dog_app

May 29, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.

Step 0: Import Datasets

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

Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

- Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dog_images`.
- Download the human dataset. Unzip the folder and place it in the home directory, at location `/lfw`.

Note: If you are using a Windows machine, you are encouraged to use 7zip to extract the folder.

In the code cell below, we save the file paths for both the human (LFW) dataset and dog dataset in the numpy arrays `human_files` and `dog_files`.

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

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

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

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

1.1.1 EDA

```
In [2]: #Data Exploration

        print('Total number of human face images:',len(glob("/data/lfw/*/*")))
        print('Total number of human face folders:', len(glob('/data/lfw/*')))
        print("Total numner of folders in 'dog_images:'", len(glob('/data/dog_images/*')))
        print("Folders in 'dog_images':",end=' ')
        print(*[x.split('/')[-1] for x in glob('/data/dog_images/*')], sep=',')
        print("Total folders(breed classes) in 'train, test, valid'",len(glob("/data/dog_images/
        print('Total images in /dog_images/train :',len(glob("/data/dog_images/train/*/*")))
        print('Total images in /dog_images/test :',len(glob("/data/dog_images/test/*/*")))
        print('Total images in /dog_images/valid :',len(glob("/data/dog_images/valid/*/*")))
```

```
Total number of human face images: 13233
Total number of human face folders: 5749
Total numner of folders in 'dog_images:' 3
Folders in 'dog_images': train,test,valid
Total folders(breed classes) in 'train, test, valid' 133
Total images in /dog_images/train : 6680
Total images in /dog_images/test : 836
Total images in /dog_images/valid : 835
```

```
In [3]: num_images_per_folder = [len(glob(x+'/*')) for x in glob("/data/dog_images/train/*")]
        name_folder = [x.split('/')[-1] for x in glob("/data/dog_images/train/*")]

In [4]: avg_no_images = sum(num_images_per_folder)/len(num_images_per_folder)
        avg_no_images

Out[4]: 50.225563909774436

In [5]: import matplotlib.pyplot as plt
        plt.figure(figsize=(16,8))
        plt.bar(name_folder, num_images_per_folder)
        plt.xticks(rotation='vertical')
        plt.axhline(avg_no_images, color='black')
        plt.xlabel('Dog breed folder')
        plt.ylabel('Number of images in folder')
        plt.title("Number of images per class- Black line corresponds to average number of image
        plt.show()

<matplotlib.figure.Figure at 0x7f1a4c7ac0b8>
```

Step 1: Detect Humans

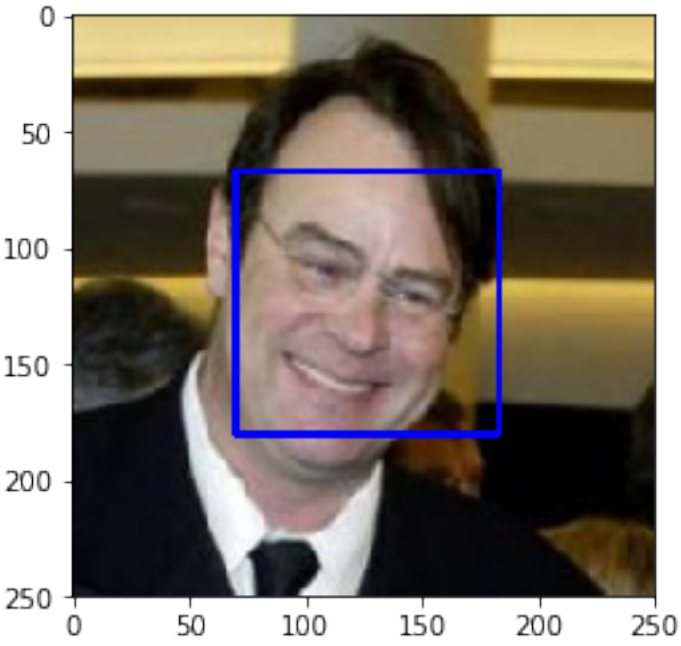
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [6]: 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()
```

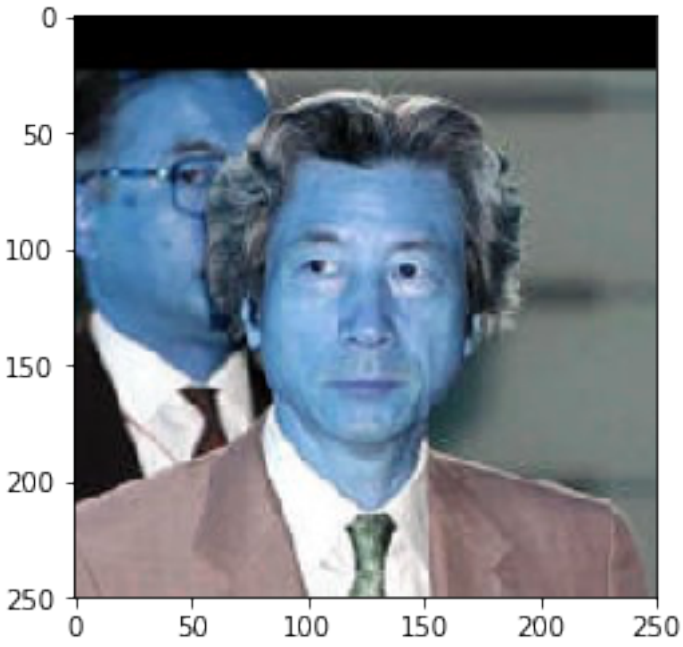
```
Number of faces detected: 1
```

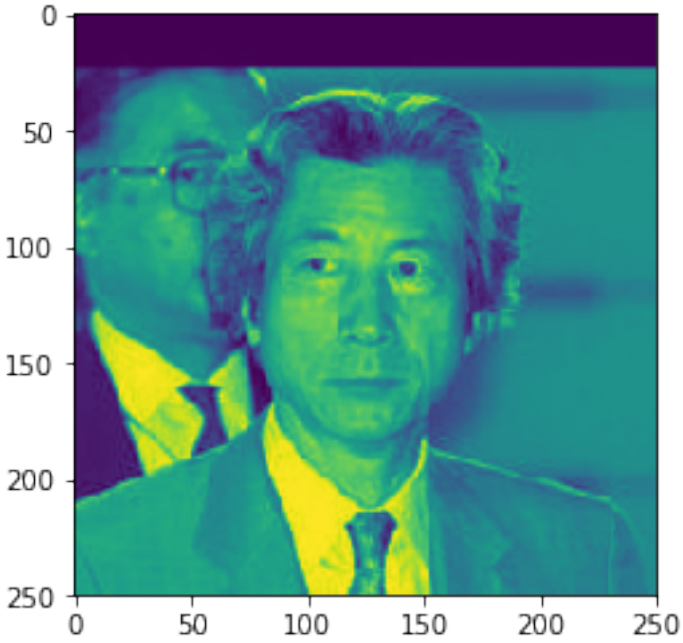


1.1.2 Playing around with faces

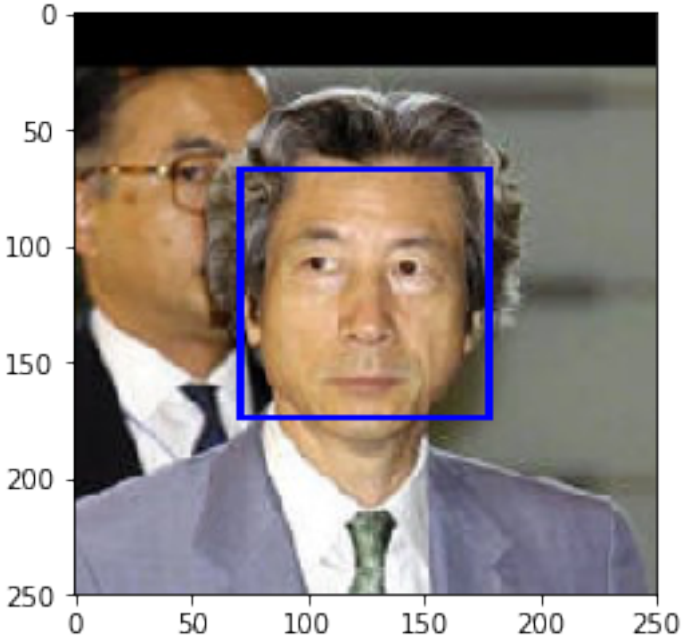
```
In [7]: img = cv2.imread(human_files[3102])
        plt.imshow(img)
        plt.show()
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        plt.imshow(gray)
        plt.show()

