Convolutional Neural Networks

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.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

Sample Dog Output

In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

- Step 0: Import Datasets
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Write your Algorithm
- Step 6: Test Your Algorithm

Step 0: Import Datasets

Make sure that you've downloaded the required human and dog datasets:

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages`.
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). 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 (http://www.7-zip.org/) 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 [4]:
```
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))
```

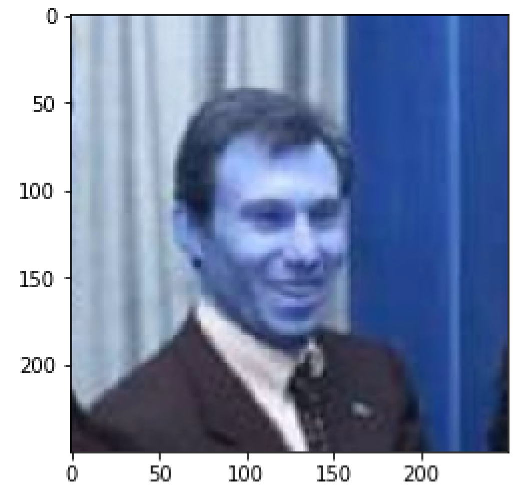
```
There are 13233 total human images.
There are 8351 total dog images.
```

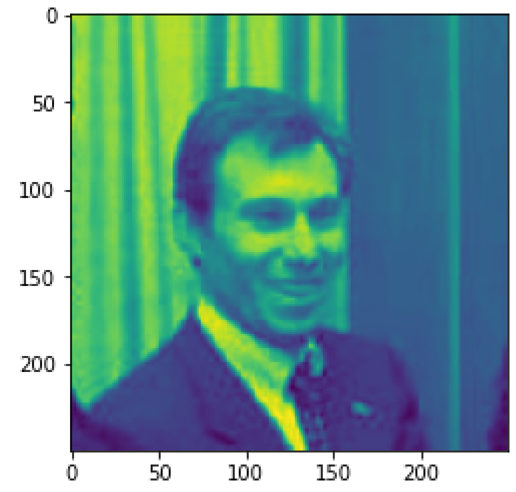
Step 1: Detect Humans

In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

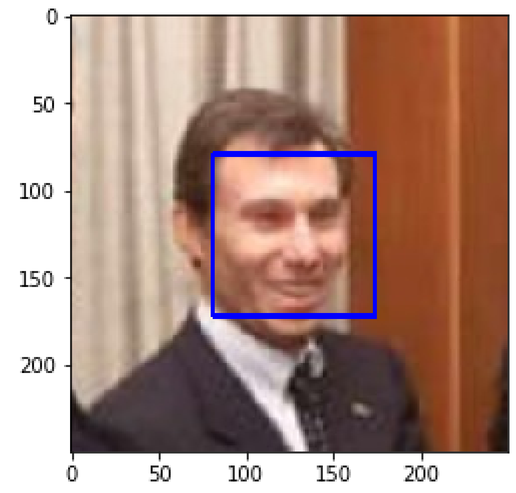
OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). We have downloaded one of these detectors and stored it in the `haarcascades` directory. In the next code cell, we demonstrate how to use this detector to find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

In [11]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

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

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

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 [12]:
```
# 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
```

(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 [13]:
```
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.
humanCount = 0
dogCount = 0
for face in human_files_short:
    if(face_detector(face)):
        humanCount+=1

for dog in dog_files_short:
    if(face_detector(dog)):
        dogCount+=1

print("The percentage of the first 100 images in human file have a detec
ted human face is {}%.".format(humanCount))
print("The percentage of the first 100 images in dog file have a detecte
d dog face is {}%.".format(dogCount))
```

```
The percentage of the first 100 images in human file have a detected hu
man face is 99%.
The percentage of the first 100 images in dog file have a detected dog
face is 18%.
```

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 [16]:
```
### (Optional)
### TODO: Test performance of another face detection algorithm.
### Feel free to use as many code cells as needed.
# Test the entire human image and dog image.
humanCount = 0
dogCount = 0
human_files_short2 = human_files[:500]
dog_files_short2 = dog_files[:500]
for face in human_files_short2:
    if(face_detector(face)):
        humanCount+=1

for dog in dog_files_short2:
    if(face_detector(dog)):
        dogCount+=1

print("The percentage of the first 500 images in human file have a detec
ted human face is {}%.".format(humanCount/len(human_files_short2)*100))
print("The percentage of the first 500 images in dog file have a detecte
d dog face is {}%.".format(dogCount/len(dog_files_short2)*100))
```

```
The percentage of the first 500 images in human file have a detected hu
man face is 98.6%.
The percentage of the first 500 images in dog file have a detected dog
face is 10.8%.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model (http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), 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 (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [7]:
```
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.

