dog_app

February 20, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- Download the human dataset. Unzip the folder and place it in the home diretcory, 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 [37]: import numpy as np
         from glob import glob

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

         # 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

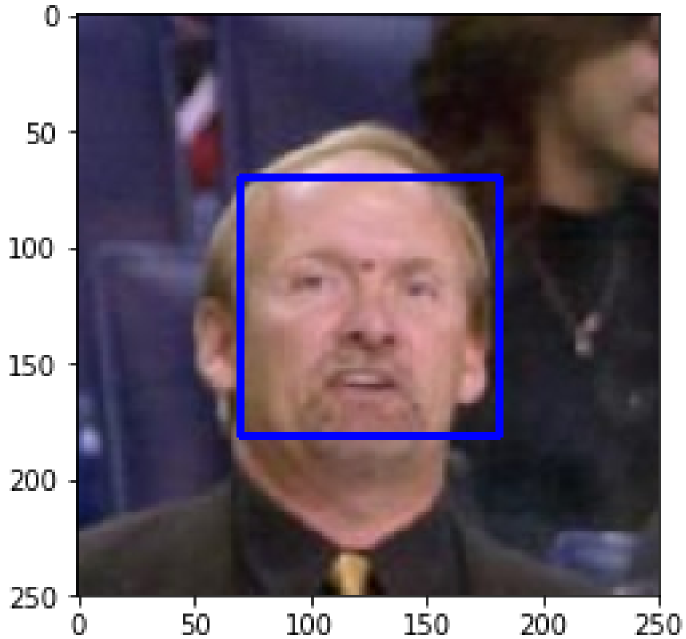
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [38]: 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 [34]: # 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 [68]: 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.

         def inference(images):
             correct_count = 0;
             total_count = len(images)
             for images in images:
                 correct_count += face_detector(images)

             return correct_count, total_count
```

```
In [5]: print("Correctly detected face in humans : {} out of  {}".format(inference(human_files_s
        print("Wrongly detected face in dogs: {} out of {}".format(inference(dog_files_short)[0]
```

```
Correctly detected face in humans : 97 out of  100
Wrongly detected face in dogs: 14 out of 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 [ ]: ### (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 [9]: 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()
```

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

         def preprocess_image(img_path):
             image = Image.open(img_path).convert('RGB')
             transform = transforms.Compose([
                                 transforms.Resize(size=(244, 244)),
                                 transforms.ToTensor()]) # normalizaiton parameters from pytorch

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

In [66]:
```
from PIL import Image
import torchvision.transforms as transforms

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

    Args:
        img_path: path to an image

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

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

In [69]:
```
print(dog_files_short[0])
VGG16_predict(dog_files_short[0])
# Class index for 243 in imagenet is bull mastiff
```

```
dogImages/train/041.Bullmastiff/Bullmastiff_02930.jpg
```

Out[69]: 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 [70]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(image_path):

    index = VGG16_predict(image_path)
```

```
        return index >= 151 and index <= 268
```

```
In [71]: print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))
```

```
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 [72]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_inference(images):
             correct_count = 0;
             total_count = len(images)
             for images in images:
                 correct_count += dog_detector(images)

             return correct_count, total_count
```

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 [73]: print("Percentage of wrong prediction of dog in human_files: {} out of {}".format(dog_i
         print("Percentage of correct prediction of dog in dog_files: {} out of {}".format(dog_i
```

```
Percentage of wrong prediction of dog in human_files: 0 out of 100
Percentage of correct prediction of dog in dog_files: 96 out of 100
```

```
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 `dogImages/train`, `dogImages/valid`, and `dogImages/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 [1]: 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

batch_size = 20
num_workers = 0

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

In [2]:
```
standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                              std=[0.229, 0.224, 0.225])
```

In [4]:
```
data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    standard_normalization]),
                   'val': transforms.Compose([transforms.Resize(256),
                                    transforms.CenterCrop(224),
                                    transforms.ToTensor(),
                                    standard_normalization]),
                   'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                    transforms.ToTensor(),
                                    standard_normalization])
                  }
```

In [5]:
```
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'])
```

In [6]:
```
train_loader = torch.utils.data.DataLoader(train_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)
test_loader = torch.utils.data.DataLoader(test_data,
                                           batch_size=batch_size,
                                           num_workers=num_workers,
                                           shuffle=False)


loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

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: Image augmentation is a importance step as it helps the model to generalize well by learning spatial invariance.So since there are various kind of dogs with similiar kind of looks its necessary to choose the right augmentation to this data.For the training set I found random crop,flips and rotations is more than enough rather than other augmentation like adding noise or ColorJitter as color is a importance feature.I did apply normalization and image resizing(224x224) to the entire set.

1.1.8 (IMPLEMENTATION) Model Architecture

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

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

        num_classes = 133
```

```
In [10]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np

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

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

                 # fully-connected
                 self.fc1 = nn.Linear(7*7*128, 500)
                 self.fc2 = nn.Linear(500, num_classes)

                 # drop-out
                 self.dropout = nn.Dropout(0.3)

             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)
            (_, C, H, W) = x.data.size()

            # flatten
            x = x.view( -1 , C * H * W)

            x = self.dropout(x)
            x = F.relu(self.fc1(x))

            x = self.dropout(x)
            x = self.fc2(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()
    print(model_scratch)
    # move tensors to GPU if CUDA is available
    if use_cuda:
        model_scratch.cuda()
```

```
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: This dataset contains dogs with slight varition which are difficult to find so I thought we need to go deeper to find more non-linear features and use dropout to avoid overfitting as going deeper we might overfit.Padding of 1 is used here as image size is 224 which is not divisble by kernel size of 3 so padding by 1 will make the image to 225x225 which makes operations easy.

