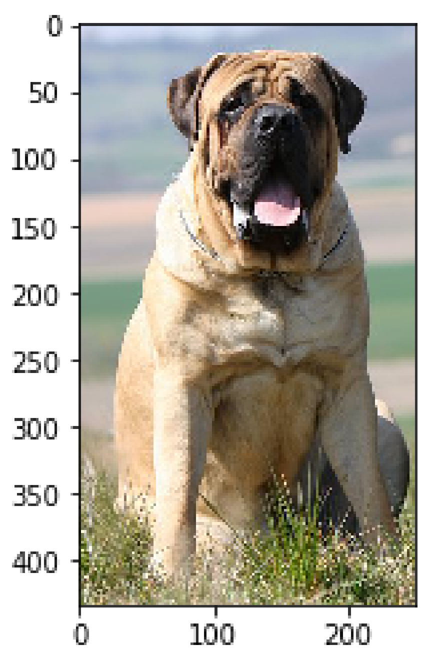

```
Hello, human!
If you were a dog..You may look like a Irish wolfhound
```



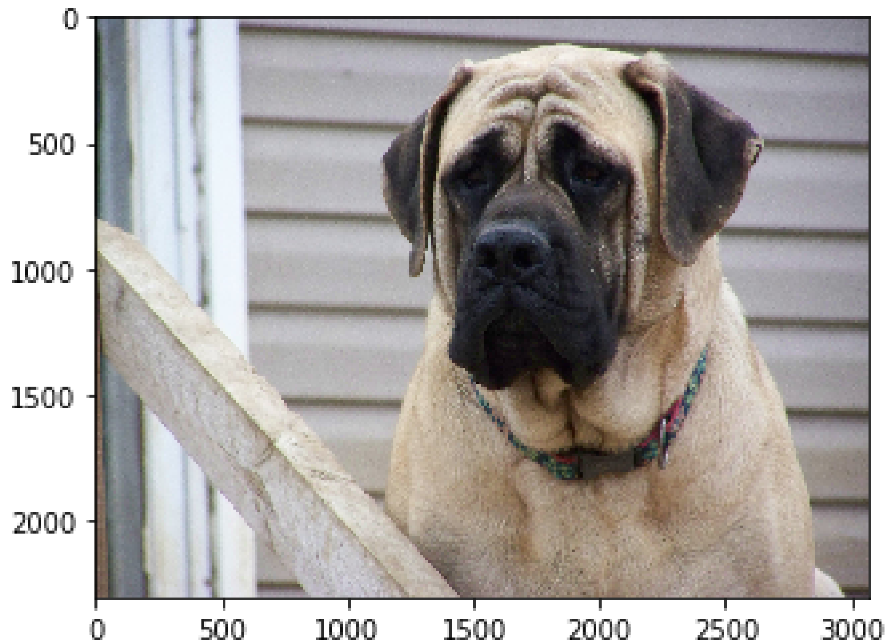
```
Dogs Detected!
It looks like a Bullmastiff
```



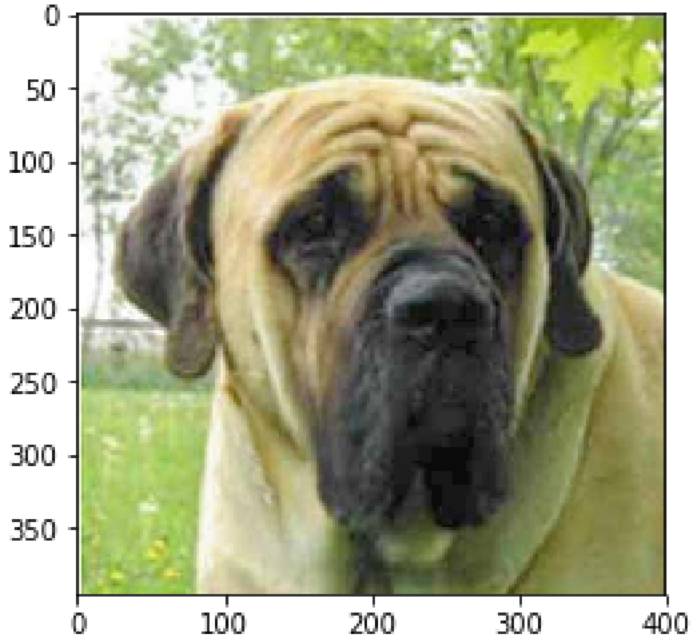

```
Dogs Detected!
It looks like a Bullmastiff
```



```
Dogs Detected!
It looks like a Bullmastiff
```




```
Dogs Detected!
It looks like a Mastiff
```




```
Dogs Detected!
It looks like a Mastiff
```

```
Dogs Detected!
It looks like a Mastiff
```

```
In [ ]:
```