(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 (http://pytorch.org/docs/stable/torchvision/models.html).

In [5]:

```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

    Args:
        img_path: path to an image

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

    ## 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 = Image.open(img_path).convert('RGB')
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.2
29, 0.224, 0.225])
    img_transform = transforms.Compose([
                        transforms.Resize(size=(224, 224)),    #VGG16 is
trained on (244,244) images
                        transforms.ToTensor(),
                        normalize])
    img = img_transform(img)[:3,:,:].unsqueeze(0)
    if use_cuda:
        img = img.cuda()
    output = VGG16(img)

    return torch.max(output,1)[1].item()
```

In [8]:
```
sample_output = VGG16_predict(dog_files[20])
print(sample_output)
```

```
215
```

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), 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 [9]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    ## TODO: Complete the function.
    predict = VGG16_predict(img_path)
    return predict>= 151 and predict<=268 # true/false
```

(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 [18]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

predict_human_face = list(map(dog_detector,human_files_short))
print('The percentage of the image in human_files_files have a detected
 dog is {}.'.format(sum(predict_human_face)/len(human_files_short)))

predict_dog = list(map(dog_detector,dog_files_short))
print('The percentage of the image in dog_files_files have a detected do
g is {}.'.format(sum(predict_dog)/len(dog_files_short)))
```

```
The percentage of the image in human_files_files have a detected dog is
0.0.
The percentage of the image in dog_files_files have a detected dog is
0.97.
```

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 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador	Black 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!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders (http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) 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 (http://pytorch.org/docs/stable/torchvision/datasets.html) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms (http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

```
In [23]: os.getcwd()
```

```
Out[23]: '/Users/shuyihuo/Desktop/2020_spring/Udancity/project-dog-classificatio
         n'
```

In [31]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/Users/shuyihuo/Desktop/2020_spring/Udancity/project-dog-cla
ssification/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')

batch_size = 20
num_workers = 0

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])

preprocess_data = {'train': transforms.Compose([transforms.RandomResized
Crop(224),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ToTensor(),
                                 normalize]),
                   'valid': transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 normalize]),
                   'test': transforms.Compose([transforms.Resize(size=(2
24,224)),
                                 transforms.ToTensor(),
                                 normalize])
                  }

train_data = datasets.ImageFolder(train_dir, transform=preprocess_data[
'train'])
valid_data = datasets.ImageFolder(valid_dir, transform=preprocess_data[
'valid'])
test_data = datasets.ImageFolder(test_dir, transform=preprocess_data['te
st'])

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
}
```

```
In [40]: loaders_scratch
```

```
Out[40]: {'train': <torch.utils.data.dataloader.DataLoader at 0x12422d310>,
          'valid': <torch.utils.data.dataloader.DataLoader at 0x156801450>,
          'test': <torch.utils.data.dataloader.DataLoader at 0x1565aacd0>}
```

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: I use the RandomiResizedCrop for trainig data which resized all the training images to (224,224), and makes a random crop of the original image so that our model is able to learn complex variations in data. I have flipped the data horizontally using RandomHorizontalFlip(p=.5) which will flip half images horizontally to add more variation to original training data. I Used CenterCrop of size (224,224) for validation data, as most of the images have dog in the center thus this will help in good validation accuracy and Resized test images to (224,224), no other transformation done here as we will test our model on raw data.

(IMPLEMENTATION) Model Architecture

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

In [32]:
```
import torch.nn as nn
import torch.nn.functional as F

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()
        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 36, 3, padding=1)
        self.conv2 = nn.Conv2d(36, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(28*28*128, 512)
        self.fc2 = nn.Linear(512, 133)
        self.dropout = nn.Dropout(0.25)
        self.batch_norm = nn.BatchNorm1d(512)

    def forward(self, x):
        ## Define forward behavior
        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 = x.view(-1, 28*28*128)

        x = F.relu(self.batch_norm(self.fc1(x)))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))


        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer: The model has 3 convolutional layers. All convolutional layers has kernal size of 3 and stride 1. The first conv layer (conv1) have in_channels =3 and the final conv layer (conv3) produces an output size of 128.