        # find faces in image
        faces = face_cascade.detectMultiScale(gray)     #gives bounding box coordinates

        # 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
```




```
In [8]: faces
```

```
Out[8]: array([[ 71,  67, 107, 107]], dtype=int32)
```

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.

1.1.3 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 [9]: # 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
```

1.1.4 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer:

- 98% images in human_files_short(first 100 images in human_files) have been detected as human face
- 17% images in dog_files_short have been incorrectly detected as human face, where as they should not have been detected.

```
In [10]: from tqdm import tqdm

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

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

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

```
        human_files_short_tq = tqdm(human_files[:100])
        dog_files_short_tq = tqdm(dog_files[:100])

        human_faces_in_humanData = list(map(face_detector, human_files_short_tq))
        human_faces_in_dogData = list(map(face_detector, dog_files_short_tq))

        print('Percentage of images in human_files that have detected human face:', (sum(human_
        print('Percentage of images in dog_files that have detected human face(incorrect detect
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
100%|| 100/100 [00:02<00:00, 36.31it/s]
100%|| 100/100 [00:32<00:00,  7.24it/s]

Percentage of images in human_files that have detected human face: 98.0
Percentage of images in dog_files that have detected human face(incorrect detection): 17.0
```

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

         def face_detector_alt(img_path):

             face_cascade_alt = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt2
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade_alt.detectMultiScale(gray)
             return len(faces) > 0

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

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

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

         human_files_short_tq = tqdm(human_files[:100])
         dog_files_short_tq = tqdm(dog_files[:100])
```

```
        human_faces_in_humanData = list(map(face_detector_alt, human_files_short_tq))
        human_faces_in_dogData = list(map(face_detector_alt, dog_files_short_tq))

        print('Percentage of images in human_files that have detected human face:', (sum(human_
        print('Percentage of images in dog_files that have detected human face(incorrect detect
```

```
  0%|          | 0/100 [00:00<?, ?it/s]
100%|| 100/100 [00:03<00:00, 28.87it/s]
100%|| 100/100 [00:33<00:00,  3.00it/s]

Percentage of images in human_files that have detected human face: 100.0
Percentage of images in dog_files that have detected human face(incorrect detection): 21.0
```

Here is a comparison of the performance between haarcascade_frontalface_alt.xml and haarcascade_frontalface_alt2.xml:

haarcascade_frontalface_alt.xml:

- Humans detected in human_files_short: 98%
- Humans detected in dog_files_short: 17%

haarcascade_frontalface_alt2.xml:

- Humans detected in human_files_short: 100 %
- Humans detected in dog_files_short: 21%

You can see that we have a better performance with the haarcascade_frontalface_alt2.xml. So we have an accuracy of 100 percent. We have an increased percentage of recognized humans in the dog_files_short. But this is not a problem as we will recognize dogs with a different detector.

Step 2: Detect Dogs

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

1.1.5 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 [12]: 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.

1.1.6 (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 [13]: from PIL import Image
         import torchvision.transforms as transforms

         def load_transform_image(img_path):
             img = Image.open(img_path).convert('RGB')
             normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.22
             img_transform = transforms.Compose([
                                 transforms.Resize(size=(224, 224)),    #VGG16 is trained on (24
                                 transforms.ToTensor(),
                                 normalize])
             img = img_transform(img)[:3,:,:].unsqueeze(0)
             return img


         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

    image = load_transform_image(img_path)

    if use_cuda:
        image = image.cuda()
    output = VGG16(image)

    ## Return the *index* of the predicted class for that image
    #     return np.argmax(output.detach().numpy()[0])
    #     return output
    # following return statement can also be used
    return torch.max(output,1)[1].item()
```

1.1.7 (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 [14]: #checking prediction on a sample image from dogs dataset
         sample_output = VGG16_predict(dog_files[20])     #change VGG16_predict to return 'outpu

In [15]: sample_output

Out[15]: 243

In [16]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             prediction = VGG16_predict(img_path)
             return (prediction>=151 and prediction<=268)
```

1.1.8 (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:

- 1% images of human_files_short have detected dog -> Incorrect Prediction
- 100% images in dog_files_short have detected dog correctly!

```
In [17]: pred = dog_detector(human_files[0])
```

```
In [18]: #sample unit test for dog_detector
         pred = dog_detector(human_files[0])
         assert(pred==False)

In [19]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         from tqdm import tqdm

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

         human_files_short_tq = tqdm(human_files[:100])
         dog_files_short_tq = tqdm(dog_files[:100])

         dogs_in_humanData = list(map(dog_detector, human_files_short_tq))
         dogs_in_dogData = list(map(dog_detector, dog_files_short_tq))

         print('Percentage of dogs detected in human_files(incorrect detection):', (sum(dogs_in_
         print('Percentage of dogs detected in dog_files(correct detection):', (sum(dogs_in_dogD

  0%|          | 0/100 [00:00<?, ?it/s]
100%|| 100/100 [00:03<00:00, 29.89it/s]
100%|| 100/100 [00:07<00:00, 25.03it/s]

Percentage of dogs detected in human_files(incorrect detection): 0.0
Percentage of dogs detected in dog_files(correct detection): 100.0
```

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

```
In [20]: ### (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!

1.1.9 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [21]: import os
         from torchvision import datasets

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

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

1.1.10 Image Normalization

```
In [22]: import torchvision.transforms as transforms
         normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                          std=[0.229, 0.224, 0.225])
```

1.1.11 Preprocessing

```
In [23]: preprocess_data = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                                         transforms.RandomHorizontalFlip(),
                                                         transforms.ToTensor(),
                                                         normalize]),
                            'valid': transforms.Compose([transforms.Resize(256),
                                                         transforms.CenterCrop(224),
                                                         transforms.ToTensor(),
                                                         normalize]),
                            'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                                         transforms.ToTensor(),
                                                         normalize])
                           }
```

1.1.12 Load Dataset

```
In [24]: 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['test'])

In [25]: batch_size = 20
         num_workers = 0
         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:

First I reduce all the images to 224x224, RandomiResizedCrop does a a random crop of the original image so that our model is able to learn complex variations in data. I have also flipped the data randomly and horizontally 5 times in order to add more variation to the training data.