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 [11]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.002)
```

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 [12]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path, last_val
             """returns trained model"""
             # initialize tracker for minimum validation loss
             if last_validation_loss is not None:
                 valid_loss_min = last_validation_loss
             else:
                 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

                     # initialize weights to zero
                     optimizer.zero_grad()

                     output = model(data)

                     # calculate loss
                     loss = criterion(output, target)

                     # back prop
                     loss.backward()
```

```
            # grad
            optimizer.step()

            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_loss))

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            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(
            epoch,
            train_loss,
            valid_loss
            ))

        ## 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))
            valid_loss_min = valid_loss

    # return trained model
    return model
```

In [10]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,  criterion
```

```
Epoch 1, Batch 1 loss: 4.900354
Epoch 1, Batch 101 loss: 4.891628
Epoch 1, Batch 201 loss: 4.860274
```

```
Epoch 1, Batch 301 loss: 4.818723
Epoch: 1        Training Loss: 4.805444         Validation Loss: 4.550626
Validation loss decreased (inf --> 4.550626).  Saving model ...
Epoch 2, Batch 1 loss: 4.491897
Epoch 2, Batch 101 loss: 4.617102
Epoch 2, Batch 201 loss: 4.596843
Epoch 2, Batch 301 loss: 4.583407
Epoch: 2        Training Loss: 4.577918         Validation Loss: 4.415434
Validation loss decreased (4.550626 --> 4.415434).  Saving model ...
Epoch 3, Batch 1 loss: 4.544852
Epoch 3, Batch 101 loss: 4.491723
Epoch 3, Batch 201 loss: 4.483330
Epoch 3, Batch 301 loss: 4.476408
Epoch: 3        Training Loss: 4.474557         Validation Loss: 4.302693
Validation loss decreased (4.415434 --> 4.302693).  Saving model ...
Epoch 4, Batch 1 loss: 4.468867
Epoch 4, Batch 101 loss: 4.426182
Epoch 4, Batch 201 loss: 4.409784
Epoch 4, Batch 301 loss: 4.378238
Epoch: 4        Training Loss: 4.372654         Validation Loss: 4.146058
Validation loss decreased (4.302693 --> 4.146058).  Saving model ...
Epoch 5, Batch 1 loss: 4.468889
Epoch 5, Batch 101 loss: 4.285531
Epoch 5, Batch 201 loss: 4.287722
Epoch 5, Batch 301 loss: 4.286022
Epoch: 5        Training Loss: 4.285462         Validation Loss: 4.095674
Validation loss decreased (4.146058 --> 4.095674).  Saving model ...
Epoch 6, Batch 1 loss: 4.408089
Epoch 6, Batch 101 loss: 4.202984
Epoch 6, Batch 201 loss: 4.203430
Epoch 6, Batch 301 loss: 4.202946
Epoch: 6        Training Loss: 4.198293         Validation Loss: 4.019811
Validation loss decreased (4.095674 --> 4.019811).  Saving model ...
Epoch 7, Batch 1 loss: 3.976004
Epoch 7, Batch 101 loss: 4.145569
Epoch 7, Batch 201 loss: 4.127645
Epoch 7, Batch 301 loss: 4.137451
Epoch: 7        Training Loss: 4.136090         Validation Loss: 3.959490
Validation loss decreased (4.019811 --> 3.959490).  Saving model ...
Epoch 8, Batch 1 loss: 3.906742
Epoch 8, Batch 101 loss: 4.046188
Epoch 8, Batch 201 loss: 4.054345
Epoch 8, Batch 301 loss: 4.073963
Epoch: 8        Training Loss: 4.081417         Validation Loss: 3.873703
Validation loss decreased (3.959490 --> 3.873703).  Saving model ...
Epoch 9, Batch 1 loss: 3.715015
Epoch 9, Batch 101 loss: 4.010583
Epoch 9, Batch 201 loss: 4.018592
```