ReLU activation function is used here. The pooling layer of (2,2) is used which will reduce the input size by 2. We have two fully connected layers that finally produces 133 dimensional output. A dropout of 0.25 is added to avoid overfitting.

In [34]:
```
model_scratch
```

Out[34]:
```
Net(
  (conv1): Conv2d(3, 36, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
  (conv2): Conv2d(36, 64, kernel_size=(3, 3), stride=(1, 1), 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, cei
l_mode=False)
  (fc1): Linear(in_features=100352, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
  (batch_norm): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
)
```

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/stable/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch` , and the optimizer as `optimizer_scratch` below.

In [49]:
```
import torch.optim as optim

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.02)
```

In [51]:
```
optimizer_scratch
```

Out[51]:
```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.02
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

In [50]:
```
criterion_scratch
```

Out[50]:
```
CrossEntropyLoss()
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

In [55]:
```
%%time
# the following import is required for training to be robust to truncate
d images
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save
_path):
    """returns trained model"""
    # initialize tracker for minimum validation loss
    #print('here') for test
    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_loss))
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.dat
a - train_loss))
            # watch training
            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.dat
a - valid_loss))
```

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

        ## TODO: save the model if validation loss has decreased
        if valid_loss < valid_loss_min:

            print('Validation loss decreased ({:.6f} --> {:.6f}). Saving
the model'.format(valid_loss_min, valid_loss))
            torch.save(model.state_dict(), save_path)
            valid_loss_min = valid_loss

    # return trained model
    return model


# train the model
model_scratch = train(15, loaders_scratch, model_scratch, optimizer_scra
tch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
# print('hi') For test
# load the model that got the best validation accuracy
model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
here
Epoch 1, Batch 1 loss: 4.674943
Epoch 1, Batch 101 loss: 4.367715
Epoch 1, Batch 201 loss: 4.353889
Epoch 1, Batch 301 loss: 4.357242
Epoch: 1        Training Loss: 4.358363         Validation Loss: 4.1597
22
Validation loss decreased (inf --> 4.159722). Saving the model
Epoch 2, Batch 1 loss: 4.538615
Epoch 2, Batch 101 loss: 4.209931
Epoch 2, Batch 201 loss: 4.243355
Epoch 2, Batch 301 loss: 4.243955
Epoch: 2        Training Loss: 4.246533         Validation Loss: 4.1266
27
Validation loss decreased (4.159722 --> 4.126627). Saving the model
Epoch 3, Batch 1 loss: 4.342855
Epoch 3, Batch 101 loss: 4.131512
Epoch 3, Batch 201 loss: 4.153565
Epoch 3, Batch 301 loss: 4.158195
Epoch: 3        Training Loss: 4.158885         Validation Loss: 3.9639
79
Validation loss decreased (4.126627 --> 3.963979). Saving the model
Epoch 4, Batch 1 loss: 4.264231
Epoch 4, Batch 101 loss: 4.107455
Epoch 4, Batch 201 loss: 4.077886
Epoch 4, Batch 301 loss: 4.064068
Epoch: 4        Training Loss: 4.062500         Validation Loss: 4.0559
01
Epoch 5, Batch 1 loss: 3.721970
Epoch 5, Batch 101 loss: 3.977966
Epoch 5, Batch 201 loss: 3.986281
Epoch 5, Batch 301 loss: 3.985107
Epoch: 5        Training Loss: 3.985501         Validation Loss: 3.7987
81
Validation loss decreased (3.963979 --> 3.798781). Saving the model
Epoch 6, Batch 1 loss: 3.839380
Epoch 6, Batch 101 loss: 3.930434
Epoch 6, Batch 201 loss: 3.912186
Epoch 6, Batch 301 loss: 3.910295
Epoch: 6        Training Loss: 3.905447         Validation Loss: 3.7144
21
Validation loss decreased (3.798781 --> 3.714421). Saving the model
Epoch 7, Batch 1 loss: 3.632780
Epoch 7, Batch 101 loss: 3.798398
Epoch 7, Batch 201 loss: 3.829486
Epoch 7, Batch 301 loss: 3.828679
Epoch: 7        Training Loss: 3.829609         Validation Loss: 3.8615
32
Epoch 8, Batch 1 loss: 3.762219
Epoch 8, Batch 101 loss: 3.737539
Epoch 8, Batch 201 loss: 3.754346
Epoch 8, Batch 301 loss: 3.754147
Epoch: 8        Training Loss: 3.755600         Validation Loss: 3.5704
80
Validation loss decreased (3.714421 --> 3.570480). Saving the model
Epoch 9, Batch 1 loss: 3.330162
Epoch 9, Batch 101 loss: 3.648473
Epoch 9, Batch 201 loss: 3.678407
Epoch 9, Batch 301 loss: 3.677758
Epoch: 9        Training Loss: 3.678039         Validation Loss: 3.6247
11
Epoch 10, Batch 1 loss: 3.525043
Epoch 10, Batch 101 loss: 3.607589
Epoch 10, Batch 201 loss: 3.636458
Epoch 10, Batch 301 loss: 3.629381
Epoch: 10       Training Loss: 3.636716         Validation Loss: 3.5614
88
Validation loss decreased (3.570480 --> 3.561488). Saving the model
Epoch 11, Batch 1 loss: 2.737492
Epoch 11, Batch 101 loss: 3.565547
Epoch 11, Batch 201 loss: 3.550848
Epoch 11, Batch 301 loss: 3.547373
Epoch: 11       Training Loss: 3.554372         Validation Loss: 3.3630
80
Validation loss decreased (3.561488 --> 3.363080). Saving the model
Epoch 12, Batch 1 loss: 3.902684
Epoch 12, Batch 101 loss: 3.430769
Epoch 12, Batch 201 loss: 3.445555
Epoch 12, Batch 301 loss: 3.457609
Epoch: 12       Training Loss: 3.467670         Validation Loss: 3.4570
30
Epoch 13, Batch 1 loss: 3.699418
Epoch 13, Batch 101 loss: 3.432821
Epoch 13, Batch 201 loss: 3.442701
Epoch 13, Batch 301 loss: 3.461972
Epoch: 13       Training Loss: 3.455570         Validation Loss: 3.2868
16
Validation loss decreased (3.363080 --> 3.286816). Saving the model
Epoch 14, Batch 1 loss: 3.548006
Epoch 14, Batch 101 loss: 3.352729
Epoch 14, Batch 201 loss: 3.354428
Epoch 14, Batch 301 loss: 3.375285
Epoch: 14       Training Loss: 3.379846         Validation Loss: 3.4055
31
Epoch 15, Batch 1 loss: 3.536599
Epoch 15, Batch 101 loss: 3.299728
Epoch 15, Batch 201 loss: 3.312292
Epoch 15, Batch 301 loss: 3.333312
Epoch: 15       Training Loss: 3.340187         Validation Loss: 3.2215
23
Validation loss decreased (3.286816 --> 3.221523). Saving the model
hi
CPU times: user 4h 28min 51s, sys: 1h 1min 46s, total: 5h 30min 38s
Wall time: 6h 42min 7s
```

```
Out[55]: <All keys matched successfully>
```

(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 [56]:
```
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 t
he 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 - te
st_loss))
        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]
        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred)))
.cpu().numpy())
        total += data.size(0)

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

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

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

```
Test Loss: 3.466868


Test Accuracy: 18% (151/836)
```

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.

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [57]:
```
## TODO: Specify data loaders
loaders_transfer = loaders_scratch.copy()
```

(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 [63]:
```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture
model_transfer = models.resnet101(pretrained=True)

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

model_transfer.fc = nn.Linear(2048, 133, bias=True)     #replacing last f
c with custom fully-connected layer which should output 133 sized vector

fc_parameters = model_transfer.fc.parameters()          #extracting fc pa
rameters
for param in fc_parameters:
    param.requires_grad = True

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

In [64]:
```
model_transfer
```

Out[64]:
```
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_ru
nning_stats=True)
  (relu): ReLU(inplace=True)
  (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=F
alse)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=Fa
lse)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, trac
k_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (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, trac
k_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), paddin
g=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, trac
k_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (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, tra
ck_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=F
alse)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, tra
ck_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (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, tra
ck_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (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, tra
ck_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, tra
ck_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, tra
ck_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (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, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
      (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, tr
ack_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (6): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (7): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (8): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (9): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (10): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (11): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (12): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (13): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
```

