dog_app

July 13, 2019

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

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

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

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

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

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

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

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

## Step 1: Detect Humans

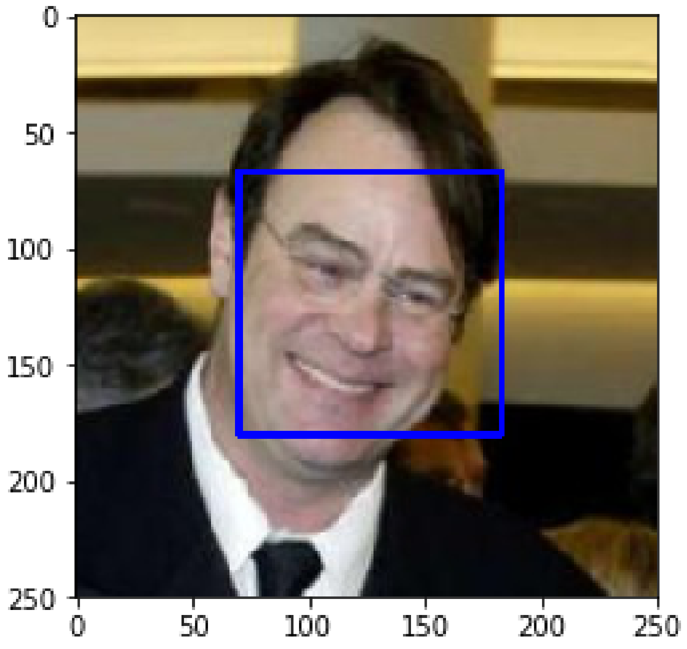
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

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

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)
Detected human face 98 out of 100 human files Detected dog face 17 out of 100 dog files:

```
In [6]: 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_perf_test(files):
            detection_cnt = 0;
            total_cnt = len(files)
            for file in files:
                detection_cnt += face_detector(file)
            return detection_cnt, total_cnt

        print("Detected human face {} out of {} human files".format(face_detection_perf_test(hum
        print("Detected dog face {} out of {} dog files:".format(face_detection_perf_test(dog_fi
```

```
Detected human face 98 out of 100 human files
Detected dog face 17 out of 100 dog files:
```

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

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

```
In [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()
        print("Is cuda available? {0}".format(use_cuda))

        # 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 [00:07<00:00, 72138292.41it/s]


Is cuda available? True
```

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

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

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

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

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

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

            Args:
                img_path: path to an image

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

            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            ## Return the *index* of the predicted class for that image


            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()])

            # discard the transparent, alpha channel (that's the :3)
            image = in_transform(image)[:3,:,:].unsqueeze(0)


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

            # return predicted class index
            return torch.max(ret,1)[1].item()

        # predict dog using ImageNet class
        VGG16_predict(dog_files_short[0])

Out[8]: 243
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [10]: def dog_detector(img_path):
             idx = VGG16_predict(img_path)
             return idx >= 151 and idx <= 268


         print("Dog Found in dog files {}".format(dog_detector(dog_files_short[0])))
         print("Dog Found in human files {}".format(dog_detector(human_files_short[0])))
```

```
Dog Found in dog files True
Dog Found in human files False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [11]: ### 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

         print("Detected a {} dog in {} human files".format(dog_detector_test(human_files_short)
         print("Detected a {} dog in {} dog files".format(dog_detector_test(dog_files_short)[0],
```

```
Detected a 0 dog in 100 human files
Detected a 96 dog in 100 dog files
```

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

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

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

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

Brittany	Welsh Springer Spaniel

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

Curly-Coated Retriever	American Water Spaniel

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

Yellow Labrador	Chocolate Labrador

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

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

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

```
In [14]: import os
         from torchvision import datasets
         import torchvision.transforms as transforms
         import torch
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True


batch_size = 20
num_workers = 0

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/')

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


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])
                  }


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'])


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: RandomResizedCrop & RandomHorizontalFlip are used on train data. That is used for image resizing and augmantation. Image augmentation gives randomnes and that help preventing overfitting.

Validation data is Resize at 256 and center cropped to make 224 x224. This configuration works well with pre-trained models.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [16]: import torch.nn as nn
         import torch.nn.functional as F
         import numpy as np

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         num_classes = 133

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 ## Define layers of a CNN
                 self.conv1 = nn.Conv2d(3, 32, 3, stride=2, padding=1)
                 self.conv2 = nn.Conv2d(32, 64, 3, stride=2, padding=1)
                 self.conv3 = nn.Conv2d(64, 128, 3, padding=1)

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

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

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

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

        # flatten
        x = x.view(-1, 7*7*128)

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

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


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

# instantiate the CNN
model_scratch = Net()


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

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

Answer: First 2 conv layers: kernel_size of 3 with stride 2, this will lead to downsize of input image by 2. Maxpooling with stride 2 is placed after 2 conv layers to help to downsize of input image by 2.

The 3rd conv layers is consist of kernel_size of 3 with stride 1, and this will not reduce input image. after final maxpooling with stride 2, the total output image size is downsized by factor of 32 and the depth will be 128. Dropout is applied to prevent overfitting.

2nd fully-connected layer is intended to produce final output size which predicts classes of breeds.

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 [17]: import torch.optim as optim
```

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr = 0.05)
```

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

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

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

                     output = model(data)

                     # calculate loss
                     loss = criterion(output, target)

                     # back prop
                     loss.backward()

                     # grad
                     optimizer.step()
```

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

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

            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                output = model(data)
                loss = criterion(output, target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


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

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

        # return trained model
        return model


    # train the model
    model_scratch = train(10, 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, Batch 1 loss: 4.093311
Epoch 1, Batch 101 loss: 4.079073
Epoch 1, Batch 201 loss: 4.069899
Epoch 1, Batch 301 loss: 4.070626
Epoch: 1        Training Loss: 4.064924        Validation Loss: 3.897441
Validation loss decreased (inf --> 3.897441).  Saving model ...
Epoch 2, Batch 1 loss: 3.834417
Epoch 2, Batch 101 loss: 4.027648
Epoch 2, Batch 201 loss: 4.042275
Epoch 2, Batch 301 loss: 4.036431
Epoch: 2        Training Loss: 4.035769        Validation Loss: 3.867630
Validation loss decreased (3.897441 --> 3.867630).  Saving model ...
Epoch 3, Batch 1 loss: 3.748473
Epoch 3, Batch 101 loss: 3.985578
Epoch 3, Batch 201 loss: 3.981268
Epoch 3, Batch 301 loss: 3.972224
Epoch: 3        Training Loss: 3.977233        Validation Loss: 3.853854
Validation loss decreased (3.867630 --> 3.853854).  Saving model ...
Epoch 4, Batch 1 loss: 3.918594
Epoch 4, Batch 101 loss: 3.959559
Epoch 4, Batch 201 loss: 3.913083
Epoch 4, Batch 301 loss: 3.917247
Epoch: 4        Training Loss: 3.915293        Validation Loss: 3.723152
Validation loss decreased (3.853854 --> 3.723152).  Saving model ...
Epoch 5, Batch 1 loss: 3.543007
Epoch 5, Batch 101 loss: 3.850476
Epoch 5, Batch 201 loss: 3.851946
Epoch 5, Batch 301 loss: 3.866573
Epoch: 5        Training Loss: 3.869247        Validation Loss: 3.698117
Validation loss decreased (3.723152 --> 3.698117).  Saving model ...
Epoch 6, Batch 1 loss: 4.542932
Epoch 6, Batch 101 loss: 3.838531
Epoch 6, Batch 201 loss: 3.838859
Epoch 6, Batch 301 loss: 3.828118
Epoch: 6        Training Loss: 3.830600        Validation Loss: 3.708741
Epoch 7, Batch 1 loss: 3.532290
Epoch 7, Batch 101 loss: 3.756562
Epoch 7, Batch 201 loss: 3.760272
Epoch 7, Batch 301 loss: 3.771093
Epoch: 7        Training Loss: 3.778624        Validation Loss: 3.640538
Validation loss decreased (3.698117 --> 3.640538).  Saving model ...
Epoch 8, Batch 1 loss: 3.900812
Epoch 8, Batch 101 loss: 3.769214
Epoch 8, Batch 201 loss: 3.761209
Epoch 8, Batch 301 loss: 3.746613
Epoch: 8        Training Loss: 3.749504        Validation Loss: 3.628509
Validation loss decreased (3.640538 --> 3.628509).  Saving model ...
Epoch 9, Batch 1 loss: 3.781700
Epoch 9, Batch 101 loss: 3.694990
Epoch 9, Batch 201 loss: 3.730481
Epoch 9, Batch 301 loss: 3.720459
Epoch: 9         Training Loss: 3.716258         Validation Loss: 3.637510
Epoch 10, Batch 1 loss: 3.506036
Epoch 10, Batch 101 loss: 3.661058
Epoch 10, Batch 201 loss: 3.668637
Epoch 10, Batch 301 loss: 3.673872
Epoch: 10         Training Loss: 3.679499         Validation Loss: 3.570432
Validation loss decreased (3.628509 --> 3.570432).  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 [22]: 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: 3.721468
Test Accuracy: 13% (111/836)
```

## 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 [23]: ## 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 [24]: 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 [00:03<00:00, 25865893.63it/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: Selected model ResNet performed best on image classification.

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 [25]: 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
         #model_transfer = # train(n_epochs, loaders_transfer, model_transfer, optimizer_transfe

         # load the model that got the best validation accuracy (uncomment the line below)
         #model_transfer.load_state_dict(torch.load('model_transfer.pt'))
         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             valid_loss_min = np.Inf

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

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()

                     # initialize weights to zero
                     optimizer.zero_grad()

                     output = model(data)

                     # calculate loss
                     loss = criterion(output, target)
```

```
            # back prop
            loss.backward()

            # grad
            optimizer.step()

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

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

        ######################
        # validate the model #
        ######################
        model.eval()
        for batch_idx, (data, target) in enumerate(loaders['valid']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            ## update the average validation loss
            output = model(data)
            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_loss)


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

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

    # return trained model
    return model

train(20, loaders_transfer, model_transfer, optimizer_transfer, criterion_transfer, use


# load the model that got the best validation accuracy
```

```
        model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch 1, Batch 1 loss: 2.350386
Epoch 1, Batch 101 loss: 2.284555
Epoch 1, Batch 201 loss: 2.285551
Epoch 1, Batch 301 loss: 2.277313
Epoch: 1         Training Loss: 2.273011         Validation Loss: 1.686224
Validation loss decreased (inf --> 1.686224).  Saving model ...
Epoch 2, Batch 1 loss: 2.185141
Epoch 2, Batch 101 loss: 2.244076
Epoch 2, Batch 201 loss: 2.235444
Epoch 2, Batch 301 loss: 2.224876
Epoch: 2         Training Loss: 2.221145         Validation Loss: 1.618313
Validation loss decreased (1.686224 --> 1.618313).  Saving model ...
Epoch 3, Batch 1 loss: 2.097821
Epoch 3, Batch 101 loss: 2.153676
Epoch 3, Batch 201 loss: 2.175415
Epoch 3, Batch 301 loss: 2.153899
Epoch: 3         Training Loss: 2.154629         Validation Loss: 1.536086
Validation loss decreased (1.618313 --> 1.536086).  Saving model ...
Epoch 4, Batch 1 loss: 2.014405
Epoch 4, Batch 101 loss: 2.117808
Epoch 4, Batch 201 loss: 2.123878
Epoch 4, Batch 301 loss: 2.122859
Epoch: 4         Training Loss: 2.119215         Validation Loss: 1.497218
Validation loss decreased (1.536086 --> 1.497218).  Saving model ...
Epoch 5, Batch 1 loss: 2.112144
Epoch 5, Batch 101 loss: 2.071465
Epoch 5, Batch 201 loss: 2.069074
Epoch 5, Batch 301 loss: 2.048761
Epoch: 5         Training Loss: 2.050658         Validation Loss: 1.473037
Validation loss decreased (1.497218 --> 1.473037).  Saving model ...
Epoch 6, Batch 1 loss: 1.885552
Epoch 6, Batch 101 loss: 1.994177
Epoch 6, Batch 201 loss: 2.011742
Epoch 6, Batch 301 loss: 1.999248
Epoch: 6         Training Loss: 2.004949         Validation Loss: 1.401448
Validation loss decreased (1.473037 --> 1.401448).  Saving model ...
Epoch 7, Batch 1 loss: 1.413045
Epoch 7, Batch 101 loss: 1.980999
Epoch 7, Batch 201 loss: 1.986720
Epoch 7, Batch 301 loss: 1.978199
Epoch: 7         Training Loss: 1.975926         Validation Loss: 1.357703
Validation loss decreased (1.401448 --> 1.357703).  Saving model ...
Epoch 8, Batch 1 loss: 1.943936
Epoch 8, Batch 101 loss: 1.937140
Epoch 8, Batch 201 loss: 1.944394
Epoch 8, Batch 301 loss: 1.936126
```

```
Epoch: 8        Training Loss: 1.931879        Validation Loss: 1.309746
Validation loss decreased (1.357703 --> 1.309746).  Saving model ...
Epoch 9, Batch 1 loss: 1.951793
Epoch 9, Batch 101 loss: 1.901610
Epoch 9, Batch 201 loss: 1.898619
Epoch 9, Batch 301 loss: 1.900769
Epoch: 9        Training Loss: 1.897734        Validation Loss: 1.279616
Validation loss decreased (1.309746 --> 1.279616).  Saving model ...
Epoch 10, Batch 1 loss: 1.804905
Epoch 10, Batch 101 loss: 1.854092
Epoch 10, Batch 201 loss: 1.899708
Epoch 10, Batch 301 loss: 1.887789
Epoch: 10        Training Loss: 1.885328        Validation Loss: 1.246197
Validation loss decreased (1.279616 --> 1.246197).  Saving model ...
Epoch 11, Batch 1 loss: 1.856759
Epoch 11, Batch 101 loss: 1.817694
Epoch 11, Batch 201 loss: 1.835331
Epoch 11, Batch 301 loss: 1.833365
Epoch: 11        Training Loss: 1.830395        Validation Loss: 1.197896
Validation loss decreased (1.246197 --> 1.197896).  Saving model ...
Epoch 12, Batch 1 loss: 1.514580
Epoch 12, Batch 101 loss: 1.796287
Epoch 12, Batch 201 loss: 1.797184
Epoch 12, Batch 301 loss: 1.803827
Epoch: 12        Training Loss: 1.801712        Validation Loss: 1.186543
Validation loss decreased (1.197896 --> 1.186543).  Saving model ...
Epoch 13, Batch 1 loss: 2.438747
Epoch 13, Batch 101 loss: 1.752473
Epoch 13, Batch 201 loss: 1.758562
Epoch 13, Batch 301 loss: 1.756177
Epoch: 13        Training Loss: 1.759254        Validation Loss: 1.159904
Validation loss decreased (1.186543 --> 1.159904).  Saving model ...
Epoch 14, Batch 1 loss: 1.710798
Epoch 14, Batch 101 loss: 1.776765
Epoch 14, Batch 201 loss: 1.759389
Epoch 14, Batch 301 loss: 1.742555
Epoch: 14        Training Loss: 1.742235        Validation Loss: 1.150779
Validation loss decreased (1.159904 --> 1.150779).  Saving model ...
Epoch 15, Batch 1 loss: 2.161588
Epoch 15, Batch 101 loss: 1.711720
Epoch 15, Batch 201 loss: 1.749598
Epoch 15, Batch 301 loss: 1.721810
Epoch: 15        Training Loss: 1.721262        Validation Loss: 1.080289
Validation loss decreased (1.150779 --> 1.080289).  Saving model ...
Epoch 16, Batch 1 loss: 1.682941
Epoch 16, Batch 101 loss: 1.672051
Epoch 16, Batch 201 loss: 1.672442
Epoch 16, Batch 301 loss: 1.669082
```

```
Epoch: 16         Training Loss: 1.672202         Validation Loss: 1.113229
Epoch 17, Batch 1 loss: 1.475960
Epoch 17, Batch 101 loss: 1.634007
Epoch 17, Batch 201 loss: 1.642253
Epoch 17, Batch 301 loss: 1.645759
Epoch: 17         Training Loss: 1.643476         Validation Loss: 1.068408
Validation loss decreased (1.080289 --> 1.068408).  Saving model ...
Epoch 18, Batch 1 loss: 1.626438
Epoch 18, Batch 101 loss: 1.586108
Epoch 18, Batch 201 loss: 1.622340
Epoch 18, Batch 301 loss: 1.643316
Epoch: 18         Training Loss: 1.645823         Validation Loss: 1.053460
Validation loss decreased (1.068408 --> 1.053460).  Saving model ...
Epoch 19, Batch 1 loss: 2.015040
Epoch 19, Batch 101 loss: 1.625617
Epoch 19, Batch 201 loss: 1.611728
Epoch 19, Batch 301 loss: 1.609136
Epoch: 19         Training Loss: 1.606944         Validation Loss: 1.005818
Validation loss decreased (1.053460 --> 1.005818).  Saving model ...
Epoch 20, Batch 1 loss: 1.554231
Epoch 20, Batch 101 loss: 1.588878
Epoch 20, Batch 201 loss: 1.598705
Epoch 20, Batch 301 loss: 1.597323
Epoch: 20         Training Loss: 1.596734         Validation Loss: 1.008483
```

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: 1.091834


Test Accuracy: 79% (667/836)
```

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.

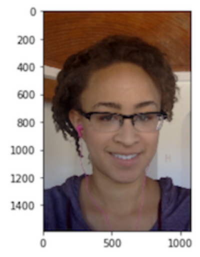
```
In [32]: ### TODO: Write a function that takes a path to an image as input
         ### and returns the dog breed that is predicted by the model.


         from PIL import Image
```

```
import torchvision.transforms as transforms

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


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]

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("{0} predition breed: {1}".format(img_path, predition))
```

```
./images/Welsh_springer_spaniel_08203.jpg predition breed: Welsh springer spaniel
./images/Labrador_retriever_06457.jpg predition breed: Labrador retriever
./images/Labrador_retriever_06455.jpg predition breed: Chesapeake bay retriever
./images/sample_human_output.png predition breed: Brussels griffon
./images/Brittany_02625.jpg predition breed: Brittany
./images/American_water_spaniel_00648.jpg predition breed: Curly-coated retriever
./images/Labrador_retriever_06449.jpg predition breed: Flat-coated retriever
./images/Curly-coated_retriever_03896.jpg predition breed: Curly-coated retriever
./images/sample_cnn.png predition breed: Brussels griffon
./images/sample_dog_output.png predition breed: Great dane
```

## 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.

```
hello, human!
```



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

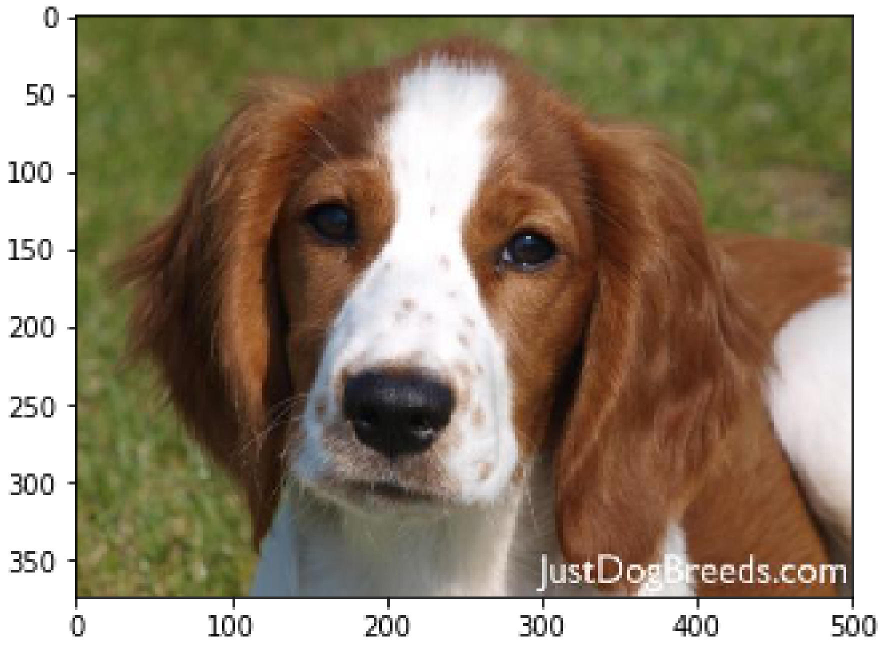
Sample Human Output

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.

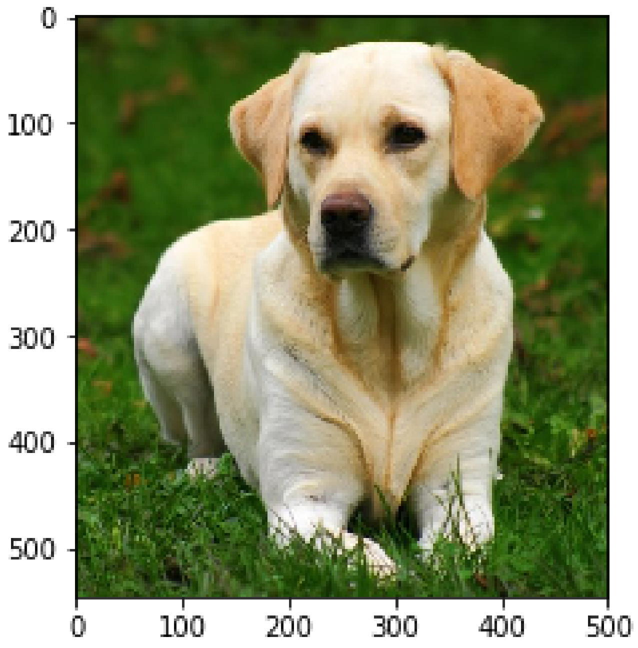
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

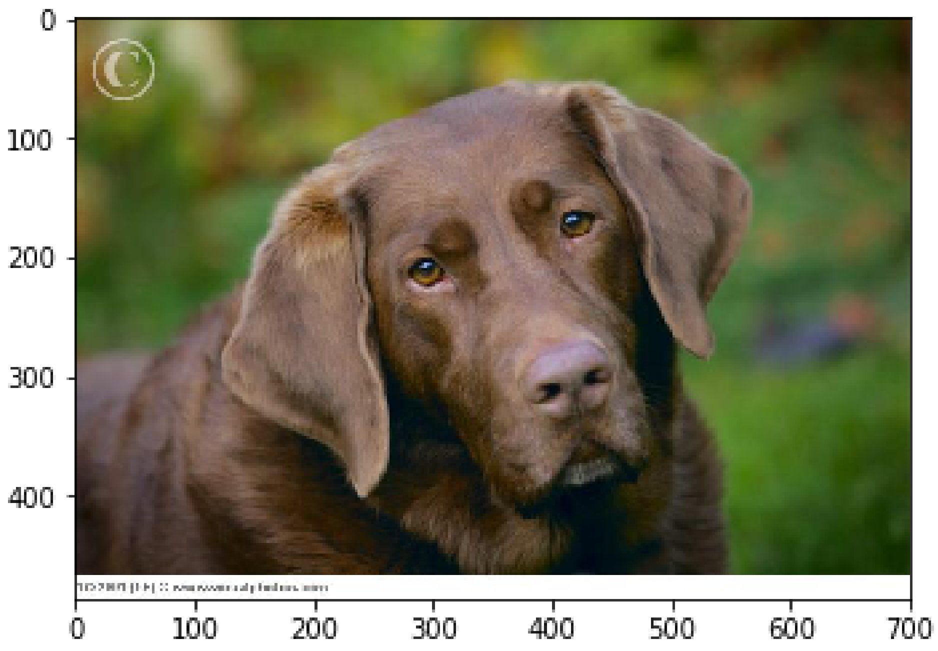
In [34]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

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..")


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



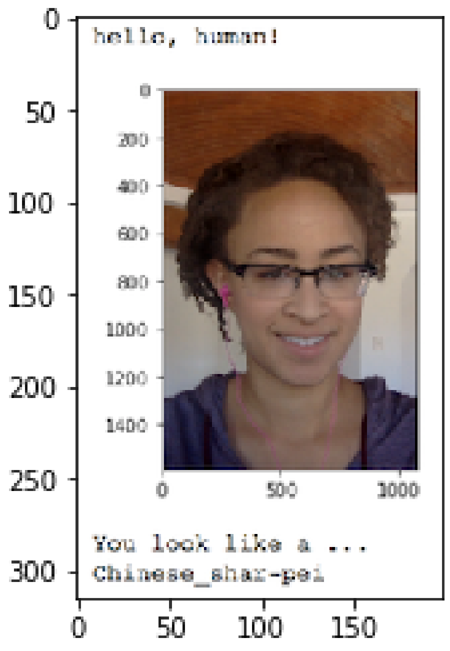
```
Dogs Detected!
It looks like a Welsh springer spaniel
```



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

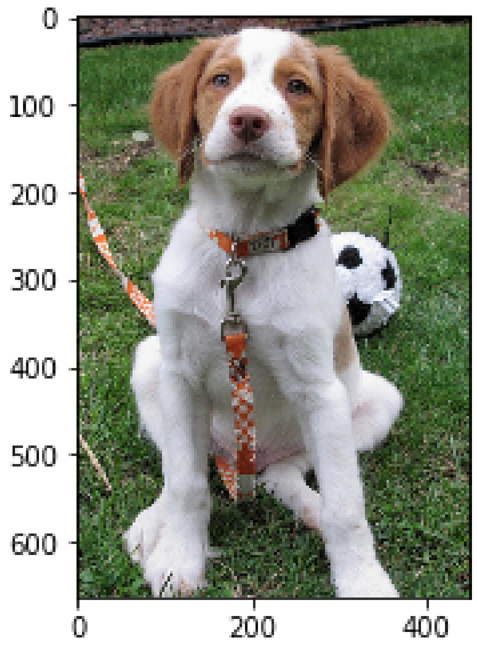


```
Dogs Detected!
It looks like a Chesapeake bay retriever
```



```
hello, human!
You look like a ...
Chinese_shar-pei
```

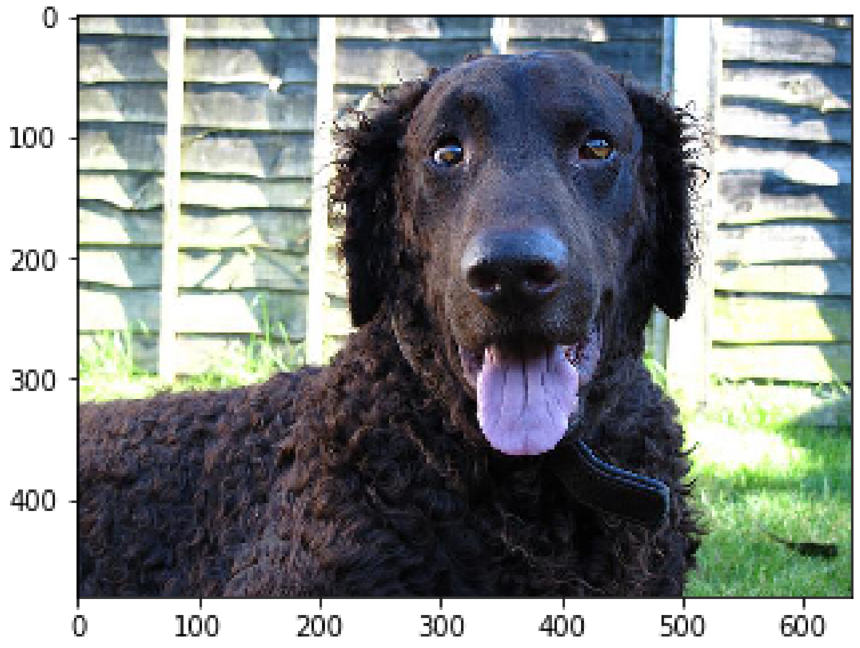
```
Hello, human!
If you were a dog..You may look like a Brussels griffon
```



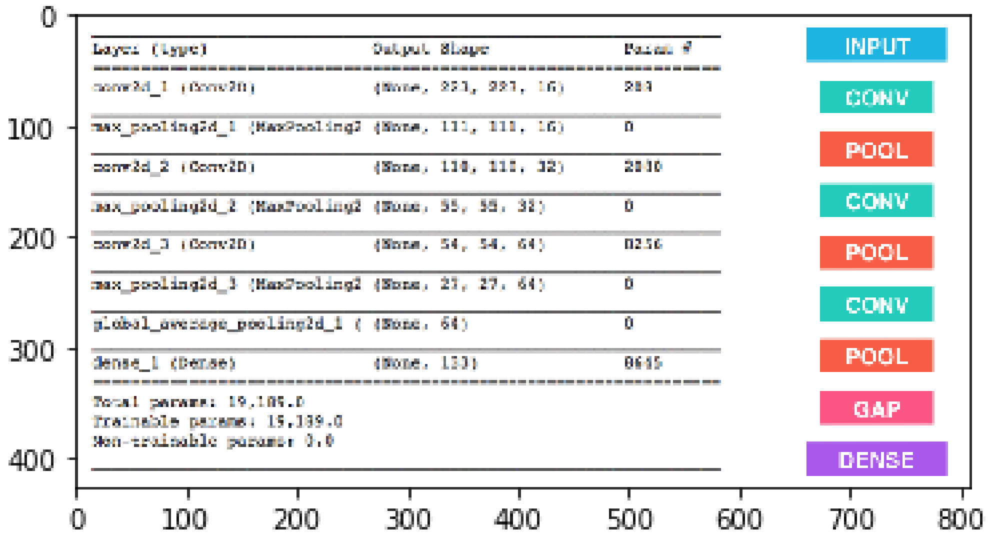
```
Dogs Detected!
It looks like a Brittany
```

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

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

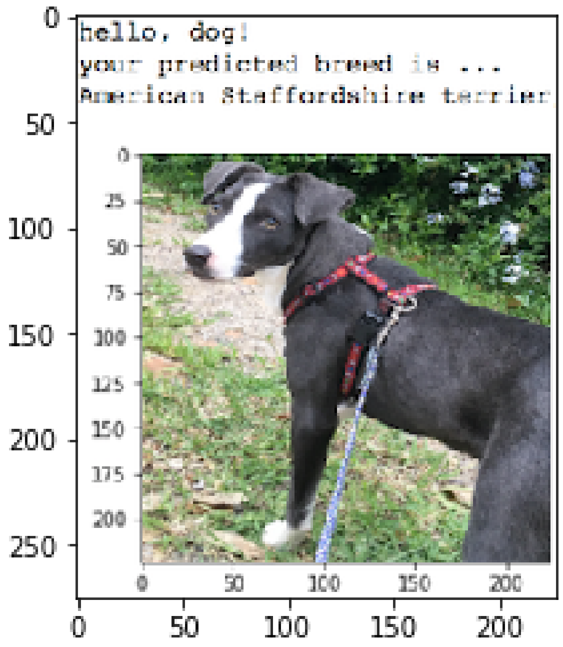


```
Dogs Detected!
It looks like a Curly-coated retriever
```



```
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 Great dane
```

## 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!

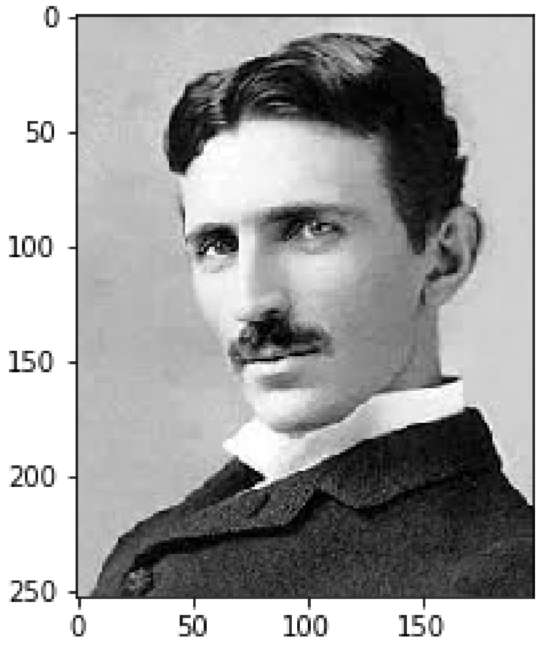
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

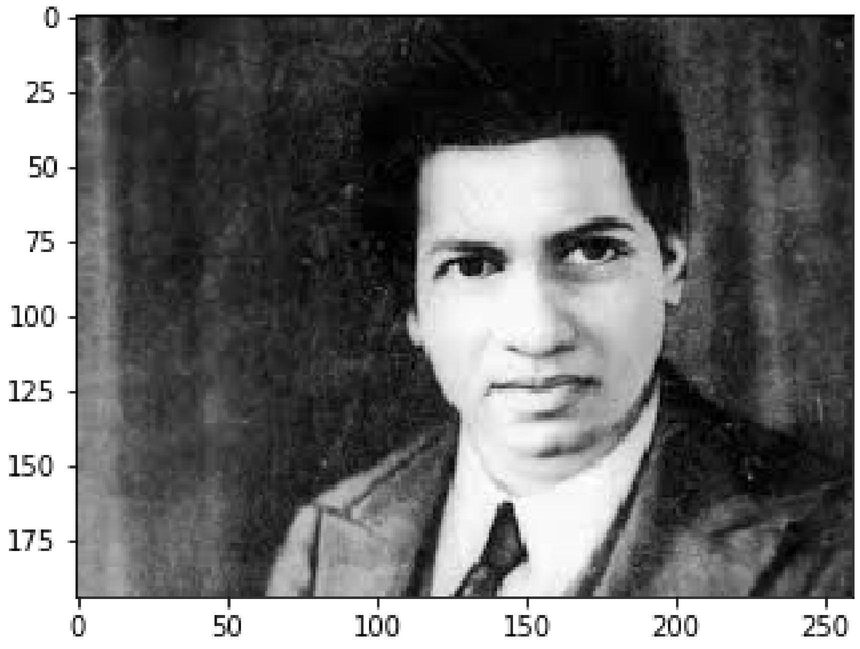
Answer: (Three possible points for improvement) Areas of improvement 1. With bigger training data we can improve accuracy. Current model only applied RandomResizedCrop & RandomHorizontalFlip but with more augmentations like Vertical Flips will improve accuracy. With current mode may miss images with flipped images.
2. Test with couple of models. Current model worked well for single dog in image. Need to test it with multiple dogs in same image. 3. Can be optimized and tested with more hyper parameters.

```
In [44]: ## 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.
         human_files = ['./test_images/tesla.jpeg', './test_images/Ramanujan.jpeg', './test_imag
         dog_files = ['./test_images/Pug.jpeg', './test_images/Labrador.jpg', './test_images/Pom

         import os
         print(os.getcwd())

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

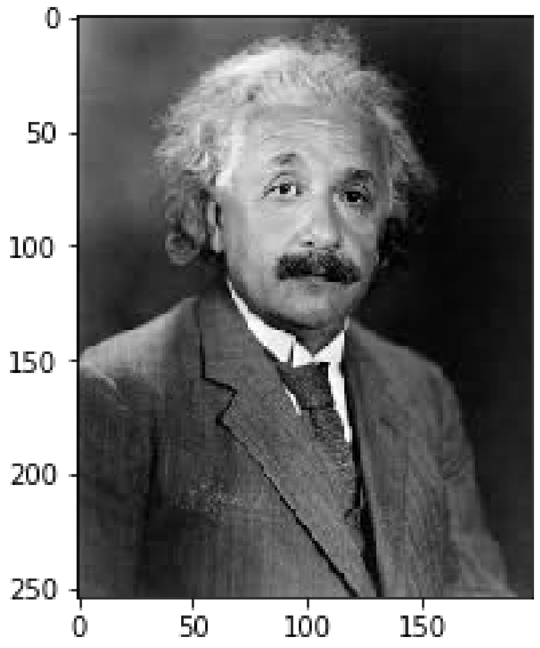
```
/home/workspace/dog_project
```



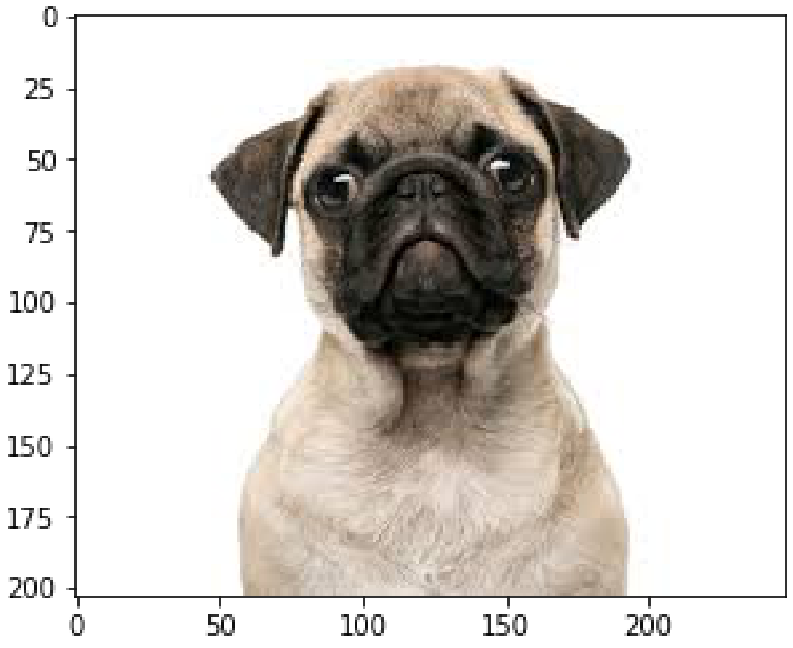
```
Hello, human!
If you were a dog..You may look like a American foxhound
```



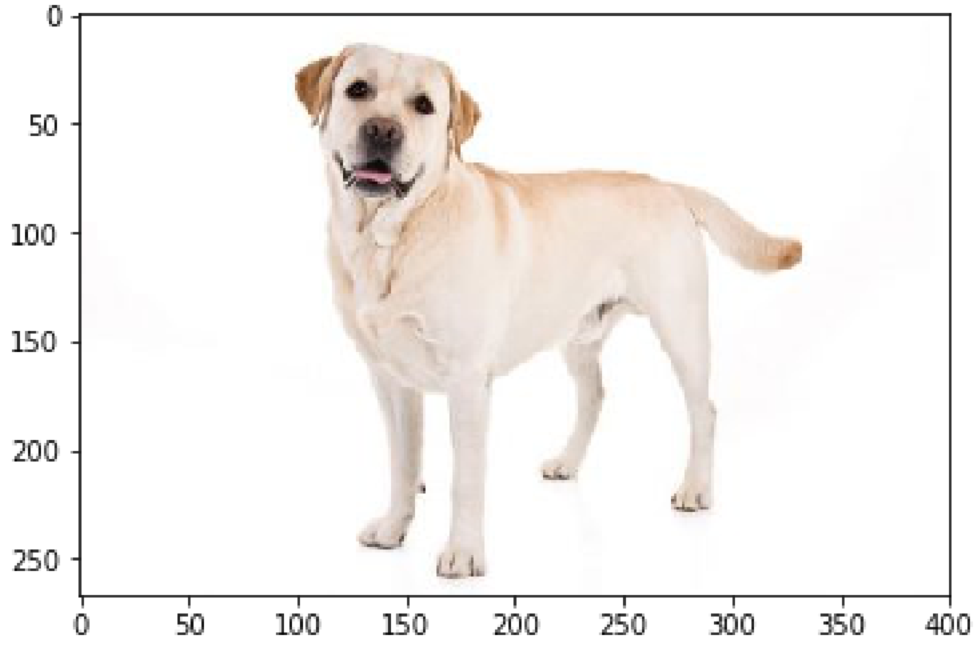
```
Hello, human!
If you were a dog..You may look like a Greyhound
```



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



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



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

```
Dogs Detected!
It looks like a American eskimo dog
```