1.1.13 (IMPLEMENTATION) Model Architecture

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

In [26]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [27]:
```
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

        # Conv Layers
        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)
        # maxpool
        self.pool = nn.MaxPool2d(2, 2)
        # fc layers
        self.fc4 = nn.Linear(7*7*128, 2048)
        self.fc5 = nn.Linear(2048, 512)
        self.fc6 = nn.Linear(512, 133)    #number of classes = 133
        # dropout
        self.dropout = nn.Dropout(0.25)    #dropout of 0.25
        self.batch_norm = nn.BatchNorm1d(512)

    def forward(self, x):
        ## Define forward behavior
        x = F.relu(self.conv1(x))
        x = self.pool(x)
        x = F.relu(self.conv2(x))
        x = self.pool(x)
        x = F.relu(self.conv3(x))
        x = self.pool(x)

        # flatten
        x = x.view(-1, 7*7*128)
        x = self.dropout(x)
        x = F.relu(self.fc4(x))
        x = self.dropout(x)
        x = F.relu(self.batch_norm(self.fc5(x)))
        x = self.dropout(x)
        x = self.fc6(x)
        return x
```

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

        # instantiate the CNN
        baseline_model = Net()

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

```
In [28]: baseline_model
```

```
Out[28]: 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)
           (fc4): Linear(in_features=6272, out_features=2048, bias=True)
           (fc5): Linear(in_features=2048, out_features=512, bias=True)
           (fc6): Linear(in_features=512, out_features=133, bias=True)
           (dropout): Dropout(p=0.25)
           (batch_norm): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_st
         )
```

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

Answer:

Model Architechture: Layer-1

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) -> size of the image will reduce by a factor of 2, as we have applied stride of (2,2) and padding of (1,1) for (3,3) kernel [(224,224) -> (112,112)] Activation: RELU (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) size of image will be reduced by factor of 2 [(112,112) -> (56,56)] Output shape for each image after his layer is expected to be (56,56,32)

Layer-2

(conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) -> size of the image will reduce by a factor of 2, as we have applied stride of (2,2) and padding of (1,1) for (3,3) kernel [(56,56) -> (28,28)] Activation: RELU (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) size of image will be reduced by factor of 2 [(28,28) -> (14,14)] Output shape for each image after his layer is expected to be (14,14,64)

Layer-3

(conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1)) -> size of the image remains the same here as we have applied stride of (1,1) and padding of (1,1) for (3,3) kernel [(14,14) -> (14,14)] Activation: RELU (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) size of image will be reduced by factor of 2 [(14,14) -> (7,7)] Output shape for each image after his layer is expected to be (7,7,128)

Downsized the image from (224,224) to (7,7) increasing the image depth from 3 to 128, we can now use this as features from images to build our breed classifier

Flattening

We flatten the image to get a 25088 sized vector or this can be imagined as 25088 hidden nodes layer with is further connected to fc4 I have then applied dropout of 0.25 which keeps deactivates 25% of neurons of this layer

Layer-4

(fc4): Linear(in_features=6272, out_features=2048, bias=True) -> We have extracted 6272 features from each image and now I have built a Fully connected layer with 512 hidden nodes. (batch_norm): BatchNorm1d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) Activation: RELU Applied dropout of 0.25 which keeps deactivates 25% of neurons of this layer

Layer-5

(fc5): Linear(in_features=2048, out_features=512, bias=True) Activation: RELU Applied dropout of 0.25 which keeps deactivates 25% of neurons of this layer

Layer-6

(fc5): Linear(in_features=512, out_features=133, bias=True) Fully-connected layer with 133 hidden nodes(number of classes) Thus finally extracted 133 sized vector, one for each data point This 133 sized vector is then used to predict classses

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_scratch`, and the optimizer as `optimizer_scratch` below.

```
In [29]: import torch.optim as optim

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

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

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 `'baseline_model.pt'`.

```
In [30]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
```

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

    # 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)

    # 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.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


        # train the model
        baseline_model = train(15, loaders_scratch, baseline_model, optimizer_scratch,
                               criterion_scratch, use_cuda, 'baseline_model.pt')
```

```
Epoch 1, Batch 1 loss: 4.825505
Epoch 1, Batch 101 loss: 5.204923
Epoch 1, Batch 201 loss: 5.042502
Epoch 1, Batch 301 loss: 4.979601
Epoch: 1         Training Loss: 4.964141         Validation Loss: 4.778291
Validation loss decreased (inf --> 4.778291).  Saving model ...
Epoch 2, Batch 1 loss: 4.702958
Epoch 2, Batch 101 loss: 4.824970
Epoch 2, Batch 201 loss: 4.818782
Epoch 2, Batch 301 loss: 4.813563
Epoch: 2         Training Loss: 4.813499         Validation Loss: 4.731102
Validation loss decreased (4.778291 --> 4.731102).  Saving model ...
Epoch 3, Batch 1 loss: 4.674215
Epoch 3, Batch 101 loss: 4.803570
Epoch 3, Batch 201 loss: 4.804308
Epoch 3, Batch 301 loss: 4.800226
Epoch: 3         Training Loss: 4.799490         Validation Loss: 4.713879
Validation loss decreased (4.731102 --> 4.713879).  Saving model ...
Epoch 4, Batch 1 loss: 4.709377
Epoch 4, Batch 101 loss: 4.769372
Epoch 4, Batch 201 loss: 4.774244
Epoch 4, Batch 301 loss: 4.770833
Epoch: 4         Training Loss: 4.768453         Validation Loss: 4.679176
Validation loss decreased (4.713879 --> 4.679176).  Saving model ...
Epoch 5, Batch 1 loss: 4.546038
Epoch 5, Batch 101 loss: 4.746377
Epoch 5, Batch 201 loss: 4.749585
```

```
Epoch 5, Batch 301 loss: 4.752195
Epoch: 5         Training Loss: 4.753931         Validation Loss: 4.712590
Epoch 6, Batch 1 loss: 4.887907
Epoch 6, Batch 101 loss: 4.747995
Epoch 6, Batch 201 loss: 4.729998
Epoch 6, Batch 301 loss: 4.733893
Epoch: 6         Training Loss: 4.731411         Validation Loss: 4.638180
Validation loss decreased (4.679176 --> 4.638180).  Saving model ...
Epoch 7, Batch 1 loss: 4.580367
Epoch 7, Batch 101 loss: 4.681383
Epoch 7, Batch 201 loss: 4.689792
Epoch 7, Batch 301 loss: 4.678575
Epoch: 7         Training Loss: 4.679544         Validation Loss: 4.645900
Epoch 8, Batch 1 loss: 4.237694
Epoch 8, Batch 101 loss: 4.642757
Epoch 8, Batch 201 loss: 4.645970
Epoch 8, Batch 301 loss: 4.656813
Epoch: 8         Training Loss: 4.654090         Validation Loss: 4.434084
Validation loss decreased (4.638180 --> 4.434084).  Saving model ...
Epoch 9, Batch 1 loss: 4.594656
Epoch 9, Batch 101 loss: 4.620029
Epoch 9, Batch 201 loss: 4.614000
Epoch 9, Batch 301 loss: 4.610202
Epoch: 9         Training Loss: 4.611492         Validation Loss: 4.427588
Validation loss decreased (4.434084 --> 4.427588).  Saving model ...
Epoch 10, Batch 1 loss: 4.675173
Epoch 10, Batch 101 loss: 4.616272
Epoch 10, Batch 201 loss: 4.603096
Epoch 10, Batch 301 loss: 4.601040
Epoch: 10         Training Loss: 4.600029         Validation Loss: 4.443689
Epoch 11, Batch 1 loss: 4.485263
Epoch 11, Batch 101 loss: 4.584201
Epoch 11, Batch 201 loss: 4.591425
Epoch 11, Batch 301 loss: 4.594946
Epoch: 11         Training Loss: 4.589386         Validation Loss: 4.332947
Validation loss decreased (4.427588 --> 4.332947).  Saving model ...
Epoch 12, Batch 1 loss: 4.750869
Epoch 12, Batch 101 loss: 4.564697
Epoch 12, Batch 201 loss: 4.570001
Epoch 12, Batch 301 loss: 4.562309
Epoch: 12         Training Loss: 4.556202         Validation Loss: 4.318972
Validation loss decreased (4.332947 --> 4.318972).  Saving model ...
Epoch 13, Batch 1 loss: 4.562172
Epoch 13, Batch 101 loss: 4.526420
Epoch 13, Batch 201 loss: 4.549870
Epoch 13, Batch 301 loss: 4.545997
Epoch: 13         Training Loss: 4.546474         Validation Loss: 4.292285
Validation loss decreased (4.318972 --> 4.292285).  Saving model ...
```

```
Epoch 14, Batch 1 loss: 4.404424
Epoch 14, Batch 101 loss: 4.548433
Epoch 14, Batch 201 loss: 4.555861
Epoch 14, Batch 301 loss: 4.553843
Epoch: 14         Training Loss: 4.553043         Validation Loss: 4.303793
Epoch 15, Batch 1 loss: 4.711486
Epoch 15, Batch 101 loss: 4.533415
Epoch 15, Batch 201 loss: 4.527799
Epoch 15, Batch 301 loss: 4.520708
Epoch: 15         Training Loss: 4.530296         Validation Loss: 4.330666
```

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 10%.

```
In [31]: ## load the model that got the best validation accuracy
         baseline_model.load_state_dict(torch.load('baseline_model.pt'))
```

```
In [32]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

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

```
Test Loss: 4.327450


Test Accuracy:  4% (40/836)
```

```
In [34]: def test(loaders, model, criterion, use_cuda):

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

             preds = []
             targets = []
             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]
                 preds.append(pred)
                 targets.append(target)
                 # 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))
             return preds, targets
```

```
In [36]: # call test function
         preds, gts = test(loaders_scratch, baseline_model, criterion_scratch, use_cuda)
```

```
Test Loss: 4.327450
```

```
Test Accuracy:  4% (40/836)
```

```
In [37]: predictions = []
         for x in preds:
             for y in np.array(x):
                 predictions.append(y[0])

         ground_truths = []
         for x in gts:
             for y in np.array(x):
                 ground_truths.append(y)

         from sklearn.metrics import confusion_matrix, precision_recall_fscore_support
         cm = confusion_matrix(ground_truths, predictions)
         precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
         recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
         print("Precision for breed classifier over test data:", precision)
         print("Recall for breed classifier over test data:", recall)
```

```
Precision for breed classifier over test data: 0.0147228496059
Recall for breed classifier over test data: 0.0360961928631
```

```
/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
```

With a test accuracy of 4%, we can tell that the model is not performing well on our test data.

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.

1.1.17 (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 [38]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

1.1.18 (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 [39]: 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 fc with custom f

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

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

```
Downloading: "https://download.pytorch.org/models/resnet101-5d3b4d8f.pth" to /root/.torch/models
100%|| 178728960/178728960 [00:03<00:00, 50229977.86it/s]
```

```
In [40]: model_transfer
```

```
Out[40]: ResNet(
           (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
           (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True
           (relu): ReLU(inplace)
           (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
           (layer1): Sequential(
             (0): Bottleneck(
               (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
               (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=
               (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
               (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               (relu): ReLU(inplace)
               (downsample): Sequential(
                 (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
                 (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
               )
             )
             (1): Bottleneck(
```

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

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

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

```
  (relu): ReLU(inplace)
)
(9): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(10): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(11): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(12): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(13): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(14): Bottleneck(
```

```
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(15): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(16): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(17): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(18): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stat
  (relu): ReLU(inplace)
)
(19): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
```

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

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

Here i chose Transfer learning to get one of the best avialable solutions to create a model for the dog breed classifer. For this approach I've chosen the ResNex-101 model. It is widely known that ResNets give outstanding performance in image classification. ResNet is build with "Residual Blocks" which are basically the skip connections between layers. Identity cnnection between the inital and final layers are created, as a result the risk of vanishing/exploding gradient problem is further reduced. So i think it will be performe better.

1.1.19 (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 [41]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.1.20 (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 [42]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf
```

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

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

        # 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))

        # 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.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 [43]:
```
# train the model
model_transfer =  train(20, loaders_transfer, model_transfer, optimizer_transfer, crite
```

```
Epoch 1, Batch 1 loss: 4.973514
Epoch 1, Batch 101 loss: 4.892207
Epoch 1, Batch 201 loss: 4.847093
Epoch 1, Batch 301 loss: 4.808314
Epoch: 1         Training Loss: 4.798481         Validation Loss: 4.604923
Validation loss decreased (inf --> 4.604923).  Saving model ...
Epoch 2, Batch 1 loss: 4.658899
Epoch 2, Batch 101 loss: 4.613069
Epoch 2, Batch 201 loss: 4.587004
Epoch 2, Batch 301 loss: 4.552963
Epoch: 2         Training Loss: 4.544950         Validation Loss: 4.315177
Validation loss decreased (4.604923 --> 4.315177).  Saving model ...
Epoch 3, Batch 1 loss: 4.505795
Epoch 3, Batch 101 loss: 4.385215
Epoch 3, Batch 201 loss: 4.364633
Epoch 3, Batch 301 loss: 4.328303
Epoch: 3         Training Loss: 4.315589         Validation Loss: 4.051013
Validation loss decreased (4.315177 --> 4.051013).  Saving model ...
Epoch 4, Batch 1 loss: 4.119211
Epoch 4, Batch 101 loss: 4.163280
```

```
Epoch 4, Batch 201 loss: 4.136557
Epoch 4, Batch 301 loss: 4.107597
Epoch: 4         Training Loss: 4.101393         Validation Loss: 3.799547
Validation loss decreased (4.051013 --> 3.799547).  Saving model ...
Epoch 5, Batch 1 loss: 3.853729
Epoch 5, Batch 101 loss: 3.981871
Epoch 5, Batch 201 loss: 3.945935
Epoch 5, Batch 301 loss: 3.914834
Epoch: 5         Training Loss: 3.907998         Validation Loss: 3.565405
Validation loss decreased (3.799547 --> 3.565405).  Saving model ...
Epoch 6, Batch 1 loss: 3.612921
Epoch 6, Batch 101 loss: 3.741554
Epoch 6, Batch 201 loss: 3.743999
Epoch 6, Batch 301 loss: 3.726072
Epoch: 6         Training Loss: 3.714974         Validation Loss: 3.362331
Validation loss decreased (3.565405 --> 3.362331).  Saving model ...
Epoch 7, Batch 1 loss: 3.798438
Epoch 7, Batch 101 loss: 3.575932
Epoch 7, Batch 201 loss: 3.552421
Epoch 7, Batch 301 loss: 3.532273
Epoch: 7         Training Loss: 3.529489         Validation Loss: 3.156446
Validation loss decreased (3.362331 --> 3.156446).  Saving model ...
Epoch 8, Batch 1 loss: 3.508535
Epoch 8, Batch 101 loss: 3.410669
Epoch 8, Batch 201 loss: 3.383527
Epoch 8, Batch 301 loss: 3.376926
Epoch: 8         Training Loss: 3.366331         Validation Loss: 2.929338
Validation loss decreased (3.156446 --> 2.929338).  Saving model ...
Epoch 9, Batch 1 loss: 3.225792
Epoch 9, Batch 101 loss: 3.251453
Epoch 9, Batch 201 loss: 3.230160
Epoch 9, Batch 301 loss: 3.215488
Epoch: 9         Training Loss: 3.207018         Validation Loss: 2.774928
Validation loss decreased (2.929338 --> 2.774928).  Saving model ...
Epoch 10, Batch 1 loss: 3.044631
Epoch 10, Batch 101 loss: 3.117269
Epoch 10, Batch 201 loss: 3.089060
Epoch 10, Batch 301 loss: 3.067130
Epoch: 10         Training Loss: 3.065283         Validation Loss: 2.573985
Validation loss decreased (2.774928 --> 2.573985).  Saving model ...
Epoch 11, Batch 1 loss: 2.923219
Epoch 11, Batch 101 loss: 3.006604
Epoch 11, Batch 201 loss: 2.977194
Epoch 11, Batch 301 loss: 2.947278
Epoch: 11         Training Loss: 2.938867         Validation Loss: 2.448886
Validation loss decreased (2.573985 --> 2.448886).  Saving model ...
Epoch 12, Batch 1 loss: 3.086581
Epoch 12, Batch 101 loss: 2.879813
```

```
Epoch 12, Batch 201 loss: 2.849325
Epoch 12, Batch 301 loss: 2.829966
Epoch: 12         Training Loss: 2.817853         Validation Loss: 2.267022
Validation loss decreased (2.448886 --> 2.267022).  Saving model ...
Epoch 13, Batch 1 loss: 2.418918
Epoch 13, Batch 101 loss: 2.717321
Epoch 13, Batch 201 loss: 2.710906
Epoch 13, Batch 301 loss: 2.698286
Epoch: 13         Training Loss: 2.695180         Validation Loss: 2.176843
Validation loss decreased (2.267022 --> 2.176843).  Saving model ...
Epoch 14, Batch 1 loss: 2.450116
Epoch 14, Batch 101 loss: 2.574448
Epoch 14, Batch 201 loss: 2.592059
Epoch 14, Batch 301 loss: 2.580299
Epoch: 14         Training Loss: 2.583135         Validation Loss: 2.097639
Validation loss decreased (2.176843 --> 2.097639).  Saving model ...
Epoch 15, Batch 1 loss: 2.090991
Epoch 15, Batch 101 loss: 2.538318
Epoch 15, Batch 201 loss: 2.511801
Epoch 15, Batch 301 loss: 2.496628
Epoch: 15         Training Loss: 2.494627         Validation Loss: 1.950028
Validation loss decreased (2.097639 --> 1.950028).  Saving model ...
Epoch 16, Batch 1 loss: 2.211379
Epoch 16, Batch 101 loss: 2.432922
Epoch 16, Batch 201 loss: 2.423085
Epoch 16, Batch 301 loss: 2.405747
Epoch: 16         Training Loss: 2.399301         Validation Loss: 1.870739
Validation loss decreased (1.950028 --> 1.870739).  Saving model ...
Epoch 17, Batch 1 loss: 2.611705
Epoch 17, Batch 101 loss: 2.347693
Epoch 17, Batch 201 loss: 2.330876
Epoch 17, Batch 301 loss: 2.323799
Epoch: 17         Training Loss: 2.322139         Validation Loss: 1.765440
Validation loss decreased (1.870739 --> 1.765440).  Saving model ...
Epoch 18, Batch 1 loss: 2.600535
Epoch 18, Batch 101 loss: 2.257538
Epoch 18, Batch 201 loss: 2.243726
Epoch 18, Batch 301 loss: 2.235616
Epoch: 18         Training Loss: 2.234337         Validation Loss: 1.713316
Validation loss decreased (1.765440 --> 1.713316).  Saving model ...
Epoch 19, Batch 1 loss: 2.229483
Epoch 19, Batch 101 loss: 2.185895
Epoch 19, Batch 201 loss: 2.171265
Epoch 19, Batch 301 loss: 2.172800
Epoch: 19         Training Loss: 2.178100         Validation Loss: 1.608305
Validation loss decreased (1.713316 --> 1.608305).  Saving model ...
Epoch 20, Batch 1 loss: 2.372375
Epoch 20, Batch 101 loss: 2.125071
```

```
Epoch 20, Batch 201 loss: 2.125735
Epoch 20, Batch 301 loss: 2.110973
Epoch: 20         Training Loss: 2.109468         Validation Loss: 1.560796
Validation loss decreased (1.608305 --> 1.560796).  Saving model ...
```

```
In [44]: model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.21 (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 [47]: def test(loaders, model, criterion, use_cuda):

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

             preds = []
             targets = []

             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]
                 preds.append(pred)
                 targets.append(target)
                 # 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))
             return preds,targets

         preds, gts = test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 1.603196


Test Accuracy: 75% (628/836)
```

The accuracy is over 60%. A high jump in predicition accuracy. Next, we'll check precision and recall.

```
In [48]: predictions = []
         for x in preds:
             for y in np.array(x):
                 predictions.append(y[0])

         ground_truths = []
         for x in gts:
             for y in np.array(x):
                 ground_truths.append(y)
```

```
In [49]: from sklearn.metrics import confusion_matrix, precision_recall_fscore_support
         cm = confusion_matrix(ground_truths, predictions)
         precision = np.mean(precision_recall_fscore_support(ground_truths, predictions)[0])
         recall = np.mean(precision_recall_fscore_support(ground_truths, predictions)[1])
         print("Precision for breed classifier over test data:", precision)
         print("Recall for breed classifier over test data:", recall)
```

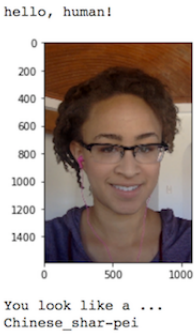
```
Precision for breed classifier over test data: 0.687219986756
Recall for breed classifier over test data: 0.691618928273


/opt/conda/lib/python3.6/site-packages/sklearn/metrics/classification.py:1135: UndefinedMetricWa
  'precision', 'predicted', average, warn_for)
```

The precision and recall have also improved a lot. We can increase the accuracy, recall and precision by doing more hyperparameter tuning. Also, training the model for more epochs will help to boost these numbers!

1.1.22 (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 [51]: ### 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
```



Sample Human Output

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

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!

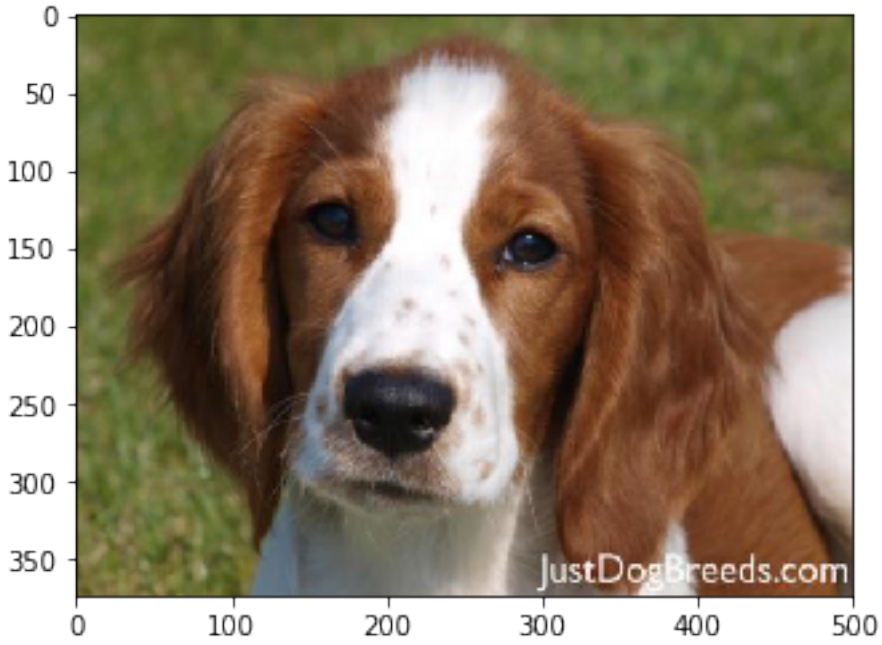
1.1.23 (IMPLEMENTATION) Write your Algorithm

```
In [52]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = Image.open(img_path)

             if dog_detector(img_path):
                 print('Dog Detected!')
                 plt.imshow(img)
                 plt.show()
                 print(f'This is a "{predict_breed_transfer(model_transfer, class_names, img_pat
             elif face_detector(img_path):
```

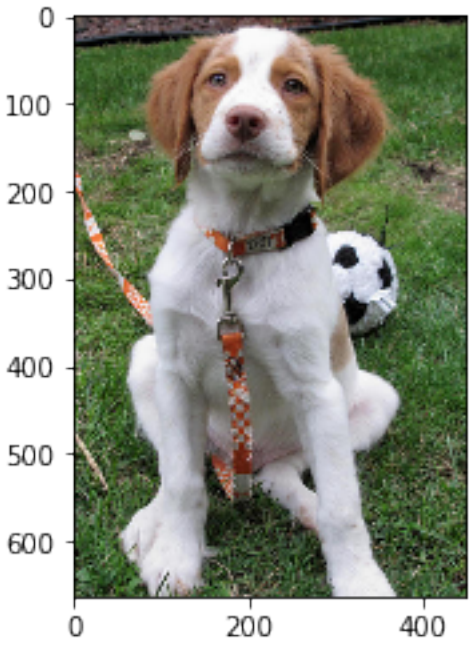
```
            print('Human Detected!')
            plt.imshow(img)
            plt.show()
            print(f'Interesting, this human looks like "{predict_breed_transfer(model_trans
        else:
            print('Ooops! Nothing to detect!')
            plt.imshow(img)
            plt.show()
```

In [53]:
```
for img_file in os.listdir('./images'):
    img_path = os.path.join('./images', img_file)
    run_app(img_path)
    print('')
```

```
Dog Detected!
```



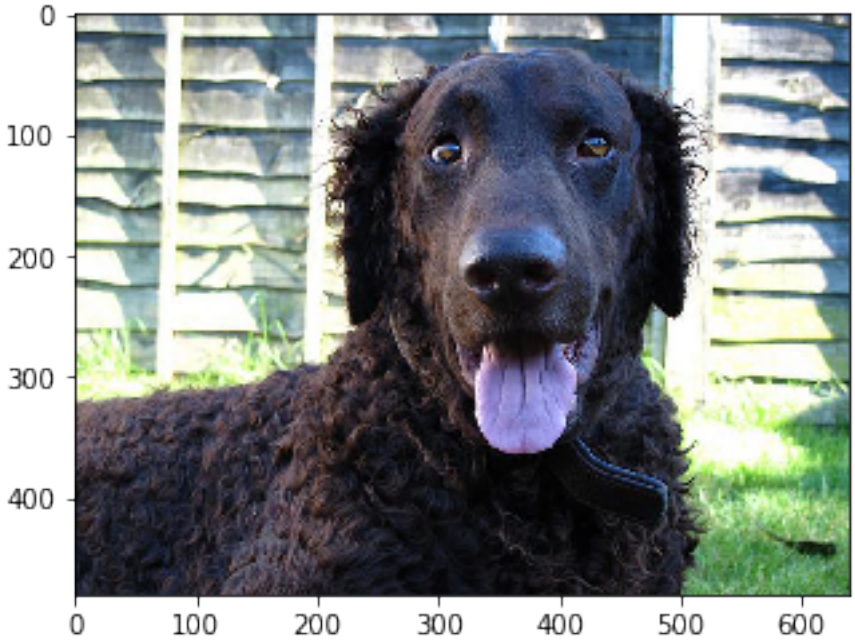

```
This is a "Welsh springer spaniel" kind of dog!

Dog Detected!
```




```
This is a "Brittany" kind of dog!

Dog Detected!
```

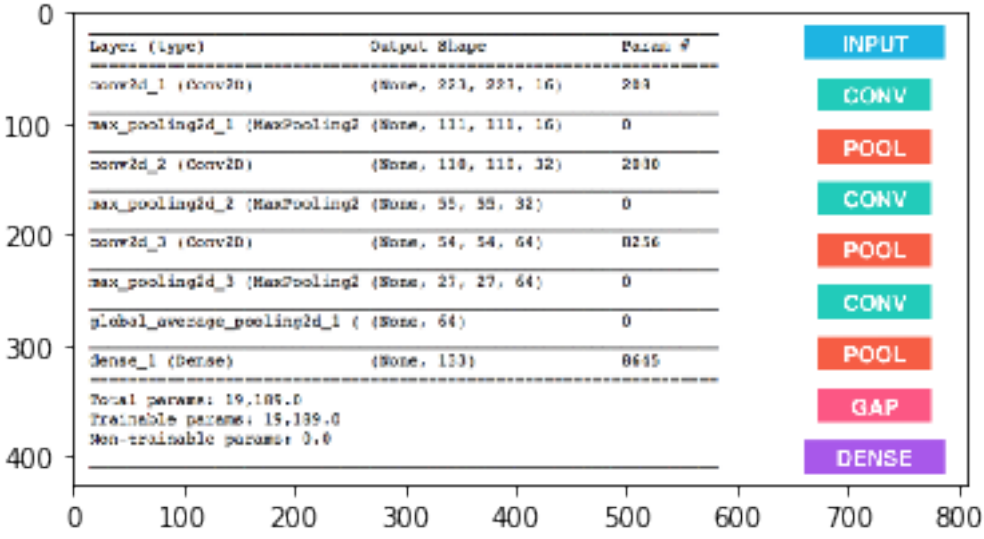
This is a "Flat-coated retriever" kind of dog!

Dog Detected!

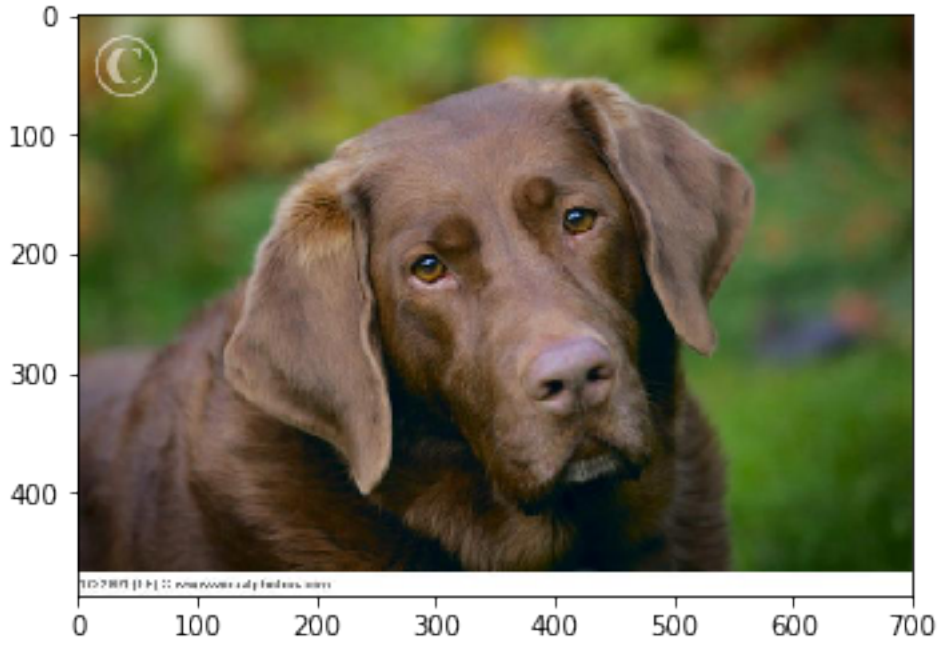


This is a "Curly-coated retriever" kind of dog!

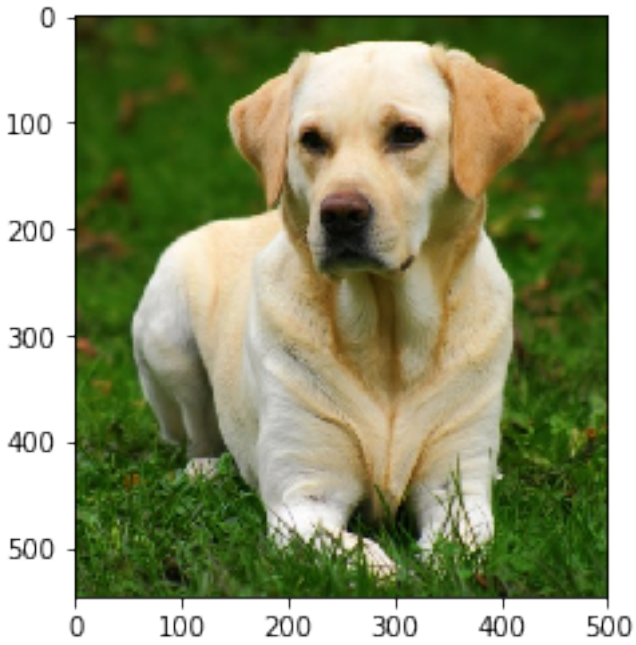
Ooops! Nothing to detect!



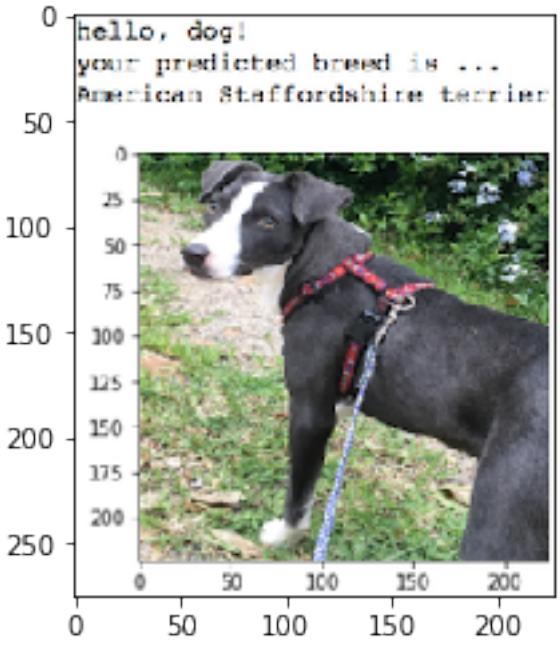
```
Dog Detected!
```



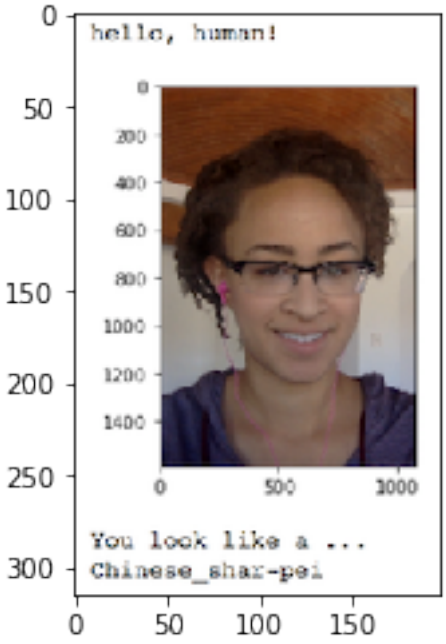

```
This is a "Chesapeake bay retriever" kind of dog!

Dog Detected!
```




```
This is a "Labrador retriever" kind of dog!

Dog Detected!
```



```
This is a "Border collie" kind of dog!

Human Detected!
```




```
Interesting, this human looks like "Bullmastiff"!

Dog Detected!
```


```
This is a "Curly-coated retriever" kind of 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?

1.1.24 (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)

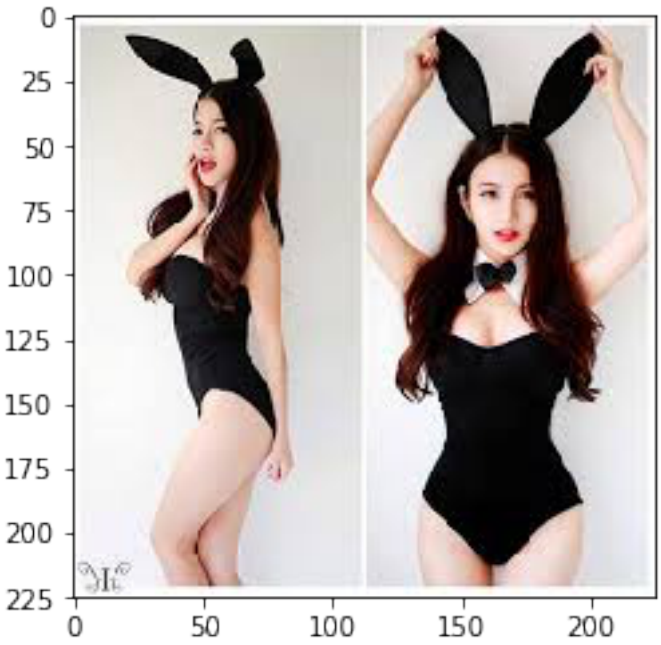
Three improvements can be:

- The algorithm could be made available in a web application to be more accessible to people, by involving more people in training the algorithm, we could develop more ways to fine-tune or create new algorithm models.
- Training on more data can help in this case, also augmentation of the data may help like criping images to correct areas may help.
- Hyperparameter tuning always helps.

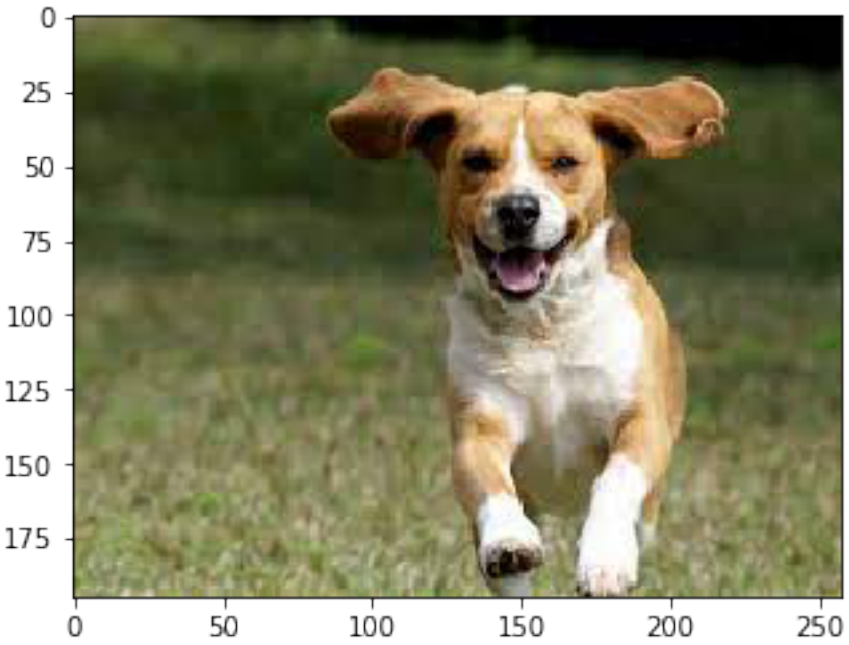
- Trying out better pretrained models.

In [54]:
```
## 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(glob('uploaded_images/*')):
    run_app(file)
    print('-' * 100)
```

```
Human Detected!
```

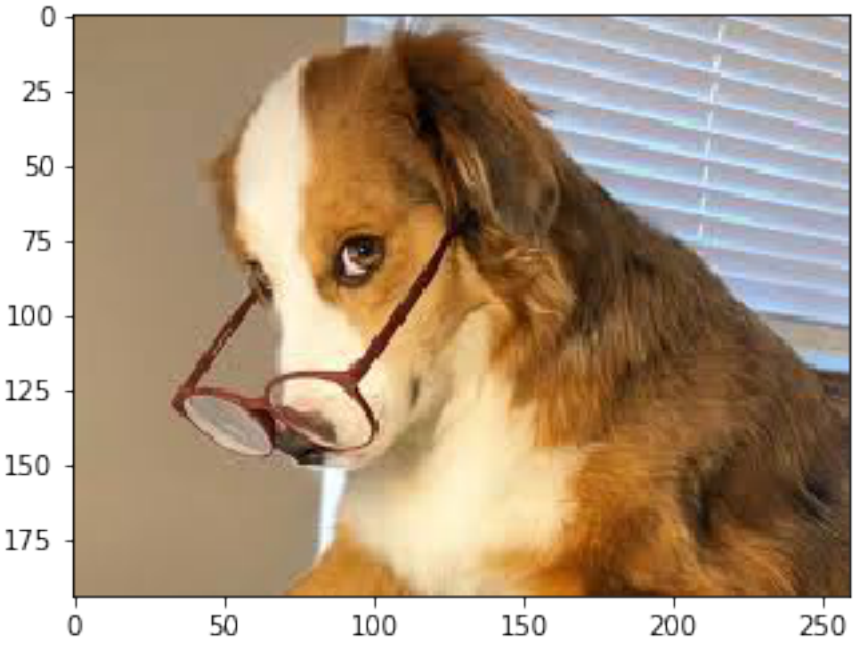



```
Interesting, this human looks like "Japanese chin"!
----------------------------------------------------------------------------------------------------
Dog Detected!
```

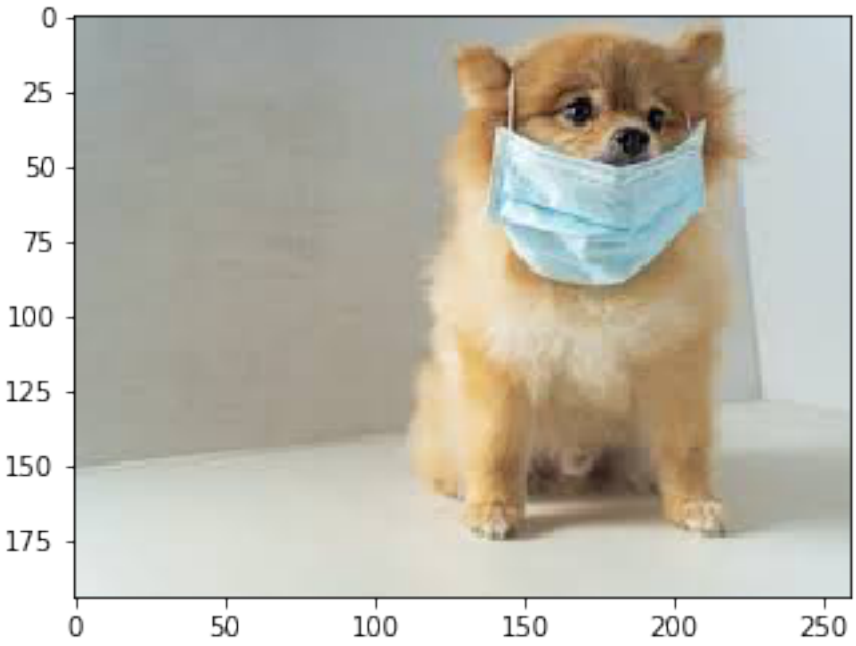



```
This is a "Beagle" kind of dog!
```

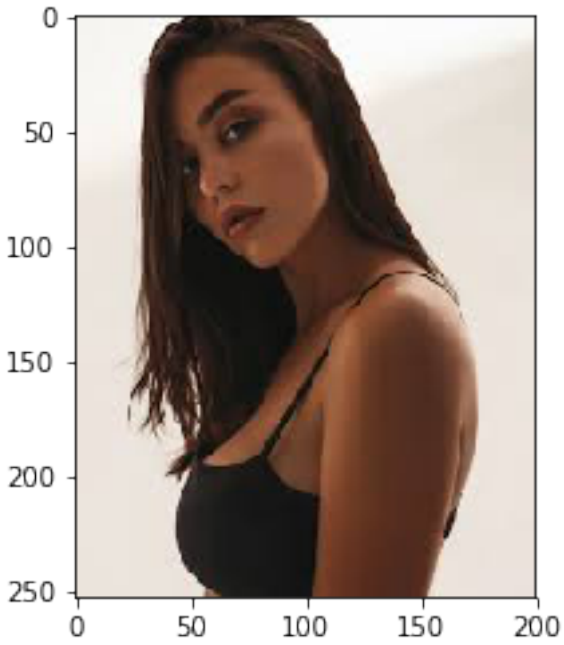
```
Ooops! Nothing to detect!
```



```
--------------------------------------------------------------------------------------------------------------
Dog Detected!
```




```
This is a "Pomeranian" kind of dog!
--------------------------------------------------------------------------------------------------------------
Ooops! Nothing to detect!
```

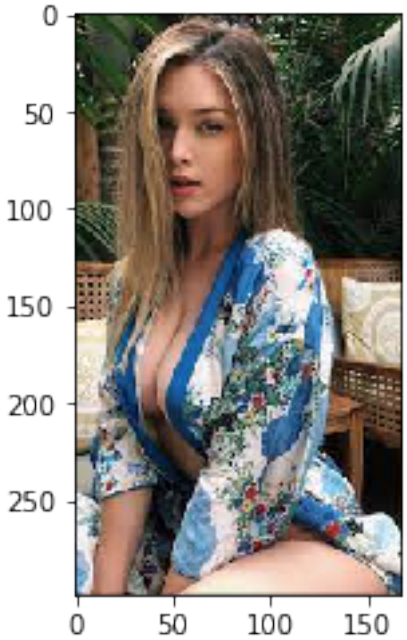


```
Dog Detected!
```


```
This is a "Golden retriever" kind of dog!
```

```
Ooops! Nothing to detect!
```

```
Human Detected!
```



```
Interesting, this human looks like "Great dane"!
--------------------------------------------------------------------------------------------------------------
Dog Detected!
```


```
This is a "Australian shepherd" kind of dog!
--------------------------------------------------------------------------------------------------------------
```

In [ ]: