dog_app

May 20, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
## Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets:
Note: if you are using the Udacity workspace, you *DO NOT* need to re-download these - they can be found in the `/data` folder as noted in the cell below.

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

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

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

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

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

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

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

## Step 1: Detect Humans

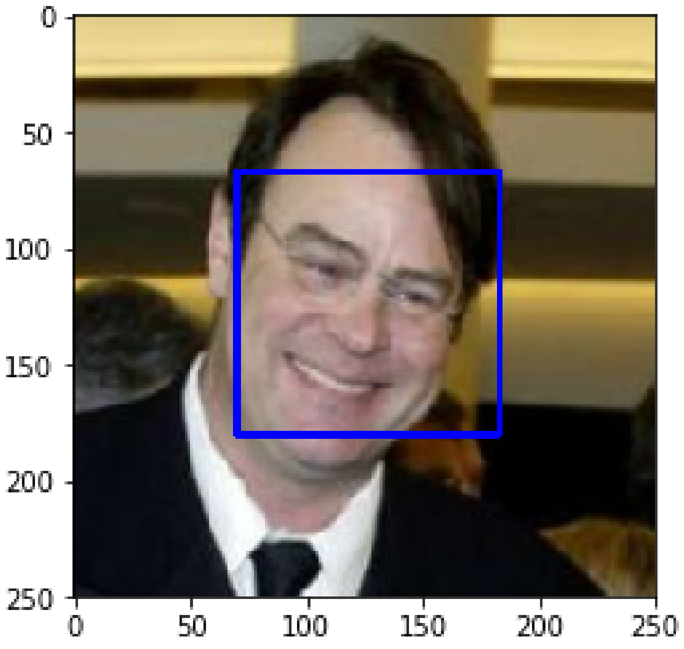
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
In [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: Percentage of human faces detected as human is 98 % & Percentage of dog faces detected as human is 17 %

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.
        def face_detection_test(files):
            detection_cnt = 0;
            total_cnt = len(files)
            for file in files:
                detection_cnt += face_detector(file)
            return detection_cnt, total_cnt
```

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

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
        print("detect face in human_files: {} / {}".format(face_detection_test(human_files_short
        print("detect face in dog_files: {} / {}".format(face_detection_test(dog_files_short)[0]
```

```
detect face in human_files: 98 / 100
detect face in dog_files: 17 / 100
```

```
In [6]: detected_human_human_faces = 0
        for img_path in human_files_short:
            if face_detector(img_path):
                detected_human_human_faces += 1
        print('Percentage of human faces detected as human is ', detected_human_human_faces, '%'

        detected_human_dog_faces = 0
        for img_path in dog_files_short:
            if face_detector(img_path):
                detected_human_dog_faces += 1
        print('Percentage of dog faces detected as human is ', detected_human_dog_faces, '%')
```

```
Percentage of human faces detected as human is  98 %
Percentage of dog faces detected as human is  17 %
```

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [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()
```

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:10<00:00, 55128321.74it/s]
```

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

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

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

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

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

Use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

```
In [8]: from PIL import Image
        import torchvision.transforms as transforms

        def load_image(img_path):
            image = Image.open(img_path).convert('RGB')
            # resize to (244, 244) because VGG16 accept this shape
            in_transform = transforms.Compose([
                                transforms.Resize(size=(244, 244)),
                                transforms.ToTensor()]) # normalizaiton parameters from pytorch

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

```
In [9]: 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 = load_image(img_path)
            if use_cuda:
                img = img.cuda()
            ret = VGG16(img)
            return torch.max(ret,1)[1].item() # predicted class index
```

```
In [10]: # predicted class index
         VGG16_predict(dog_files_short[0])
```

```
Out[10]: 243
```

```
In [11]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             idx = VGG16_predict(img_path)
             return idx >= 151 and idx <= 268 # true/false
```

```
In [12]: print(dog_detector(dog_files_short[0]))
         print(dog_detector(human_files_short[0]))
```

```
True
False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.
- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer: Percentage of the images in `human_files_short` have a detected dog is 0% What percentage of the images in `dog_files_short` have a detected dog is 96%

```
In [13]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detector_test(files):
             detection_cnt = 0;
             total_cnt = len(files)
             for file in files:
                 detection_cnt += dog_detector(file)
             return detection_cnt, total_cnt
```

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 [14]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
         print("detect a dog in human_files: {} / {}".format(dog_detector_test(human_files_short
         print("detect a dog in dog_files: {} / {}".format(dog_detector_test(dog_files_short)[0]
```