```
Epoch 9, Batch 301 loss: 4.029079
Epoch: 9        Training Loss: 4.028146         Validation Loss: 3.867620
Validation loss decreased (3.873703 --> 3.867620).  Saving model ...
Epoch 10, Batch 1 loss: 3.541428
Epoch 10, Batch 101 loss: 3.962226
Epoch 10, Batch 201 loss: 3.969743
Epoch 10, Batch 301 loss: 3.971709
Epoch: 10        Training Loss: 3.967946         Validation Loss: 3.748860
Validation loss decreased (3.867620 --> 3.748860).  Saving model ...
Epoch 11, Batch 1 loss: 3.919062
Epoch 11, Batch 101 loss: 3.929842
Epoch 11, Batch 201 loss: 3.931310
Epoch 11, Batch 301 loss: 3.951674
Epoch: 11        Training Loss: 3.951593         Validation Loss: 3.740092
Validation loss decreased (3.748860 --> 3.740092).  Saving model ...
Epoch 12, Batch 1 loss: 3.698643
Epoch 12, Batch 101 loss: 3.873048
Epoch 12, Batch 201 loss: 3.899515
Epoch 12, Batch 301 loss: 3.889152
Epoch: 12        Training Loss: 3.891398         Validation Loss: 3.748124
Epoch 13, Batch 1 loss: 3.804721
Epoch 13, Batch 101 loss: 3.831264
Epoch 13, Batch 201 loss: 3.851583
Epoch 13, Batch 301 loss: 3.868336
Epoch: 13        Training Loss: 3.872385         Validation Loss: 3.686241
Validation loss decreased (3.740092 --> 3.686241).  Saving model ...
Epoch 14, Batch 1 loss: 3.103914
Epoch 14, Batch 101 loss: 3.780283
Epoch 14, Batch 201 loss: 3.794536
Epoch 14, Batch 301 loss: 3.804257
Epoch: 14        Training Loss: 3.812171         Validation Loss: 3.664676
Validation loss decreased (3.686241 --> 3.664676).  Saving model ...
Epoch 15, Batch 1 loss: 3.443187
Epoch 15, Batch 101 loss: 3.762853
Epoch 15, Batch 201 loss: 3.795690
Epoch 15, Batch 301 loss: 3.799425
Epoch: 15        Training Loss: 3.795832         Validation Loss: 3.617503
Validation loss decreased (3.664676 --> 3.617503).  Saving model ...
Epoch 16, Batch 1 loss: 3.950806
Epoch 16, Batch 101 loss: 3.750493
Epoch 16, Batch 201 loss: 3.757022
Epoch 16, Batch 301 loss: 3.765776
Epoch: 16        Training Loss: 3.773532         Validation Loss: 3.603326
Validation loss decreased (3.617503 --> 3.603326).  Saving model ...
Epoch 17, Batch 1 loss: 3.839179
Epoch 17, Batch 101 loss: 3.680423
Epoch 17, Batch 201 loss: 3.695931
Epoch 17, Batch 301 loss: 3.710709
```

```
Epoch: 17         Training Loss: 3.724113         Validation Loss: 3.614374
Epoch 18, Batch 1 loss: 3.818890
Epoch 18, Batch 101 loss: 3.685809
Epoch 18, Batch 201 loss: 3.713356
Epoch 18, Batch 301 loss: 3.710198
Epoch: 18         Training Loss: 3.718298         Validation Loss: 3.583520
Validation loss decreased (3.603326 --> 3.583520).  Saving model ...
Epoch 19, Batch 1 loss: 3.843981
Epoch 19, Batch 101 loss: 3.659343
Epoch 19, Batch 201 loss: 3.677769
Epoch 19, Batch 301 loss: 3.683173
Epoch: 19         Training Loss: 3.680303         Validation Loss: 3.574263
Validation loss decreased (3.583520 --> 3.574263).  Saving model ...
Epoch 20, Batch 1 loss: 3.395614
Epoch 20, Batch 101 loss: 3.658358
Epoch 20, Batch 201 loss: 3.676794
Epoch 20, Batch 301 loss: 3.686553
Epoch: 20         Training Loss: 3.688696         Validation Loss: 3.550816
Validation loss decreased (3.574263 --> 3.550816).  Saving model ...
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [13]: 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
        model_scratch.load_state_dict(torch.load('saved_models/model_scratch.pt'))
        test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.710245


Test Accuracy: 15% (128/836)
```

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 [7]: ## TODO: Specify data loaders
        loaders_transfer = loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [8]: 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.requires_grad = False
```

```
model_transfer.fc = nn.Linear(2048, 133, bias=True)
fc_parameters = model_transfer.fc.parameters()

for param in fc_parameters:
    param.requires_grad = True
use_cuda = torch.cuda.is_available()

if use_cuda:
    model_transfer = model_transfer.cuda()

model_transfer
```

Out[8]:
```
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=T
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (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=T
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (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=T
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=T
      (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_stats
      (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
      )
    )
    (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_stats
      (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_stats
      (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_stats
      (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_stats
      (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_stats
        (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_stats
        (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
        )
      )
      (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_stats
        (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_stats
        (relu): ReLU(inplace)
      )
    )
    (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
    (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: I tried different architecture such as alexnet, resnet18 and vgg-19 neither of them gave me better accuracy then resnet50. I do not want to use inception or resnet higher versions as I felt these pretrained layers where already trained on dog breeds.

Also resnet have skip connections in them which helps to prevent overfitting when training and I also replaced the output neurons to 133 classes of dog

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 [9]: import torch.optim as optim
        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.002)
```

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 [10]: 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()

                     # initialize weights to zero
                     optimizer.zero_grad()

                     output = model(data)

                     # calculate loss
                     loss = criterion(output, target)

                     # back prop
                     loss.backward()
```

```
                # grad
                optimizer.step()

                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_loss))

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                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(
                epoch,
                train_loss,
                valid_loss
                ))

            ## 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))
                valid_loss_min = valid_loss

        # return trained model
        return model
```

In [11]:
```
# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer, criter