```
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (14): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (15): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (16): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (17): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (18): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (19): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (20): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (21): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
```

```
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (22): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
      (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, tr
ack_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
```

```
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padd
ing=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, tra
ck_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bia
s=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, tr
ack_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=133, bias=True)
)
```

In [ ]:

```
```

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

Answer: I used the resnet101 architecture which is pre-trained on Imagenet dataset, The architecture is 101 layers deep, within just 5 epochs, the model got 84% accuracy. If we train for more epochs, the accuracy can be significantly improved. Steps:Import pre-trained resnet101 model, Change the out_features of fully connected layer to 133 to solve the classification problem, CrossEntropy loss function is chosen as loss function.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a [loss function (http://pytorch.org/docs/master/nn.html#loss-functions)](http://pytorch.org/docs/master/nn.html#loss-functions) and [optimizer (http://pytorch.org/docs/master/optim.html)](http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer` , and the optimizer as `optimizer_transfer` below.

In [65]:

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.02)
```

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'` .

In [66]:
```
# train the model
model_transfer = train(5, loaders_transfer, model_transfer, optimizer_tr
ansfer, criterion_transfer, use_cuda, 'model_transfer.pt')

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

```
here
Epoch 1, Batch 1 loss: 4.911633
Epoch 1, Batch 101 loss: 4.373168
Epoch 1, Batch 201 loss: 3.857408
Epoch 1, Batch 301 loss: 3.451298
Epoch: 1        Training Loss: 3.347613         Validation Loss: 1.6554
56
Validation loss decreased (inf --> 1.655456). Saving the model
Epoch 2, Batch 1 loss: 1.836044
Epoch 2, Batch 101 loss: 2.040964
Epoch 2, Batch 201 loss: 1.916585
Epoch 2, Batch 301 loss: 1.803925
Epoch: 2        Training Loss: 1.769905         Validation Loss: 0.9379
83
Validation loss decreased (1.655456 --> 0.937983). Saving the model
Epoch 3, Batch 1 loss: 1.308915
Epoch 3, Batch 101 loss: 1.436004
Epoch 3, Batch 201 loss: 1.369036
Epoch 3, Batch 301 loss: 1.356029
Epoch: 3        Training Loss: 1.344865         Validation Loss: 0.6914
71
Validation loss decreased (0.937983 --> 0.691471). Saving the model
Epoch 4, Batch 1 loss: 0.987667
Epoch 4, Batch 101 loss: 1.127870
Epoch 4, Batch 201 loss: 1.163524
Epoch 4, Batch 301 loss: 1.134126
Epoch: 4        Training Loss: 1.122588         Validation Loss: 0.5803
69
Validation loss decreased (0.691471 --> 0.580369). Saving the model
Epoch 5, Batch 1 loss: 0.958909
Epoch 5, Batch 101 loss: 1.025054
Epoch 5, Batch 201 loss: 1.034012
Epoch 5, Batch 301 loss: 1.018946
Epoch: 5        Training Loss: 1.016042         Validation Loss: 0.5099
19
Validation loss decreased (0.580369 --> 0.509919). Saving the model
```

Out[66]:
```
<All keys matched successfully>
```

(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 [67]:
```
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.574042


Test Accuracy: 84% (709/836)
```

(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 [69]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.
data_transfer = loaders_transfer
# list of class names by index, i.e. a name can be accessed like class_n
ames[0]
class_names = [item[4:].replace("_", " ") for item in data_transfer['tra
in'].dataset.classes]

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    image = Image.open(img_path).convert('RGB')
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.2
29, 0.224, 0.225])
    transformations = transforms.Compose([transforms.Resize(size=(224, 2
24)),
                                          transforms.ToTensor(),
                                          normalize])

    transformed_image = transformations(image)[:3,:,:].unsqueeze(0)

    if use_cuda:
        transformed_image = transformed_image.cuda()

    output = model_transfer(transformed_image)

    pred_index = torch.max(output,1)[1].item()

    return class_names[pred_index]
```

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 `dog_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!