```
detect a dog in human_files: 0 / 100
detect a dog in dog_files: 96 / 100
```

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [15]: import os
         import torchvision
         from torchvision import datasets
         import torchvision.transforms as transforms
         import torch
         import numpy as np
         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         ### TODO: Write data loaders for training, validation, and test sets
         # load filenames for human and dog images
         #data_dir = glob.glob("/data/dog_images/")
         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/')
         print(train_dir)
         ## Specify appropriate transforms, and batch_sizes
         batch_size = 20
         num_workers=0

         # prepare data loaders
         loaders = {};
         loaders['train'] = torch.utils.data.DataLoader(train_dir, batch_size=batch_size, num_wo
         loaders['valid'] = torch.utils.data.DataLoader(valid_dir, batch_size=batch_size, num_wo
         loaders['test'] = torch.utils.data.DataLoader(test_dir, batch_size=batch_size, num_work
```

```
/data/dog_images/train/
```

```
In [16]: #**Standard Normalization**
         standard_normalization = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                       std=[0.229, 0.224, 0.225])
```

```
In [17]: data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                              transforms.RandomHorizontalFlip(),
                                              transforms.ToTensor(),
                                              standard_normalization]),
                              'val': transforms.Compose([transforms.Resize(256),
                                                  transforms.CenterCrop(224),
                                                  transforms.ToTensor(),
                                                  standard_normalization]),
                              'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                                  transforms.ToTensor(),
                                                  standard_normalization])
                             }
```

```
In [18]: train_data = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
         valid_data = datasets.ImageFolder(valid_dir, transform=data_transforms['val'])
         test_data = datasets.ImageFolder(test_dir, transform=data_transforms['test'])
```

```
In [19]: 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: I've applied RandomResizedCrop & RandomHorizontalFlip to just train_data. This will do both image augmentations and resizing jobs. Image augmentation will give randomness to the dataset so, it prevents overfitting and I can expect better performance of model when it's predicting toward test_data. On the other hand, I've done Resize of (256) and then, center crop to make 224 X 224. Since valid_data will be used for validation check, I will not do image augmentations. For the test_data, I've applied only image resizing.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [20]: import torch.nn as nn
         import torch.nn.functional as F
```

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)   # input: 3x224x224, output: 16x22
        self.pool1 = nn.MaxPool2d(2, 2)               # input: 16x224x224, output: 16x1

        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)  # input: 16x112x112, output: 32x1
        self.pool2 = nn.MaxPool2d(2, 2)               # input: 32x112x112, output: 32x5

        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)  # input: 32x56x56, output: 64x56x
        self.pool3 = nn.MaxPool2d(2, 2)               # input: 64x56x56, output: 64x28x

        self.conv4 = nn.Conv2d(64, 128, 3, padding=1) # input: 64x28x28, output: 128x28
        self.pool4 = nn.MaxPool2d(2, 2)               # input: 128x28x28, output: 128x1

        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)# input: 128x14x14, output: 256x1
        self.pool5 = nn.MaxPool2d(2, 2)               # input: 256x14x14, output: 256x7

        self.fc1 = nn.Linear(256 * 7 * 7, 500)        # linear layer (256 * 7 * 7 -> 50

        self.fc2 = nn.Linear(500, 133)                # linear layer (500 -> 133)
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        x = self.pool1(F.relu(self.conv1(x)))
        x = self.pool2(F.relu(self.conv2(x)))
        x = self.pool3(F.relu(self.conv3(x)))
        x = self.pool4(F.relu(self.conv4(x)))
        x = self.pool5(F.relu(self.conv5(x)))
        x = self.dropout(x)
        x = x.view(-1, 256 * 7 * 7)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.fc2(x)

        return x

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

# instantiate the CNN
model_scratch = Net()
print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=12544, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

        total_dog_classes = 133 # total classes of dog

        # define the CNN architecture
        class Net(nn.Module):
            ### TODO: choose an architecture, and complete the class
            def __init__(self):
                super(Net, self).__init__()
                ## Define layers of a CNN
                self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
                self.norm2d1 = nn.BatchNorm2d(32)
                self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
                self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

                size_linear_layer = 500

                # linear layer (128 * 28 * 28 -> 500)
                self.fc1 = nn.Linear(128 * 28 * 28, size_linear_layer)
                self.fc2 = nn.Linear(size_linear_layer, total_dog_classes)

            def forward(self, x):
                x = self.pool(F.relu(self.norm2d1(self.conv1(x))))
                x = self.pool(F.relu(self.conv2(x)))
                x = self.pool(F.relu(self.conv3(x)))
                #print(x.shape)

                # flatten image input
                x = x.view(-1, 128 * 28 * 28)
            x = F.relu(self.fc1(x))
            x = self.fc2(x)
            return x
    #-#-# You so NOT have to modify the code below this line. #-#-#

    # instantiate the CNN
    model_scratch = Net()
    print(model_scratch)

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

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

Answer: Since I was getting the Test Accuracy as 1%, so i modified the architecture again.

I took the following steps : 1. I started with a light weight VGG-like architecture called Net() with alternate conv, relu, pool layers with couple of fully connected (FC) layers towards the end. I also added small dropouts in the FC layers for regularization. 2. Set the loss function to cross entropy loss and the optimization function to Adam optimizer 3. Trained the network for 100 epochs at learning rate of 0.001. Saved the model with lowest validation loss. 4. Evaluated the performance on the test data & got the accuracy=16%

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [21]: import torch.optim as optim

         ### TODO: select loss function
         ### TODO: select optimizer

         criterion_scratch = nn.CrossEntropyLoss()

         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_scratch.pt'`.

```
In [22]: 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:
            #cuda0 = torch.device('cuda:0')  # CUDA GPU 0
            #data = data.to(cuda0)
            #target = target.to(cuda0)
            data, target = data.cuda(), target.cuda()
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the batch loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model paramet
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()

        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

        #if batch_idx % 100 == 0:
        #    print('Epoch %d, Batch %d loss: %.6f' % (epoch, batch_idx + 1, train_l

    ######################
    # 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 ...'
                      .format(valid_loss_min, valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model

    # train the model
    model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

    # load the model that got the best validation accuracy
    model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
Epoch: 1         Training Loss: 4.861279         Validation Loss: 4.726482
Validation loss decreased (inf --> 4.726482).  Saving model ...
Epoch: 2         Training Loss: 4.676719         Validation Loss: 4.477589
Validation loss decreased (4.726482 --> 4.477589).  Saving model ...
Epoch: 3         Training Loss: 4.517503         Validation Loss: 4.370293
Validation loss decreased (4.477589 --> 4.370293).  Saving model ...
Epoch: 4         Training Loss: 4.415063         Validation Loss: 4.190855
Validation loss decreased (4.370293 --> 4.190855).  Saving model ...
Epoch: 5         Training Loss: 4.304856         Validation Loss: 4.120739
Validation loss decreased (4.190855 --> 4.120739).  Saving model ...
Epoch: 6         Training Loss: 4.241769         Validation Loss: 4.029843
Validation loss decreased (4.120739 --> 4.029843).  Saving model ...
Epoch: 7         Training Loss: 4.159904         Validation Loss: 4.009131
Validation loss decreased (4.029843 --> 4.009131).  Saving model ...
Epoch: 8         Training Loss: 4.113400         Validation Loss: 3.902313
Validation loss decreased (4.009131 --> 3.902313).  Saving model ...
Epoch: 9         Training Loss: 4.087402         Validation Loss: 3.906456
Epoch: 10         Training Loss: 3.991817         Validation Loss: 3.824522
Validation loss decreased (3.902313 --> 3.824522).  Saving model ...
Epoch: 11         Training Loss: 3.953603         Validation Loss: 3.821079
Validation loss decreased (3.824522 --> 3.821079).  Saving model ...
Epoch: 12         Training Loss: 3.905518         Validation Loss: 3.775762
Validation loss decreased (3.821079 --> 3.775762).  Saving model ...
```

```
Epoch: 13         Training Loss: 3.895503        Validation Loss: 3.683364
Validation loss decreased (3.775762 --> 3.683364).  Saving model ...
Epoch: 14         Training Loss: 3.820930        Validation Loss: 3.656262
Validation loss decreased (3.683364 --> 3.656262).  Saving model ...
Epoch: 15         Training Loss: 3.774843        Validation Loss: 3.586143
Validation loss decreased (3.656262 --> 3.586143).  Saving model ...
Epoch: 16         Training Loss: 3.726455        Validation Loss: 3.599086
Epoch: 17         Training Loss: 3.695903        Validation Loss: 3.553918
Validation loss decreased (3.586143 --> 3.553918).  Saving model ...
Epoch: 18         Training Loss: 3.678231        Validation Loss: 3.531502
Validation loss decreased (3.553918 --> 3.531502).  Saving model ...
Epoch: 19         Training Loss: 3.628726        Validation Loss: 3.502654
Validation loss decreased (3.531502 --> 3.502654).  Saving model ...
Epoch: 20         Training Loss: 3.589680        Validation Loss: 3.480582
Validation loss decreased (3.502654 --> 3.480582).  Saving model ...
```

1.1.11 (IMPLEMENTATION) Test the Model

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

```
In [23]: 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, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.597610


Test Accuracy: 16% (140/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

```
In [24]: ## TODO: Specify data loaders
         loaders_transfer = loaders_scratch.copy()
```

1.1.13 (IMPLEMENTATION) Model Architecture

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

```
In [25]: import torchvision.models as models
         import torch.nn as nn

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

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

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

         fc_parameters = model_transfer.fc.parameters()

         for param in fc_parameters:
             param.requires_grad = True
```

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:01<00:00, 85643084.02it/s]
```

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: ResNet is chosen as known as excellent performance for image classification.

final fully-connected layer is add with fully-connected layer with output of 133 (total calsses of dog).

1.1.14 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

```
In [26]: criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

```
In [27]: # train the model
         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

            #if batch_idx % 100 == 0:
            #    print('Epoch %d, Batch %d loss: %.6f' % (epoch, batch_idx + 1, train_l

        ######################
        # 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 ...'
                  .format(valid_loss_min, valid_loss))
            valid_loss_min = valid_loss

    # return trained model
    return model

n_epochs = 20
```

```
    model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1 	Training Loss: 4.817084 	Validation Loss: 4.640794
Validation loss decreased (inf --> 4.640794).  Saving model ...
Epoch: 2 	Training Loss: 4.600976 	Validation Loss: 4.393160
Validation loss decreased (4.640794 --> 4.393160).  Saving model ...
Epoch: 3 	Training Loss: 4.404707 	Validation Loss: 4.148517
Validation loss decreased (4.393160 --> 4.148517).  Saving model ...
Epoch: 4 	Training Loss: 4.218315 	Validation Loss: 3.924692
Validation loss decreased (4.148517 --> 3.924692).  Saving model ...
Epoch: 5 	Training Loss: 4.041693 	Validation Loss: 3.712546
Validation loss decreased (3.924692 --> 3.712546).  Saving model ...
Epoch: 6 	Training Loss: 3.872077 	Validation Loss: 3.498503
Validation loss decreased (3.712546 --> 3.498503).  Saving model ...
Epoch: 7 	Training Loss: 3.714423 	Validation Loss: 3.299213
Validation loss decreased (3.498503 --> 3.299213).  Saving model ...
Epoch: 8 	Training Loss: 3.560394 	Validation Loss: 3.115517
Validation loss decreased (3.299213 --> 3.115517).  Saving model ...
Epoch: 9 	Training Loss: 3.420659 	Validation Loss: 2.971718
Validation loss decreased (3.115517 --> 2.971718).  Saving model ...
Epoch: 10 	Training Loss: 3.290693 	Validation Loss: 2.775751
Validation loss decreased (2.971718 --> 2.775751).  Saving model ...
Epoch: 11 	Training Loss: 3.163588 	Validation Loss: 2.649623
Validation loss decreased (2.775751 --> 2.649623).  Saving model ...
Epoch: 12 	Training Loss: 3.046010 	Validation Loss: 2.517282
Validation loss decreased (2.649623 --> 2.517282).  Saving model ...
Epoch: 13 	Training Loss: 2.943332 	Validation Loss: 2.410411
Validation loss decreased (2.517282 --> 2.410411).  Saving model ...
Epoch: 14 	Training Loss: 2.837499 	Validation Loss: 2.273657
Validation loss decreased (2.410411 --> 2.273657).  Saving model ...
Epoch: 15 	Training Loss: 2.727579 	Validation Loss: 2.174567
Validation loss decreased (2.273657 --> 2.174567).  Saving model ...
Epoch: 16 	Training Loss: 2.652123 	Validation Loss: 2.069108
Validation loss decreased (2.174567 --> 2.069108).  Saving model ...
Epoch: 17 	Training Loss: 2.574926 	Validation Loss: 1.949706
Validation loss decreased (2.069108 --> 1.949706).  Saving model ...
Epoch: 18 	Training Loss: 2.495912 	Validation Loss: 1.894914
Validation loss decreased (1.949706 --> 1.894914).  Saving model ...
Epoch: 19 	Training Loss: 2.432360 	Validation Loss: 1.808928
Validation loss decreased (1.894914 --> 1.808928).  Saving model ...
Epoch: 20 	Training Loss: 2.359845 	Validation Loss: 1.753901
Validation loss decreased (1.808928 --> 1.753901).  Saving model ...
```

1.1.16 (IMPLEMENTATION) Test the Model

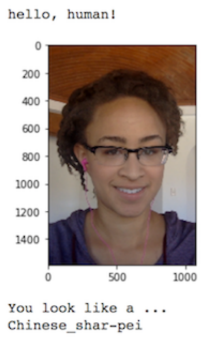
Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 60%.

```
In [28]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)

Test Loss: 1.822286


Test Accuracy: 72% (608/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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

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

         data_transfer = loaders_transfer.copy()
         # 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 data_transfer['train'].dataset.cl
         def load_input_image(img_path):
             image = Image.open(img_path).convert('RGB')
             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                              transforms.ToTensor(),
                                              standard_normalization])

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

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

In [70]: for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             predition = predict_breed_transfer(model_transfer, class_names, img_path)
             print("image_file_name: {0}, \t predition breed: {1}".format(img_path, predition))
```



```
hello, human!
```

```
You look like a ...
Chinese_shar-pei
```

Sample Human Output

```
image_file_name: ./images/Welsh_springer_spaniel_08203.jpg,          predition breed: Welsh spri
image_file_name: ./images/sample_human_output.png,          predition breed: Affenpinscher
image_file_name: ./images/Labrador_retriever_06457.jpg,          predition breed: Golden retriev
image_file_name: ./images/Curly-coated_retriever_03896.jpg,          predition breed: Curly-coat
image_file_name: ./images/sample_cnn.png,          predition breed: Affenpinscher
image_file_name: ./images/Brittany_02625.jpg,          predition breed: Brittany
image_file_name: ./images/Labrador_retriever_06449.jpg,          predition breed: Flat-coated re
image_file_name: ./images/American_water_spaniel_00648.jpg,          predition breed: Irish wate
image_file_name: ./images/sample_dog_output.png,          predition breed: Greyhound
image_file_name: ./images/Labrador_retriever_06455.jpg,          predition breed: Chesapeake bay
```

## Step 5: Write your Algorithm
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

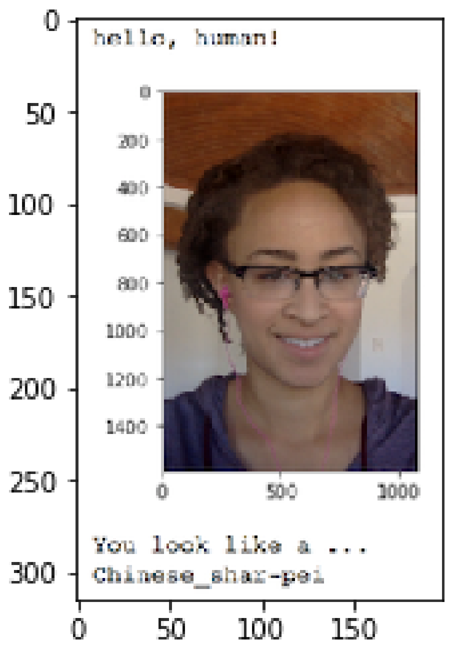
1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [71]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither
             img = Image.open(img_path)
             plt.imshow(img)
             plt.show()
             if dog_detector(img_path) is True:
```

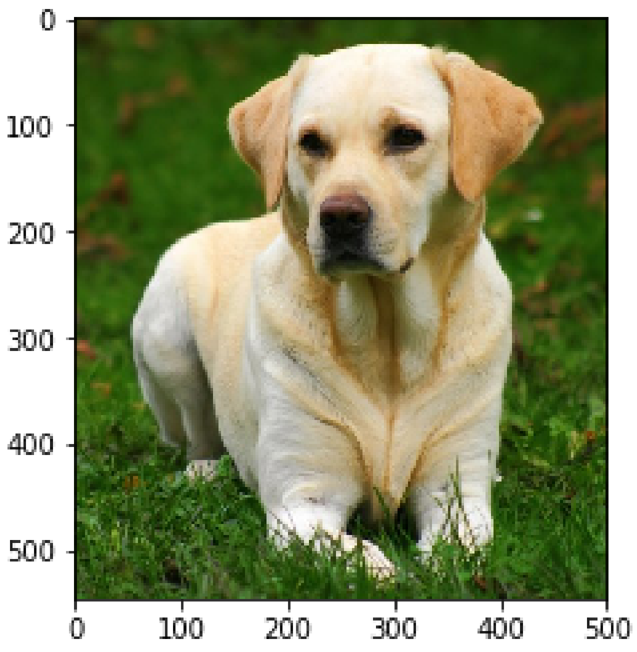
```
            prediction = predict_breed_transfer(model_transfer, class_names, img_path)
            print("Dogs Detected!\nIt looks like a {0}".format(prediction))
        elif face_detector(img_path) > 0:
            prediction = predict_breed_transfer(model_transfer, class_names, img_path)
    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

```
In [72]: for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             run_app(img_path)
```

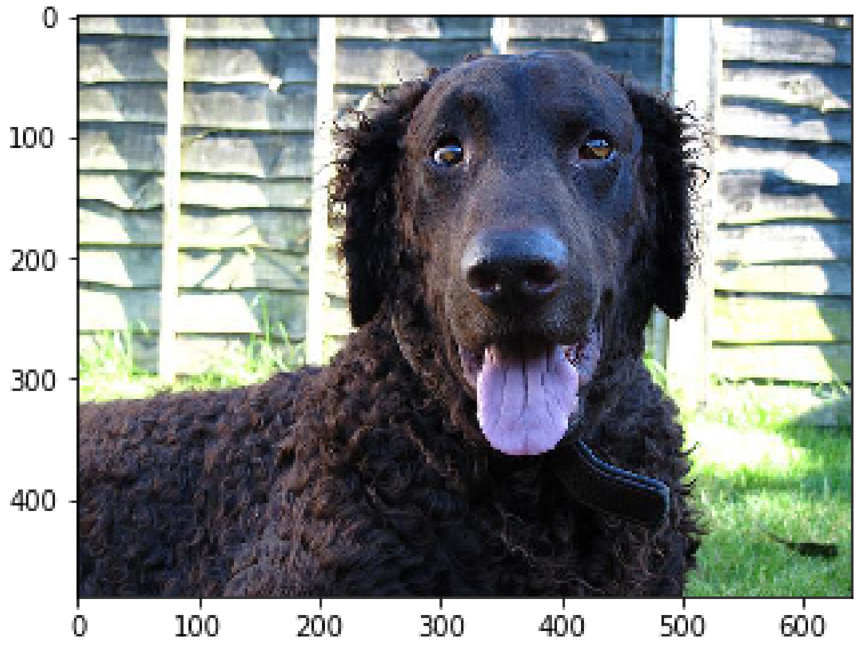
```
Dogs Detected!
It looks like a Welsh springer spaniel
```



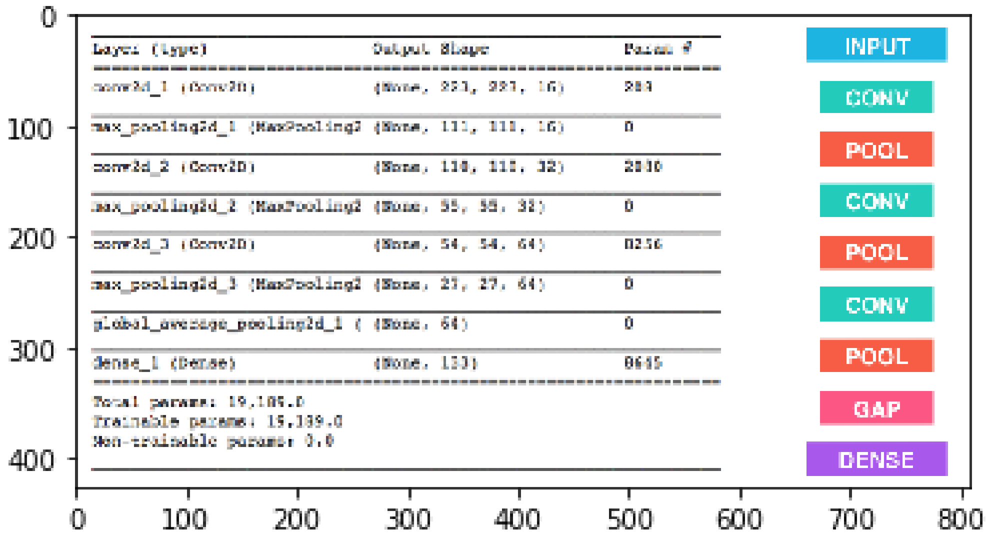
```
Hello, human!
If you were a dog..You may look like a Affenpinscher
```



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

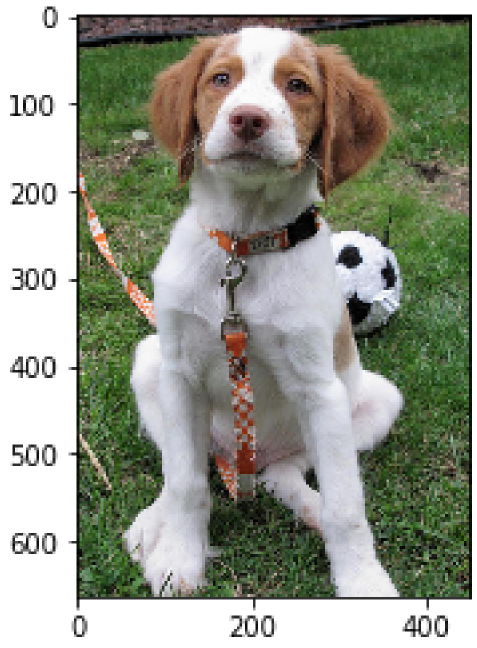


```
Dogs Detected!
It looks like a Curly-coated retriever
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

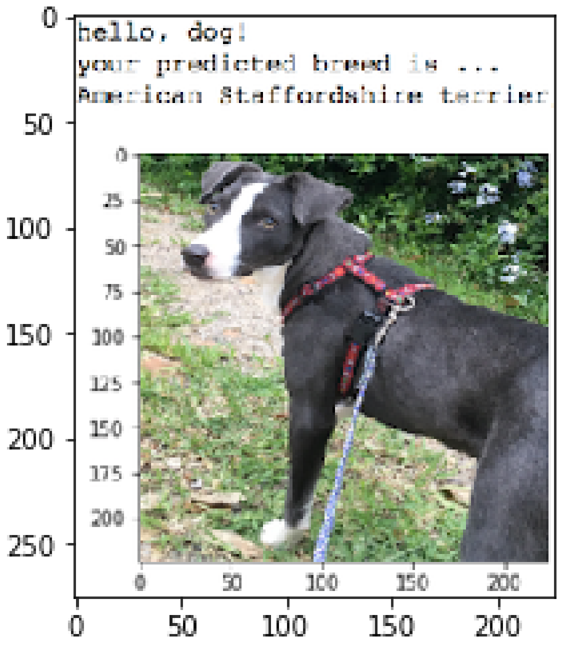
```
Error! Can't detect anything..
```



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

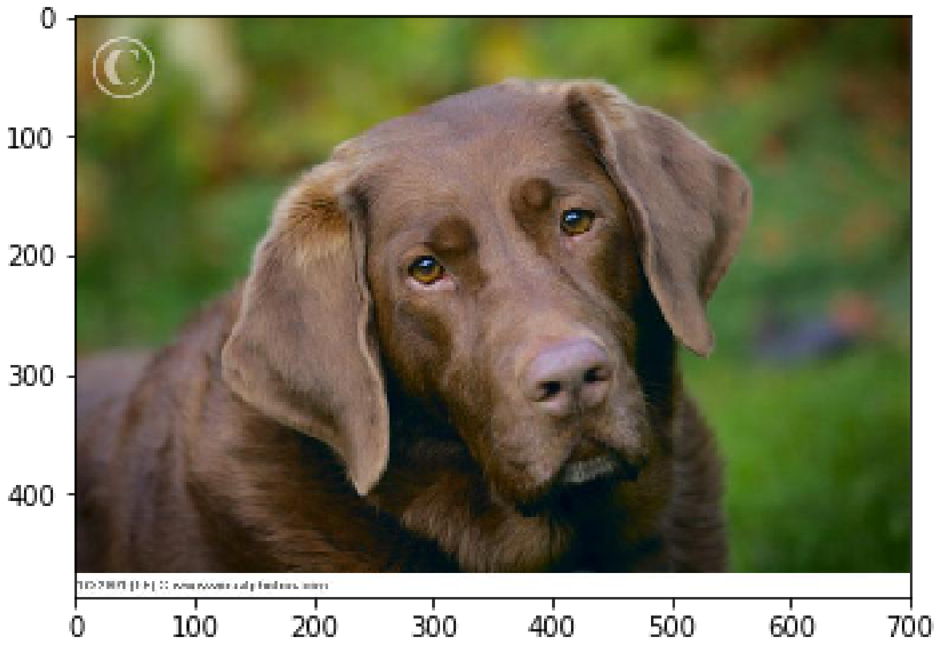
```
Dogs Detected!
It looks like a Flat-coated retriever
```

```
Dogs Detected!
It looks like a Irish water spaniel
```



```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```

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



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

## Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

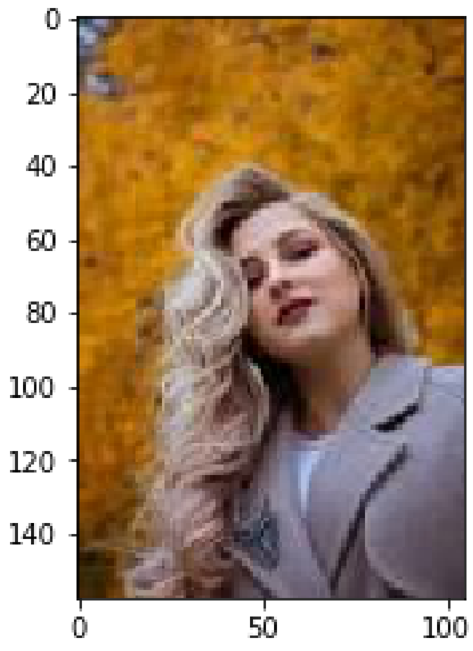
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

Answer: (Three possible points for improvement): 1. Increasing the number of images for class dog, will help in increasing the accuracy. 2. Increasing the number of Epochs 3. I tried one variant of my base light VGG-like architecuture. Will definitely try more variants. This is definitely an area of improvement

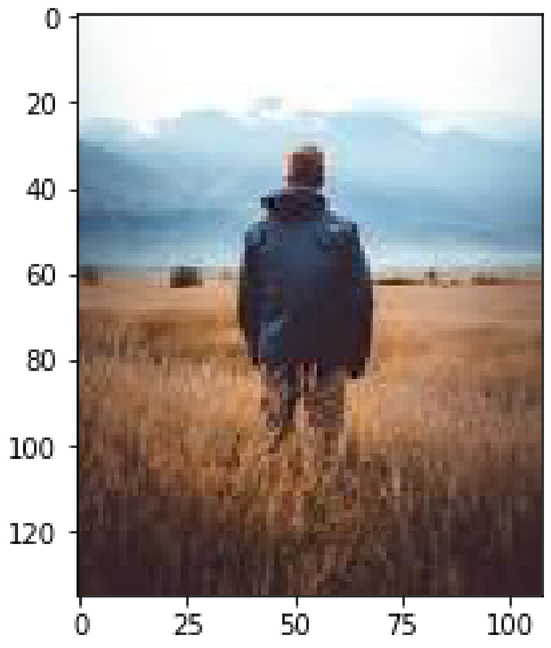
```
In [73]: human_test_files = ['./human_test_files/hum1.jpg', './human_test_files/hum2.jpg', './hu
         dog_test_files = ['./dog_test_files/dog1.jpg', './dog_test_files/dog2.jpg', './dog_test
```

```
In [74]: ## 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.
         for file in np.hstack((human_test_files, dog_test_files)):
             run_app(file)
```

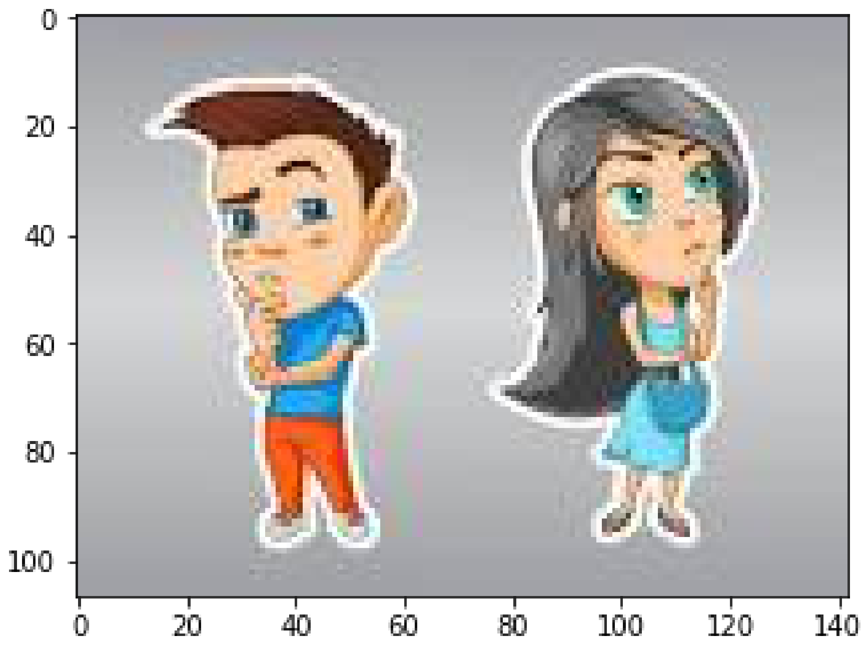


```
Error! Can't detect anything..
```

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

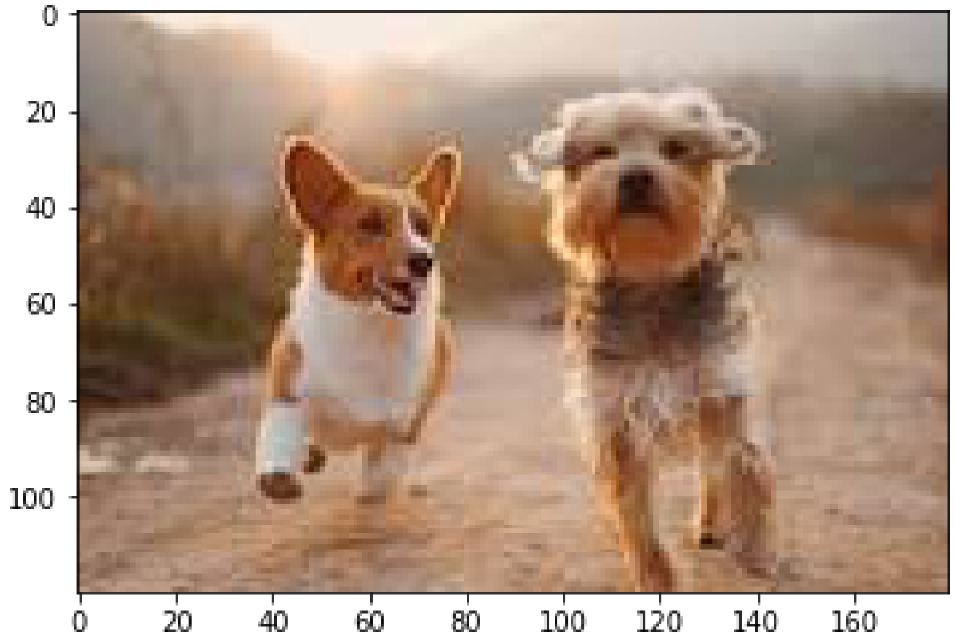


```
Error! Can't detect anything..
```

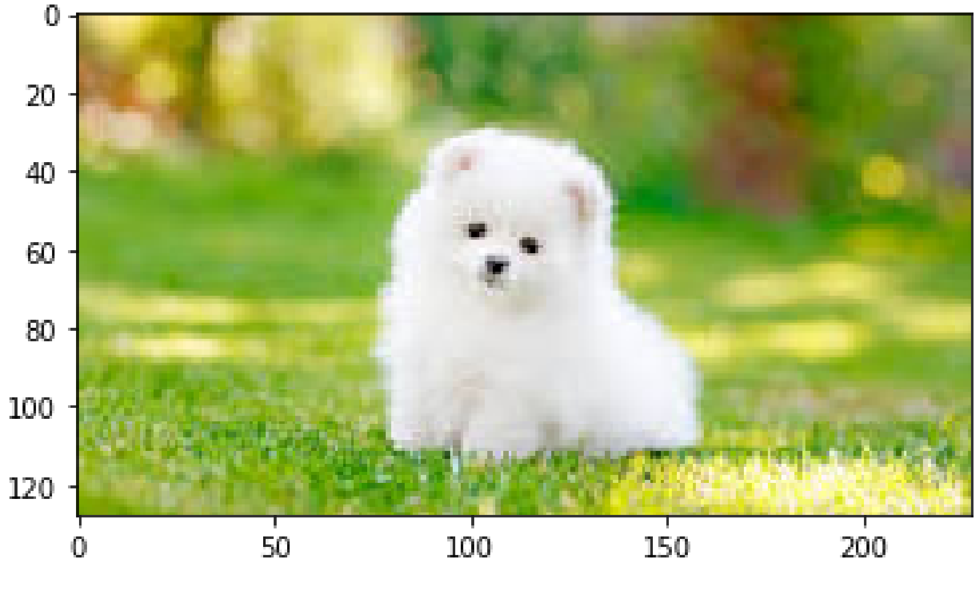


```
Error! Can't detect anything..
```

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



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



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

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

```
In [ ]:
```