# 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, Batch 1 loss: 4.969561
```

```
Epoch 1, Batch 101 loss: 3.953213
Epoch 1, Batch 201 loss: 2.987026
Epoch 1, Batch 301 loss: 2.602097
Epoch: 1        Training Loss: 2.524087        Validation Loss: 0.835076
Validation loss decreased (inf --> 0.835076).  Saving model ...
Epoch 2, Batch 1 loss: 1.290344
Epoch 2, Batch 101 loss: 1.503408
Epoch 2, Batch 201 loss: 1.432817
Epoch 2, Batch 301 loss: 1.446790
Epoch: 2        Training Loss: 1.452079        Validation Loss: 0.819379
Validation loss decreased (0.835076 --> 0.819379).  Saving model ...
Epoch 3, Batch 1 loss: 0.889915
Epoch 3, Batch 101 loss: 1.217222
Epoch 3, Batch 201 loss: 1.263864
Epoch 3, Batch 301 loss: 1.296778
Epoch: 3        Training Loss: 1.334217        Validation Loss: 0.724161
Validation loss decreased (0.819379 --> 0.724161).  Saving model ...
Epoch 4, Batch 1 loss: 0.973336
Epoch 4, Batch 101 loss: 1.316749
Epoch 4, Batch 201 loss: 1.348248
Epoch 4, Batch 301 loss: 1.340886
Epoch: 4        Training Loss: 1.344624        Validation Loss: 0.941000
Epoch 5, Batch 1 loss: 1.575531
Epoch 5, Batch 101 loss: 1.285066
Epoch 5, Batch 201 loss: 1.310887
Epoch 5, Batch 301 loss: 1.316084
Epoch: 5        Training Loss: 1.319014        Validation Loss: 0.767244
Epoch 6, Batch 1 loss: 0.637768
Epoch 6, Batch 101 loss: 1.211399
Epoch 6, Batch 201 loss: 1.215844
Epoch 6, Batch 301 loss: 1.261168
Epoch: 6        Training Loss: 1.265057        Validation Loss: 0.800590
Epoch 7, Batch 1 loss: 1.208344
Epoch 7, Batch 101 loss: 1.094293
Epoch 7, Batch 201 loss: 1.128990
Epoch 7, Batch 301 loss: 1.204599
Epoch: 7        Training Loss: 1.224074        Validation Loss: 0.731082
Epoch 8, Batch 1 loss: 1.237505
Epoch 8, Batch 101 loss: 1.158067
Epoch 8, Batch 201 loss: 1.169969
Epoch 8, Batch 301 loss: 1.175279
Epoch: 8        Training Loss: 1.172624        Validation Loss: 0.651235
Validation loss decreased (0.724161 --> 0.651235).  Saving model ...
Epoch 9, Batch 1 loss: 0.964920
Epoch 9, Batch 101 loss: 1.159220
Epoch 9, Batch 201 loss: 1.143718
Epoch 9, Batch 301 loss: 1.164517
Epoch: 9        Training Loss: 1.174699        Validation Loss: 1.045676
```

```
Epoch 10, Batch 1 loss: 2.175784
Epoch 10, Batch 101 loss: 1.216078
Epoch 10, Batch 201 loss: 1.270727
Epoch 10, Batch 301 loss: 1.290237
Epoch: 10         Training Loss: 1.294921         Validation Loss: 0.739737
```

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 [12]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

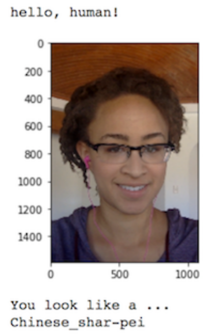
```
Test Loss: 0.943375


Test Accuracy: 80% (669/836)
```

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 [13]:
```
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(),
                                     standard_normalization])

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

In [18]:
```
### 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 loaders_transfer['train'].dataset


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()
```



Sample Human Output

```
            model.eval()
            idx = torch.argmax(model(img))
            return class_names[idx]
```

```
In [19]: for img in os.listdir('./images'):
             img_path = os.path.join('./images', img)
             predition = predict_breed_transfer(model_transfer, class_names, img_path)
             print("image name: {0},  predition breed: {1}".format(img_path, predition))
```

```
image name: ./images/Curly-coated_retriever_03896.jpg,  predition breed: Curly-coated retriever
image name: ./images/Brittany_02625.jpg,  predition breed: Brittany
image name: ./images/Welsh_springer_spaniel_08203.jpg,  predition breed: Irish red and white set
image name: ./images/sample_human_output.png,  predition breed: Chihuahua
image name: ./images/American_water_spaniel_00648.jpg,  predition breed: Chesapeake bay retrieve
image name: ./images/Labrador_retriever_06457.jpg,  predition breed: Labrador retriever
image name: ./images/Labrador_retriever_06449.jpg,  predition breed: Labrador retriever
image name: ./images/sample_cnn.png,  predition breed: Australian shepherd
image name: ./images/Labrador_retriever_06455.jpg,  predition breed: Labrador retriever
image name: ./images/sample_dog_output.png,  predition breed: Entlebucher mountain dog
```

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.

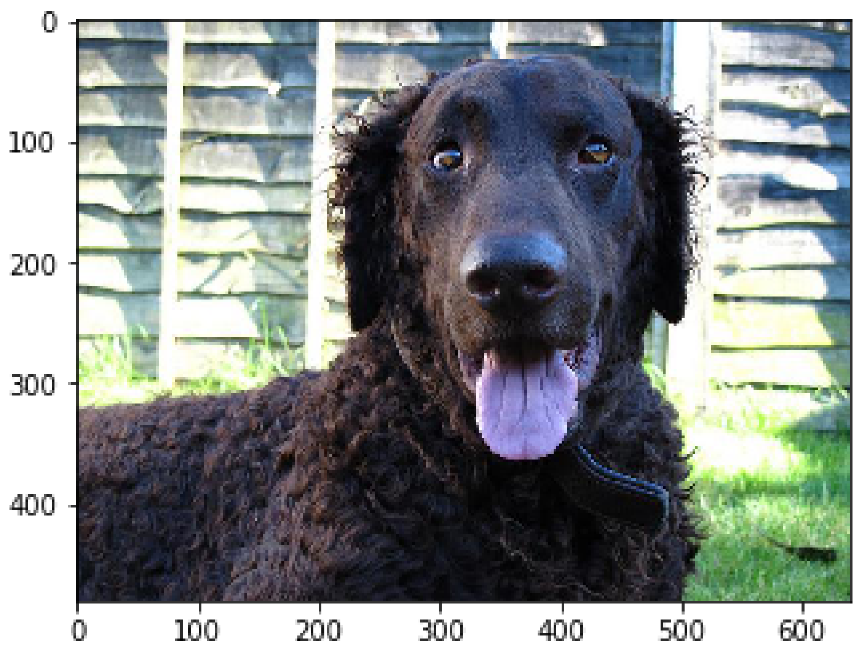
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

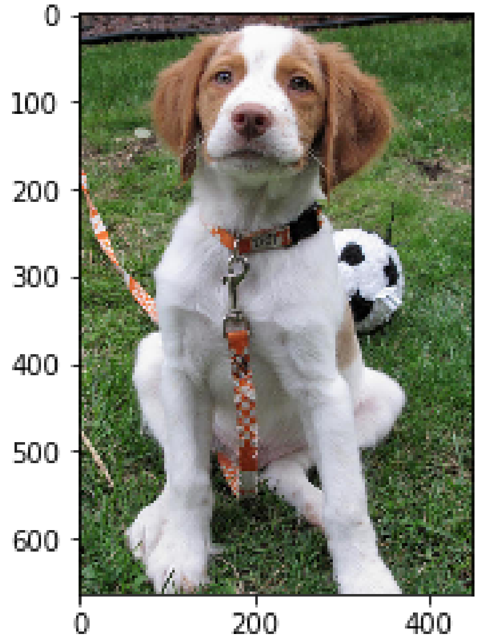
```
In [25]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
```

```
import matplotlib.pyplot as plt
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Dogs Detected!\nIt looks like a {0}".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(model_transfer, class_names, img_path)
        print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predic
    else:
        print("Error!")
```

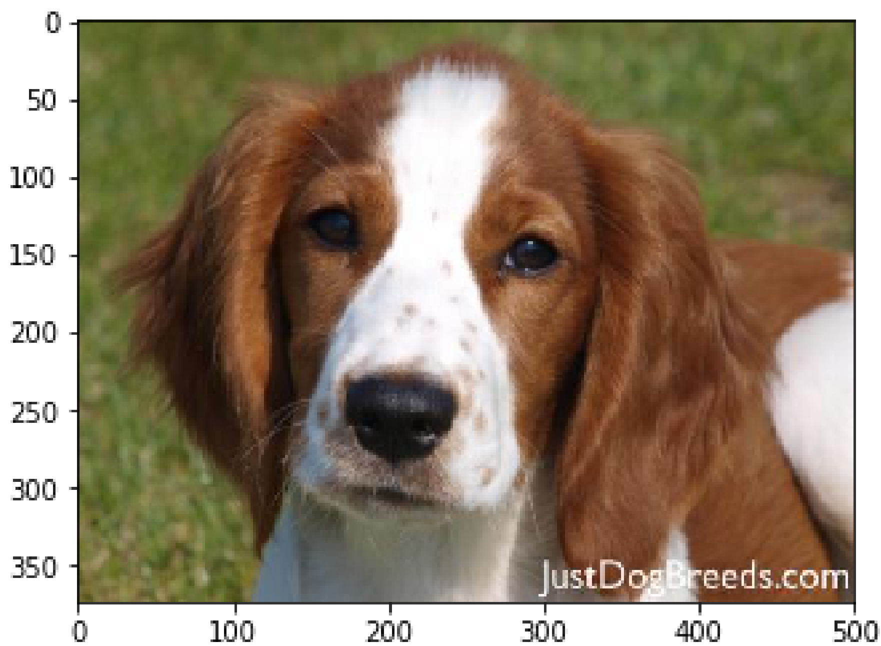
In [39]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
```



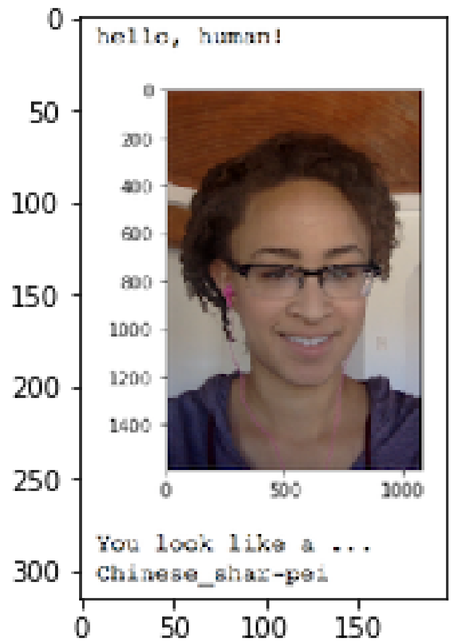
```
Dogs Detected!
It looks like a Curly-coated retriever
```



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

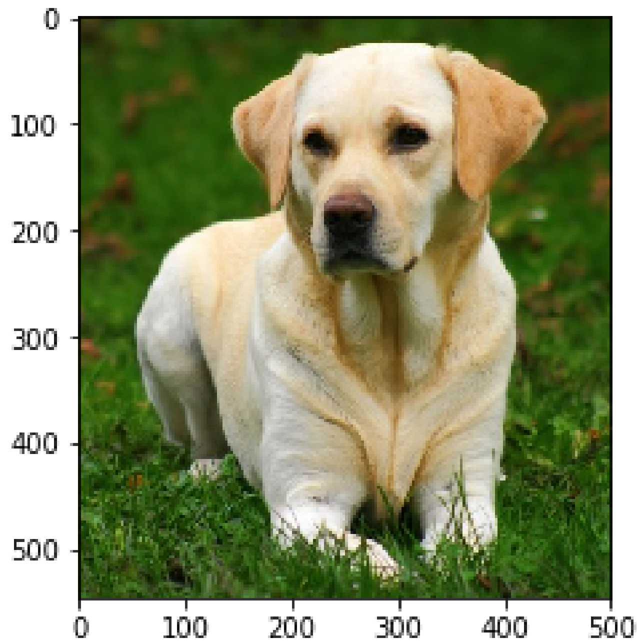


```
Dogs Detected!
It looks like a Irish red and white setter
```



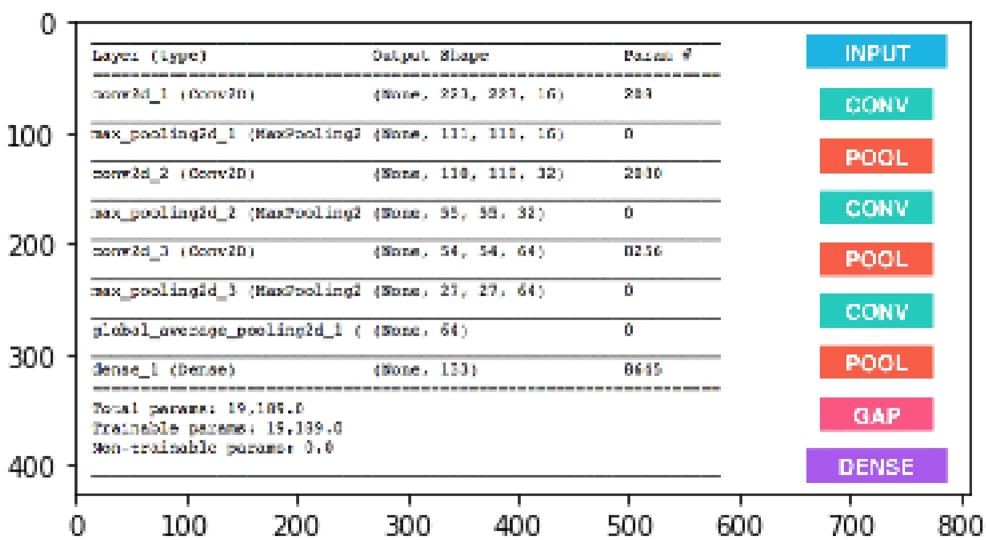

```
Hello, human!
If you were a dog..You may look like a Chihuahua
```

```
Dogs Detected!
It looks like a Chesapeake bay retriever
```

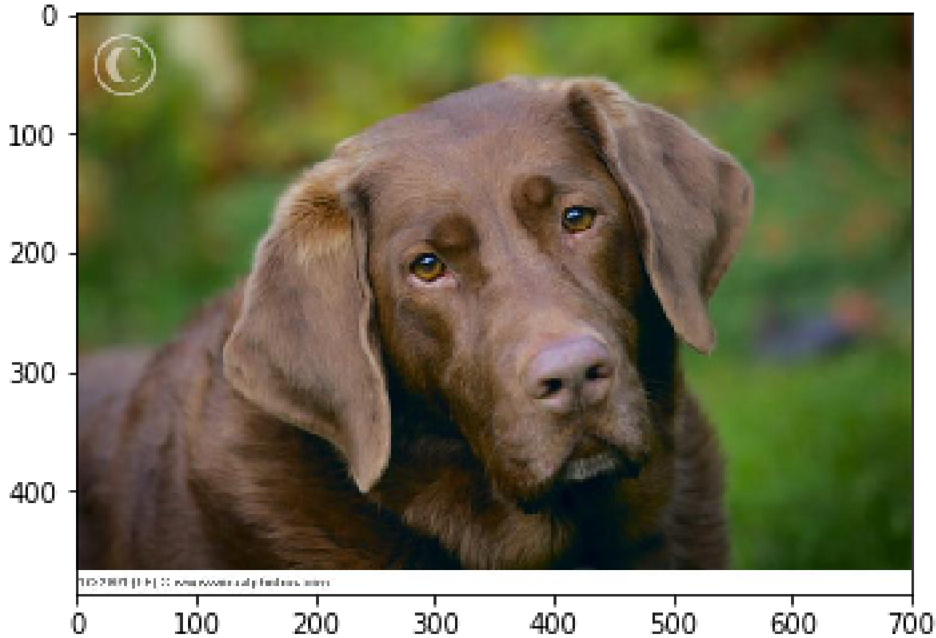


```
Dogs Detected!
It looks like a Labrador retriever
```

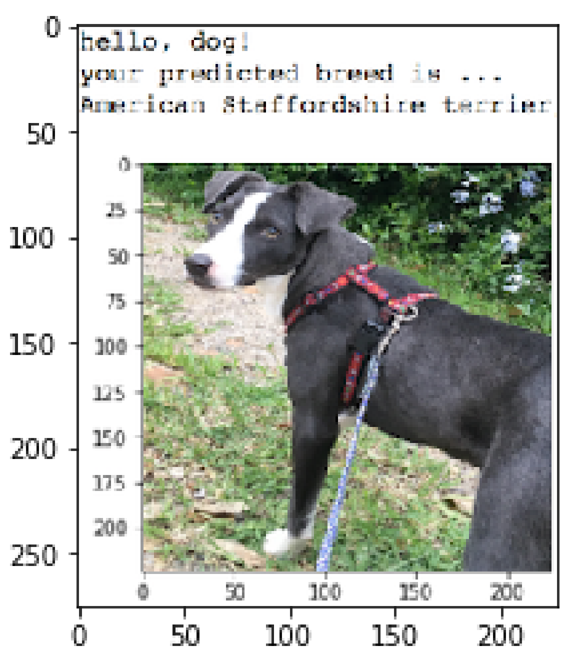
Dogs Detected!
It looks like a Labrador retriever



Error!



Dogs Detected!
It looks like a Labrador retriever



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

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.

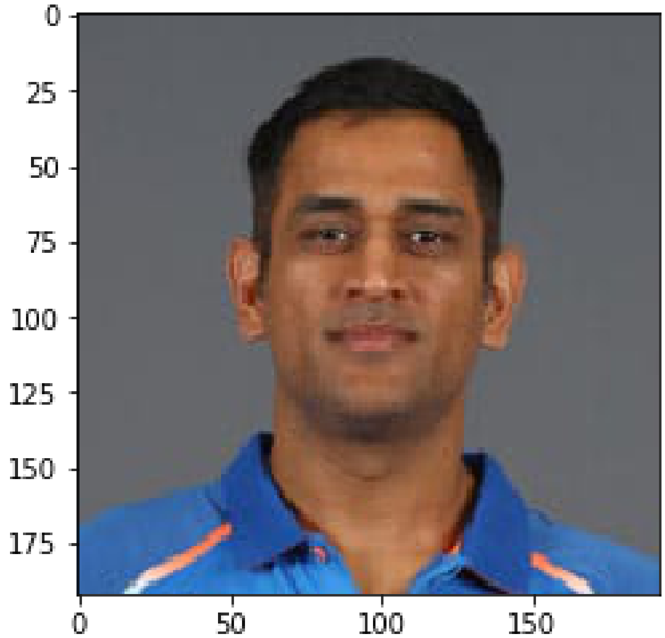
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

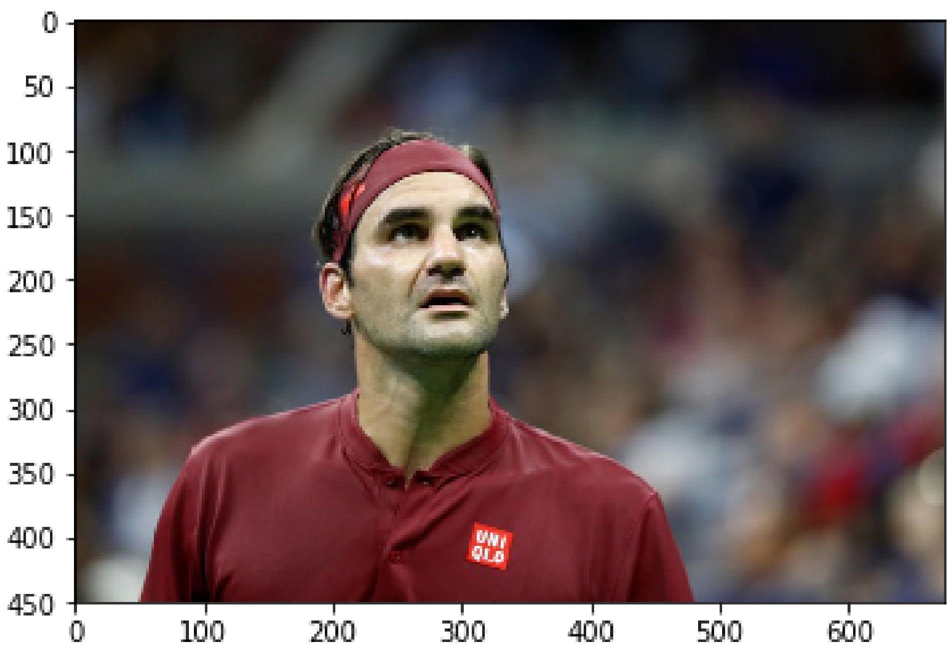
Ouput has come out really well as expected but we can still improve the model by following ways 1. Since deep learning is data hungry more the data better the model 2. We can change the hyperparamter to have better accurate models by have right inital weights,add more FC layers, dropouts and data augmentation 3. We can combine various models output by voting to predict the correct value.

```
In [44]: human = ['./my_images/human_1.jpg', './my_images/human_2.jpg', './my_images/human_3.jpe
         dogs = ['./my_images/dog_Irish_terrier.jpg', './my_images/dog_Gordon_setter.jpg', './my
```

```
In [45]: ## 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[:3], dogs[:3])):
             run_app(file)
```

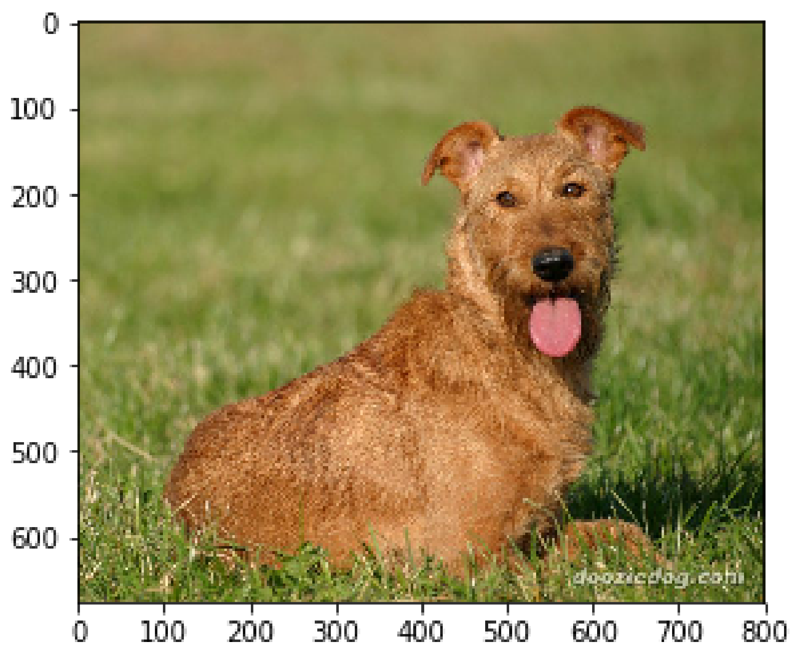


Hello, human!
If you were a dog..You may look like a Australian shepherd

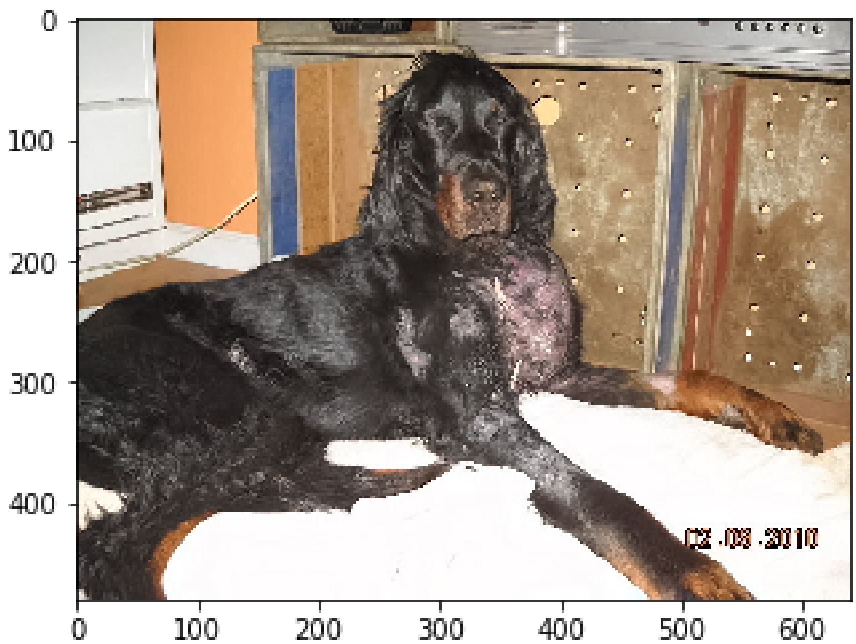


```
Hello, human!
If you were a dog..You may look like a Alaskan malamute
```

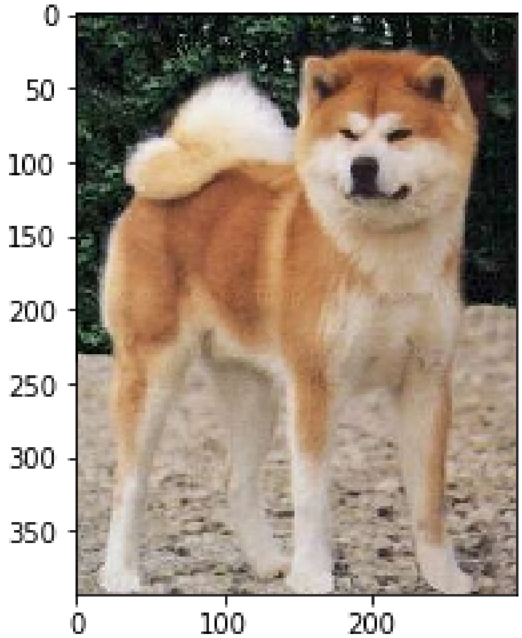

```
Hello, human!
If you were a dog..You may look like a Australian shepherd
```



```
Dogs Detected!
It looks like a Irish terrier
```




```
Dogs Detected!
It looks like a Gordon setter
```




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