Sample Human Output

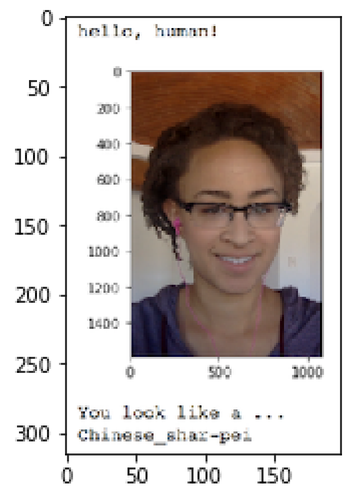
(IMPLEMENTATION) Write your Algorithm

In [76]:

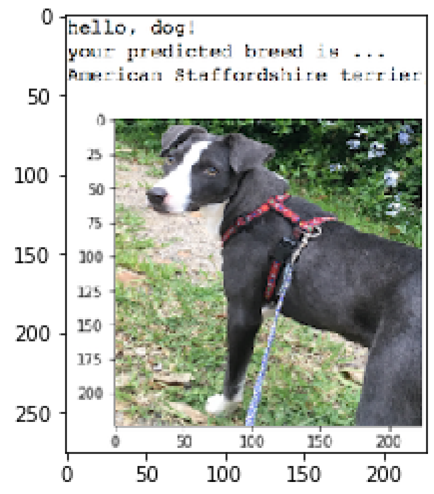
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def Show_image(img_path):
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    if face_detector(img_path):
        print ("Hello Human!")
        predicted_breed = predict_breed_transfer(img_path)
        print("This human looks like : ",predicted_breed)
        Show_image(img_path)

    elif dog_detector(img_path):
        print ("Hello Dog!")
        predicted_breed = predict_breed_transfer(img_path)
        print("This dog breed is : ",predicted_breed)
        Show_image(img_path)

    else:
        print ("Invalid image")
```

In [77]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
    print('')
```

```
Hello Human!
This human looks like :  Bichon frise
```



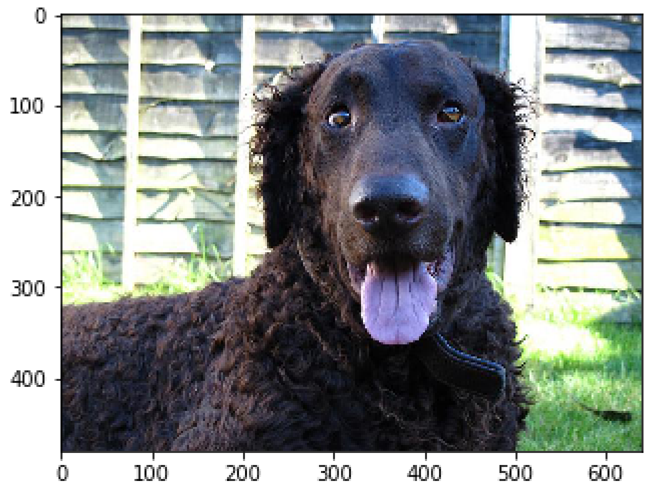
```
Hello Dog!
This dog breed is :  Great dane
```



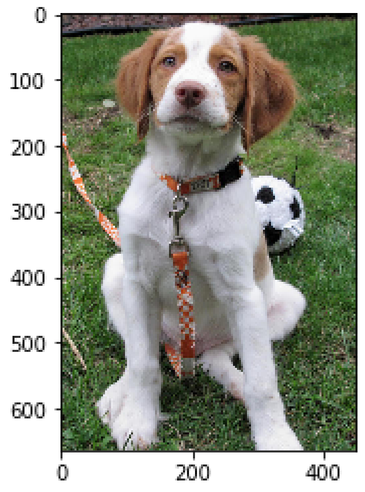
```
Hello Dog!
This dog breed is :  Labrador retriever
```

```
Hello Dog!
This dog breed is :  American water spaniel
```

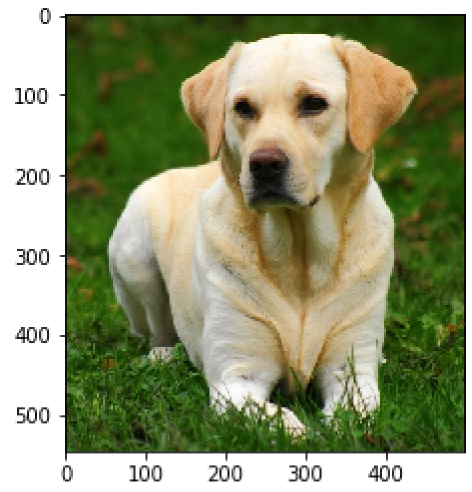
```
Hello Dog!
This dog breed is :  Curly-coated retriever
```



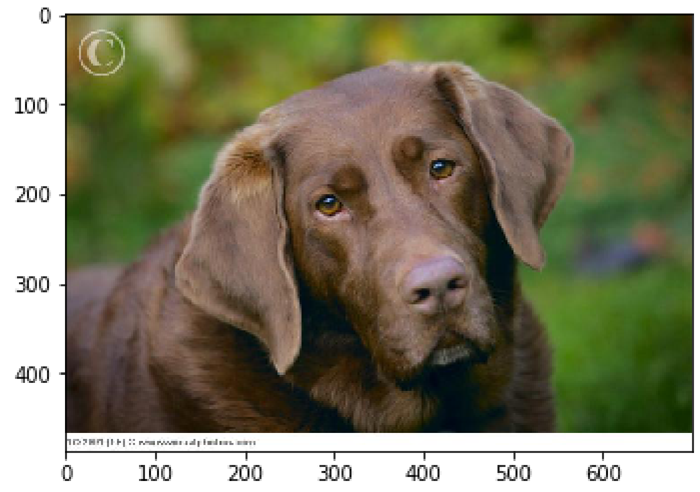
```
Hello Dog!
This dog breed is :  Brittany
```



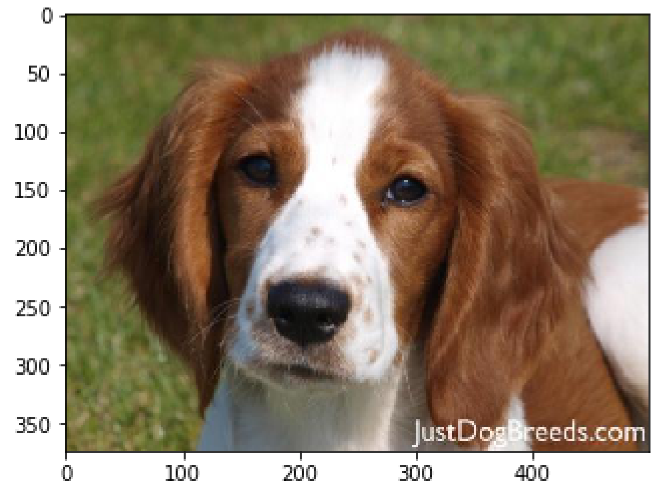
```
Hello Dog!
This dog breed is :  Labrador retriever
```



```
Hello Dog!
This dog breed is :  Labrador retriever
```



```
Invalid image

Hello Dog!
This dog breed is :  Welsh springer spaniel
```



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?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

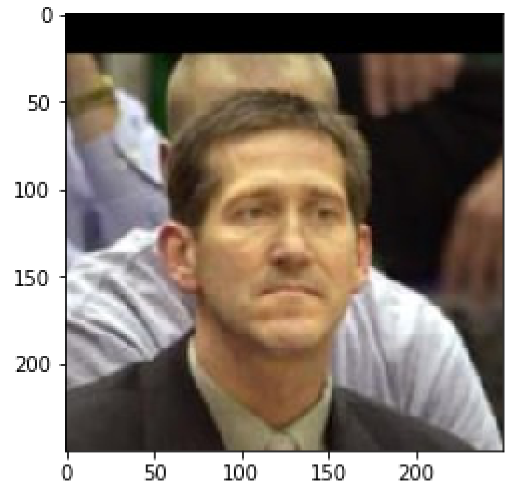
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

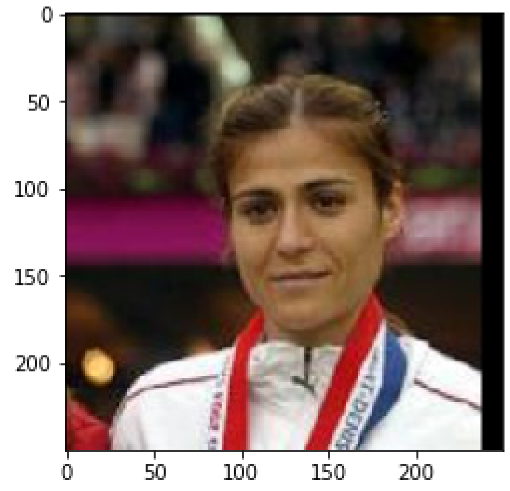
Answer: (Three possible points for improvement) I think the output is better than I expected and the pretrained model from transfer learning is better than custom cnn model from scratch. Three points for improvement: 1. Hyperparameter tuning might help to choose the better parameter for the model to improve performance. 2. Training more dataset might also help to improve the accuracy. 3. Using more powerful pretrained models and boosting these models to choose the best the model for detection.

In [78]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

## suggested code, below
for file in np.hstack((human_files[3:6], dog_files[3:6])):
    run_app(file)
```

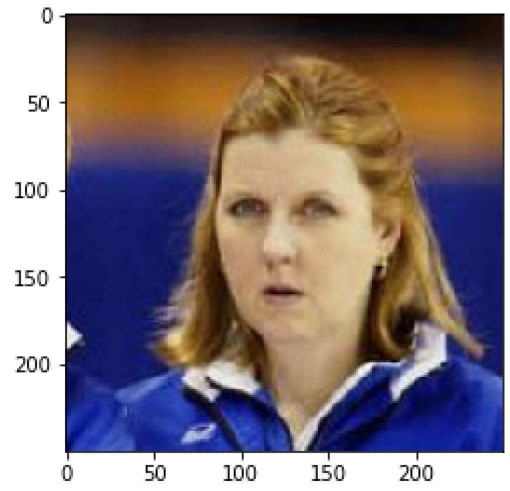
```
Hello Human!
This human looks like :  Australian shepherd
```



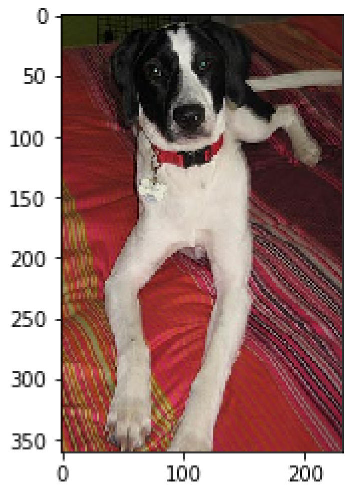
```
Hello Human!
This human looks like :  Australian shepherd
```



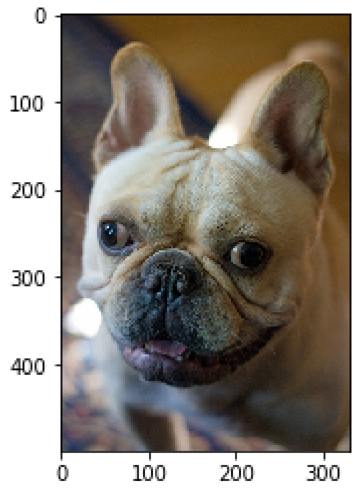
```
Hello Human!
This human looks like :  Australian shepherd
```



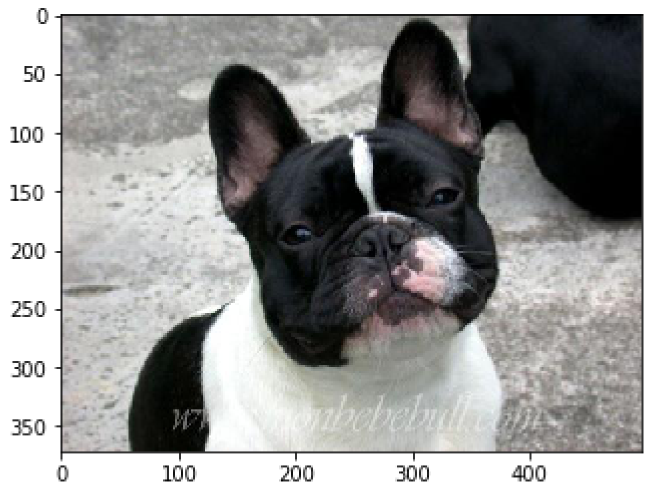
```
Hello Dog!
This dog breed is :  Pointer
```



```
Hello Dog!
This dog breed is :  French bulldog
```



```
Hello Dog!
This dog breed is :  French bulldog
```



In [ ]: