dog_app

October 1, 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 [16]: 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

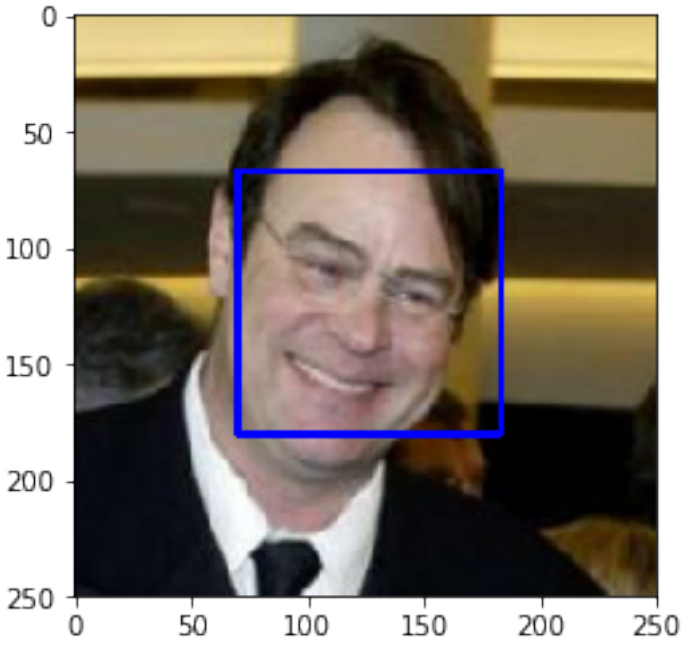
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [17]: 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 [18]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)

```
In [21]: 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 faces_detected_percent(images):
             detected_cnt = 0

             for img in images:
                 detected_cnt += face_detector(img)

             return round((detected_cnt/len(images))*100)

In [22]: print(f"Percentage of human faces detected: {faces_detected_percent(human_files_short)}
         print(f"Percentage of faces detected in dog files: {faces_detected_percent(dog_files_sh

Percentage of human faces detected: 98%
Percentage of faces detected in dog files: 17%
```

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 [23]: ### (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 [24]: 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()
         device = torch.device("cuda" if use_cuda else "cpu")

         # move model to GPU if CUDA is available
         if use_cuda:
             VGG16 = VGG16.cuda()
```

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

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

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

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

In [25]:
```
from PIL import Image
import torchvision.transforms as transforms

def load_image(img_path):
    image = Image.open(img_path).convert('RGB')

    #Resize and normalize input images, refer - https://pytorch.org/docs/stable/torchvi
    in_transform = transforms.Compose([
        transforms.Resize(size=(224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5],
                             std=[0.5, 0.5, 0.5])])


    #discard the transparent, aplha channel (:3), and add the batch dimension
    image = in_transform(image)[:3,:,:].unsqueeze(0)
    return image

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
    img = load_image(img_path).to(device)

    prediction = VGG16(img) #[1, 1000] tensor of class predictions

    max_prediction = torch.max(prediction,1)
    max_prediction_index = max_prediction[1]

    ## Return the *index* of the predicted class for that image
    return max_prediction_index.item()
```

In [26]:
```
VGG16_predict(dog_files_short[0])
```

Out[26]: 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 [27]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             img = load_image(img_path).to(device)

             prediction = VGG16(img)

             prediction_class_index = torch.max(prediction,1)[1].item()

             return prediction_class_index in range(151,268)
```

```
In [28]: dog_detector(dog_files_short[0])
```

```
Out[28]: True
```

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 [29]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         def dog_detector_accuracy(images):
             num_detected = 0
             for image in images:
                 num_detected += dog_detector(image)
             return (num_detected/len(images))*100
```

```
In [30]: print(f'Detected dogs in {len(dog_files_short)} dog images with accuracy: {dog_detector
         print(f'Detected dogs in {len(human_files_short)} human images with accuracy: {dog_dete
```

```
Detected dogs in 100 dog images with accuracy: 100.0%
Detected dogs in 100 human images with accuracy: 1.0%
```

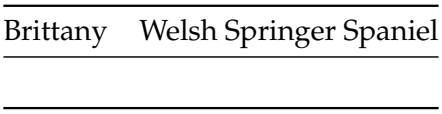
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

```
In [31]: ### (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)

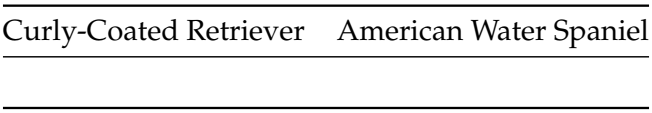
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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



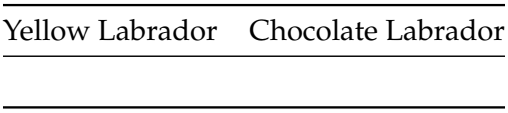
Brittany Welsh Springer Spaniel

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 [64]:
```
import os
from torchvision import datasets

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

#number of subprocesses to use for data loading
num_workers = 0
#number of samples per batch to load
batch_size = 5

data_dir = '/data/dog_images/'

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

data_transforms = {'train': transforms.Compose([
                                    transforms.Resize(size=(224, 224)),
                                    transforms.RandomHorizontalFlip(), #randomly flip a
                                    transforms.RandomRotation(10), #randomly rotate ima
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                    ]),
                   'valid': transforms.Compose([
                                    transforms.Resize(size=(224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                    ]),
                    'test': transforms.Compose([
                                    transforms.Resize(size=(224, 224)),
                                    transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                    ])
                  }



training_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
validation_dataset = datasets.ImageFolder(validation_dir, transform=data_transforms['va
test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

print(f'Number of training images: {len(training_dataset)}')
print(f'Number of validation images: {len(validation_dataset)}')
print(f'Number of validation images: {len(test_dataset)}')

#Define data loaders
train_loader = torch.utils.data.DataLoader(training_dataset,
```

```
                                                batch_size=batch_size,
                                                num_workers=num_workers,
                                                shuffle=True)
        valid_loader = torch.utils.data.DataLoader(validation_dataset,
                                                batch_size=batch_size,
                                                num_workers=num_workers,
                                                shuffle=True)
        test_loader = torch.utils.data.DataLoader(test_dataset,
                                                batch_size=batch_size,
                                                num_workers=num_workers,
                                                shuffle=True)

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

```
Number of training images: 6680
Number of validation images: 835
Number of validation images: 836
```

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? Yes, to give the data some geometric variation, I used RandomHorizontalFlip to randomly flip an image along it's horizontal image, and RandomRotation to randomly rotate images 10 degrees. The transformations introduce rotation and translation invariance in the training data.

Answer: - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why?

I used a Resize transformation to resize images to 224 x 224 to emulate a VGG-16 input.

- Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Yes I augmented the training dataset, not the validation and test datasets. To give the data some geometric variation, I used RandomHorizontalFlip to randomly flip an image along it's horizontal image, and RandomRotation to randomly rotate images 10 degrees. The transformations introduce rotation and translation invariance in the training data.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [65]: import torch.nn as nn
         import torch.nn.functional as F
```

```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

num_breeds = 133

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        #convolutional layer 1 (sees 224x224x3 images)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)

        #convolutional layer 2 (sees 112x112x16 tensor)
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        #convolutional layer 3 (sees 56x56x32 tensor)
        self.conv3 = nn.Conv2d(32, 64, 2, padding=1)

        #max pooling layer to down sample the xy dimensions of the input by 2
        self.pool = nn.MaxPool2d(2, 2)

        #linear layer (64 x 28 x 28 -> 500)
        self.fc1 = nn.Linear(64 * 28 * 28, 500)

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

        #dropout layer with a probability of 25%
        self.dropout = nn.Dropout(0.25)

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

        #flatten input image
        x = x.view(-1, 64 * 28 * 28)

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

1. Define 3 convolutional layers conv1: input - 224x224x3 images with depth of 3 for the r,g,b channels output - a stack of 16 feature maps convolutional kernel size - 3x3 stride - 2 padding - 1 conv2: Doubles the depth of the output to 32 conv3: Doubles the depth of the output to 64

2. Define a max pool layer with kernel size and stride of 2, to down sample the inputs by 2.

3. Define two linear layers

   layer 1: input - down sampled image of size 28x28 (224 downsampled after 3 pooling layers), produces 500 outputs

   layer 2: input - 500 outputs from the previous layer, produces 133 output classes, for the dog breeds.

4. Define a dropout layer with a probability of 25% to prevent over fitting.

5. Forward function:

   The input image is passed through a series of convolutional and pooling layers. A Relu activation function is applied after each convolutional layer. The image is then flatten to produce a vector that is fed into the linear layers. A dropout layer is applied before each of the two linear layers. The first linear layer takes in the downsampled image as input and generates 500 outputs. The second linear layer takes in these 500 outputs as inputs and generates 133 output classes, for the dog breeds.

6. Network architecture: Net((conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (conv3): Conv2d(32, 64, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1)) (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (fc1): Linear(in_features=200704, out_features=500, bias=True) (fc2): Linear(in_features=500, out_features=133, bias=True) (dropout): Dropout(p=0.25))

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

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

### TODO: select optimizer
optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.01)
```

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 [82]: 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

                     #initialize weights to zero
                     optimizer.zero_grad()

                     output = model(data)

                     loss = criterion(output, target)

                     #back propagation
                     loss.backward()

                     #gradient descent step
                     optimizer.step()

                     ## record the average training loss
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - 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)

                    ## record the average validation loss
                    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 state.'
                    valid_loss_min,
                    valid_loss))
                    valid_loss_min = valid_loss

            # return trained model
            return model

In [85]: # train the model
         model_scratch = train(10, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.018727         Validation Loss: 4.175555
Validation loss decreased: (inf --> 4.175555), saving model state.
Epoch: 2         Training Loss: 3.850160         Validation Loss: 4.079144
Validation loss decreased: (4.175555 --> 4.079144), saving model state.
Epoch: 3         Training Loss: 3.701059         Validation Loss: 4.039497
Validation loss decreased: (4.079144 --> 4.039497), saving model state.
Epoch: 4         Training Loss: 3.501649         Validation Loss: 3.950392
Validation loss decreased: (4.039497 --> 3.950392), saving model state.
Epoch: 5         Training Loss: 3.330673         Validation Loss: 3.899379
Validation loss decreased: (3.950392 --> 3.899379), saving model state.
```

```
Epoch: 6         Training Loss: 3.081876         Validation Loss: 3.911480
Epoch: 7         Training Loss: 2.886781         Validation Loss: 4.001039
Epoch: 8         Training Loss: 2.635304         Validation Loss: 4.154109
Epoch: 9         Training Loss: 2.410137         Validation Loss: 4.152674
Epoch: 10         Training Loss: 2.156147         Validation Loss: 4.334012
```

```
In [86]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

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 [87]: 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))
```

```
In [88]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.826551
```

```
Test Accuracy: 12% (106/836)
```

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 [32]: import os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

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

         #number of subprocesses to use for data loading
         num_workers = 0
         #number of samples per batch to load
         batch_size = 20

         data_dir = '/data/dog_images/'

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


         data_transforms = {'train': transforms.Compose([
                                          transforms.Resize(size=(224, 224)),
                                          transforms.RandomHorizontalFlip(), #randomly flip a
                                          transforms.RandomRotation(10), #randomly rotate ima
                                          transforms.ToTensor(),
                                          transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                          ]),
                            'valid': transforms.Compose([
```

```
                                        transforms.Resize(size=(224, 224)),
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                        ]),
                                'test': transforms.Compose([
                                        transforms.Resize(size=(224, 224)),
                                        transforms.ToTensor(),
                                        transforms.Normalize((0.485, 0.456, 0.406), (0.229,
                                        ])
                            }



        training_dataset = datasets.ImageFolder(train_dir, transform=data_transforms['train'])
        validation_dataset = datasets.ImageFolder(validation_dir, transform=data_transforms['va
        test_dataset = datasets.ImageFolder(test_dir, transform=data_transforms['test'])

        print(f'Number of training images: {len(training_dataset)}')
        print(f'Number of validation images: {len(validation_dataset)}')
        print(f'Number of validation images: {len(test_dataset)}')

        #Define data loaders
        train_loader = torch.utils.data.DataLoader(training_dataset,
                                                   batch_size=batch_size,
                                                   num_workers=num_workers,
                                                   shuffle=True)
        valid_loader = torch.utils.data.DataLoader(validation_dataset,
                                                   batch_size=batch_size,
                                                   num_workers=num_workers,
                                                   shuffle=True)
        test_loader = torch.utils.data.DataLoader(test_dataset,
                                                  batch_size=batch_size,
                                                  num_workers=num_workers,
                                                  shuffle=True)

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

Number of training images: 6680
Number of validation images: 835
Number of validation images: 836
```

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 [71]: import torchvision.models as models
         import torch.nn as nn

         num_dog_breeds = 133

         # check if CUDA is available
         use_cuda = torch.cuda.is_available()
         device = torch.device("cuda" if use_cuda else "cpu")

         ## TODO: Specify model architecture
         model_transfer = models.vgg16(pretrained=True)

         if use_cuda:
             model_transfer = model_transfer.cuda()

         #Freeze training for all "features" layers. "features" is the group of all convolutiona
         for param in model_transfer.features.parameters():
             param.require_grad = False

         n_inputs = model_transfer.classifier[6].in_features
         last_layer = nn.Linear(n_inputs, num_dog_breeds, bias=True)
         model_transfer.classifier[6] = last_layer
```

```
In [81]: print(model_transfer)
```

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

```
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

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

Answer:

I am starting with the pre-trained VGG-16 network. The VGG-16 network has already been trained on the Imagenet dataset that contains 1000 image classes, including images of dogs. Also, since our training data set is small (6680 images) and similar to the imagenet dataset, VGG-16 is a good choice.

The first few convolutional layers are used as feature extractors that extract features that are common to any image in any dataset, such as colors and shapes. The weights in these convolutional layers are frozen (param.require_grad=False) and their pre-trained values are reused.

The final classification layer is replaced with one that produces 133 classes for the dog breeds and will act as the final dog breed classifier.

Only the last linear layers in the classifier group will be fully trained.

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

```
criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.SGD(model_transfer.classifier.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 [60]:
```
import numpy as np

def train(n_epocs, loaders, model, optimizer, criterion, use_cuda, save_path):

    valid_loss_min = np.Inf #to track change in validation loss

    for epoch in range(1, n_epochs+1):
        ###################
        # train the model #
        ###################
        train_loss = 0.0

        model.train() #training mode

        for batch_num, (data, target) in enumerate(loaders['train']):
            #move tensors to GPU
            if use_cuda:
                data, target = data.to(device), target.to(device)

            #reset the gradients
            optimizer.zero_grad()

            #forward pass through the network
            output = model(data)

            #calculate loss for this iteration
            loss = criterion(output, target)

            #back propagation
            loss.backward()

            #update the parameters (weights)
            optimizer.step()

            #update the training loss
            #train_loss += loss.data
            train_loss = train_loss + ((1 / (batch_num + 1)) * (loss.data - train_loss))

            if batch_num % 100 == 0:
                print(f'Batch number: {batch_num}; train loss: {train_loss} ')
```

```
                ######################
                # validate the model #
                ######################
                valid_loss = 0.0

                model.eval() #eval mode

                for batch_num, (data, target) in enumerate(loaders['valid']):
                    #move tensors to GPU
                    if use_cuda:
                        data, target = data.to(device), target.to(device)

                    output = model(data)

                    #Calculate the batch loss
                    loss = criterion(output, target)

                    #Update average validation loss
                    valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

                #print training/validation statistics
                print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(epoc

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

            return model
```

In [53]:
```
n_epochs = 1 #Since the network is pretrained, shouldn't need too many epochs.

# train the model
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,
```

```
Epoch: 1         Training Loss: 0.425963         Validation Loss: 0.004979
Validation loss decreased (inf --> 0.004979).  Saving model ...
```

In [58]:
```
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

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 [69]: if use_cuda:
             model_transfer = model_transfer.cuda()

         test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.796652


Test Accuracy: 76% (642/836)
```

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

         data_transfer = {
             'train': loaders_transfer['train'].dataset,
             'valid': loaders_transfer['valid'].dataset,
             'test': loaders_transfer['test'].dataset
         }

         # 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]
```

```
In [20]: print(class_names[:10])
```

```
['Affenpinscher', 'Afghan hound', 'Airedale terrier', 'Akita', 'Alaskan malamute', 'American esk
```

```
In [51]: from PIL import Image

         from torchvision import transforms, models

         #Loads and transforms an image
         def load_image(img_path):

             image = Image.open(img_path).convert('RGB')

             in_transform = transforms.Compose([
                                             transforms.Resize(size=(224, 224)),
```

```
                                                transforms.ToTensor(),
                                                transforms.Normalize((0.485, 0.456, 0.406),
                                                                     (0.229, 0.224, 0.225))])

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

            return image
```

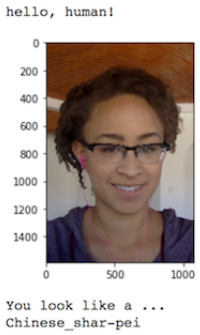
```
In [52]: def predict_breed_transfer(img_path):
            # load the image and return the predicted breed
            img = load_image(img_path)
            model_transfer.cpu()

            #Get predictions from the network
            model_transfer.eval()
            output = model_transfer(img)

            class_index = torch.argmax(output)
            predicted_breed = class_names[class_index]

            return predicted_breed
```

```
In [53]: model_transfer.cuda()

         for img_file in os.listdir('./images'):
             img_path = os.path.join('./images', img_file)
             predicted_breed = predict_breed_transfer(img_path)
             print(f'Image path: {img_path}; Predicted breed: {predicted_breed}')
```

```
Image path: ./images/American_water_spaniel_00648.jpg; Predicted breed: American water spaniel
Image path: ./images/Labrador_retriever_06449.jpg; Predicted breed: Labrador retriever
Image path: ./images/Labrador_retriever_06455.jpg; Predicted breed: Labrador retriever
Image path: ./images/Welsh_springer_spaniel_08203.jpg; Predicted breed: Irish red and white sett
Image path: ./images/sample_cnn.png; Predicted breed: Dachshund
Image path: ./images/Curly-coated_retriever_03896.jpg; Predicted breed: Curly-coated retriever
Image path: ./images/sample_dog_output.png; Predicted breed: Greyhound
Image path: ./images/Brittany_02625.jpg; Predicted breed: Brittany
Image path: ./images/sample_human_output.png; Predicted breed: Dachshund
Image path: ./images/Labrador_retriever_06457.jpg; Predicted breed: Labrador retriever
```

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.



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.

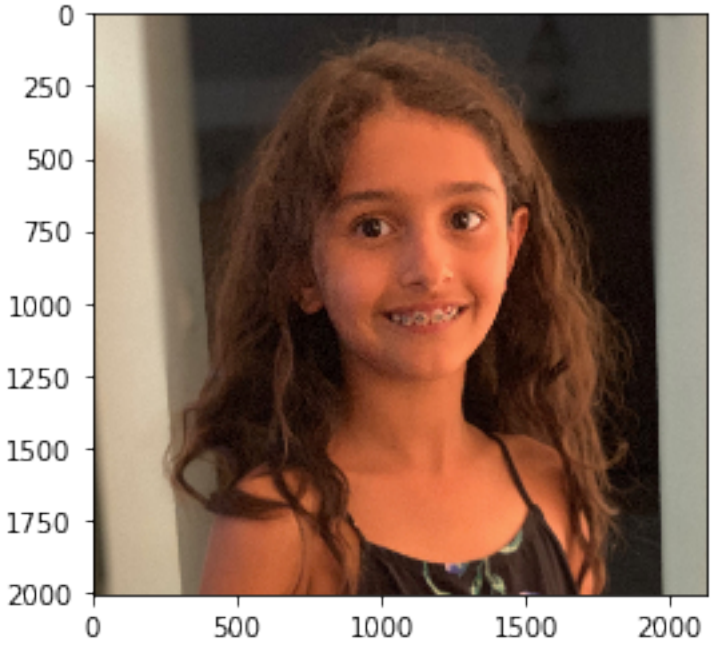
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [60]:
```
### 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)

    if dog_detector(img_path) is True:
        breed_prediction = predict_breed_transfer(img_path)
        print("\n\nDog detected")
        plt.show()
        print(f"What a fine {breed_prediction}!")
    elif face_detector(img_path) > 0:
        breed_prediction = predict_breed_transfer(img_path)
        print('\n\nHello, human!')
        plt.show()
        print(f"You look like a {breed_prediction}!")
    else:
        plt.show()
        print("\n\nCould not detect a face in the image")
```

In [72]:
```
images=[file for file in os.listdir('images') if file.endswith(('.jpg', '.png', '.jpeg'

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

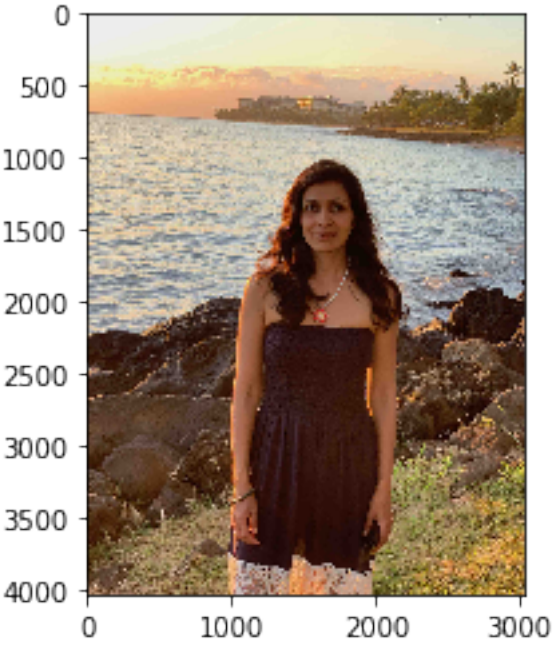
```
Hello, human!
```




```
You look like a Chinese crested!


Dog detected
```

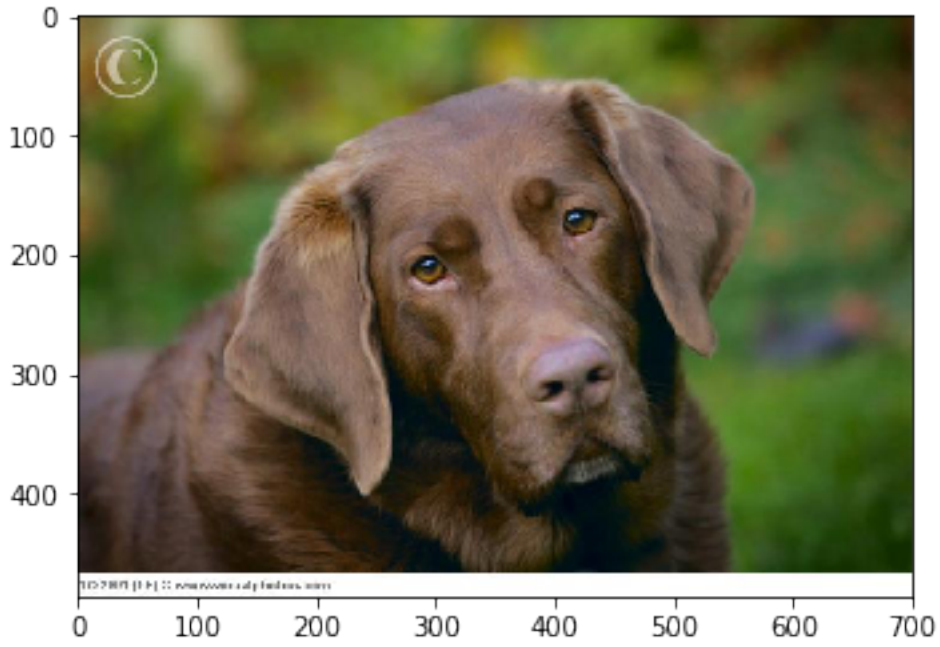

```
What a fine American water spaniel!
```

```
Hello, human!
```



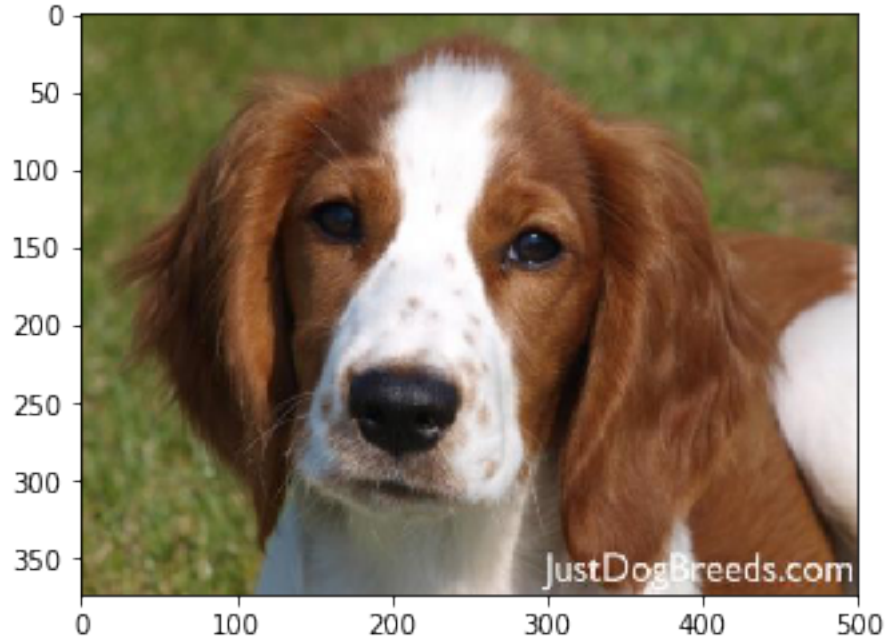
```
You look like a Bullmastiff!


Dog detected
```


```
What a fine Labrador retriever!


Dog detected
```

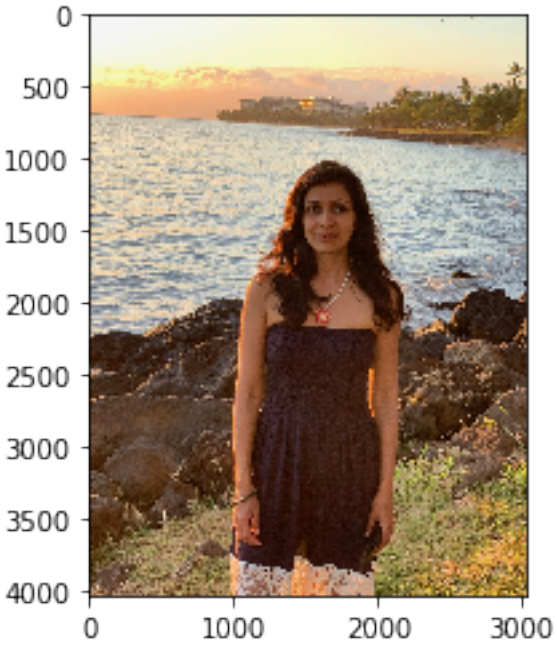


What a fine Labrador retriever!

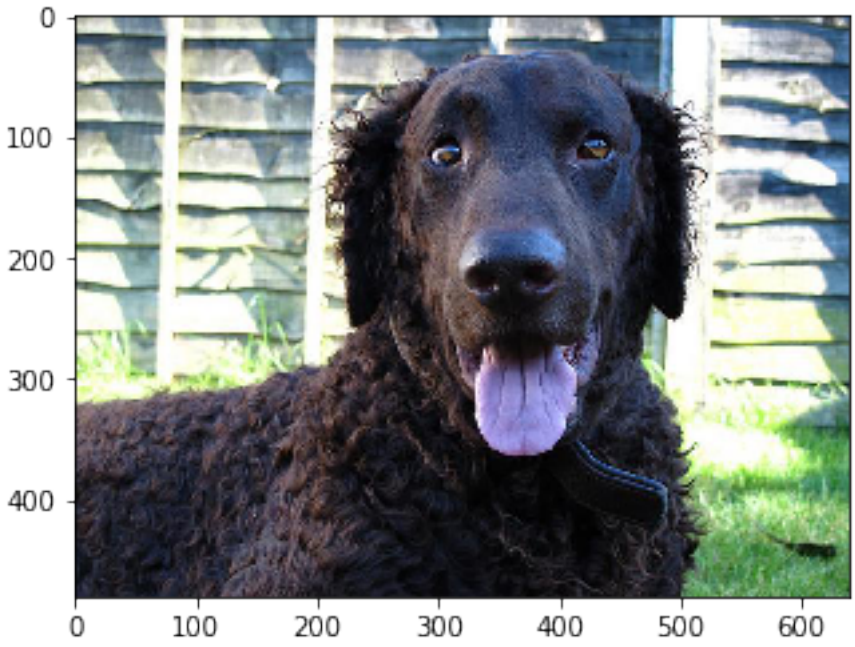
Dog detected



```
What a fine Irish red and white setter!

Could not detect a face in the image

Hello, human!
```

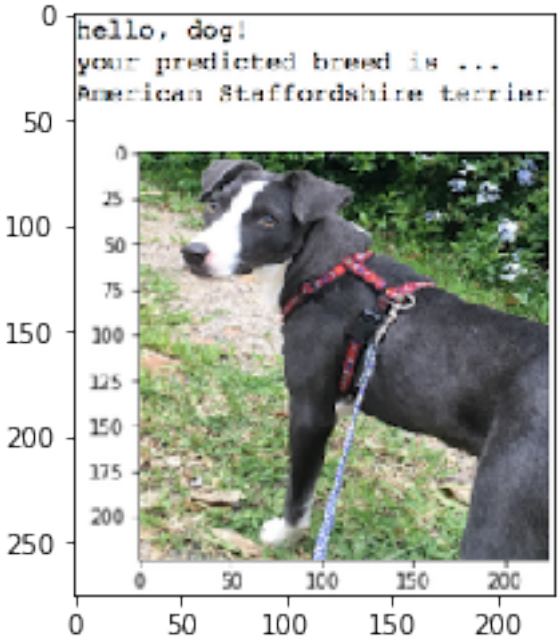



```
You look like a Bullmastiff!

Dog detected
```

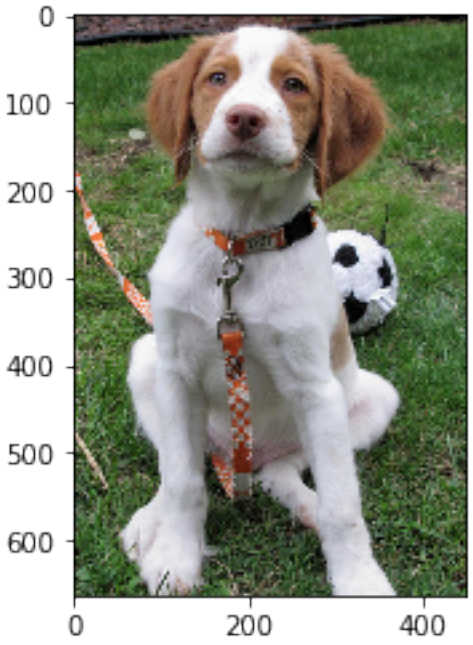



```
What a fine Curly-coated retriever!
```

```
Dog detected
```



```
What a fine Greyhound!

Dog detected
```

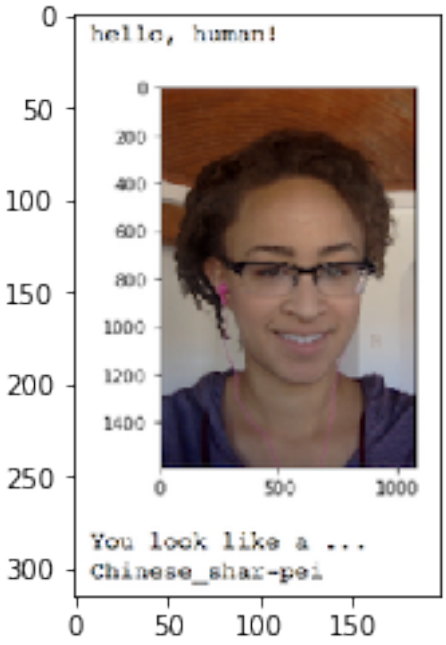



```
What a fine Brittany!

Hello, human!
```

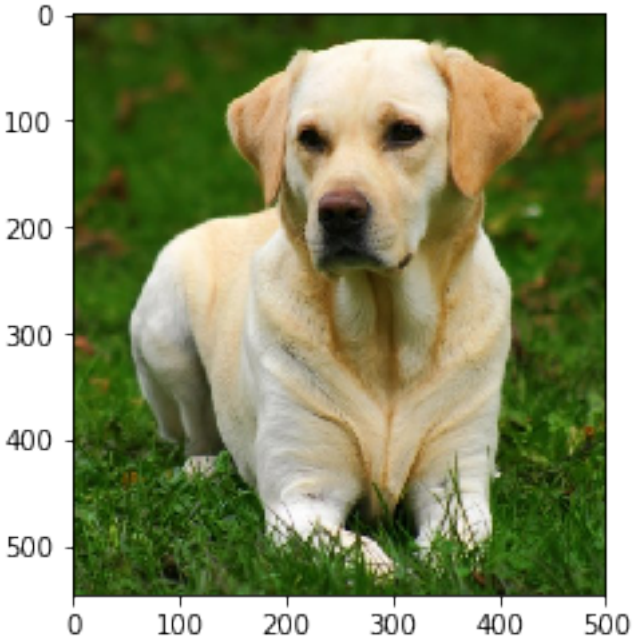

```
You look like a Dachshund!
```

```
Hello, human!
```

```
You look like a Dogue de bordeaux!


Hello, human!
```

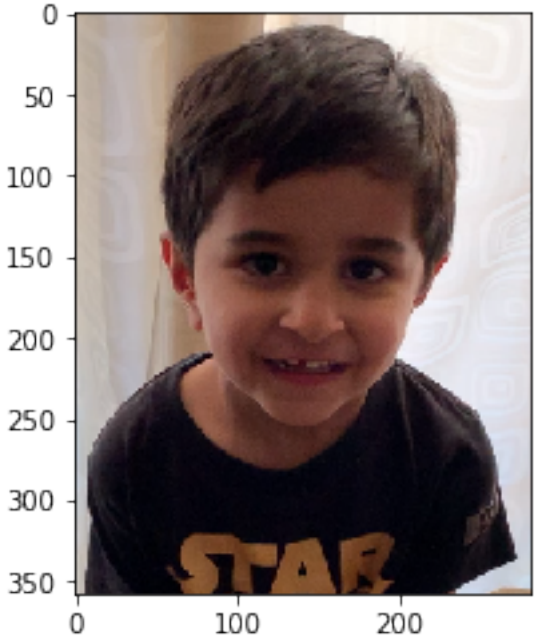



```
You look like a Dachshund!


Dog detected
```




```
What a fine Labrador retriever!
```

```
Hello, human!
```



```
You look like a Maltese!
```

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.

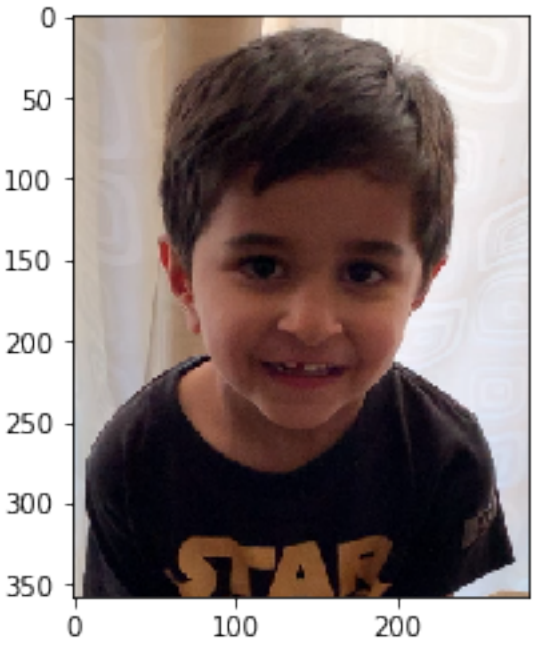
Answer: (Three possible points for improvement)

The output is better than I expected!

Three possible points of improvement: 1. Improve the accuracy with more training data. 2. Improve the accuracy of the model with introducing more transformations for image augmentation. Introduce more rotation and translation invariance in the training data. 3. If the training data is increased, maybe fine tune the parameters of the convolutional layer.

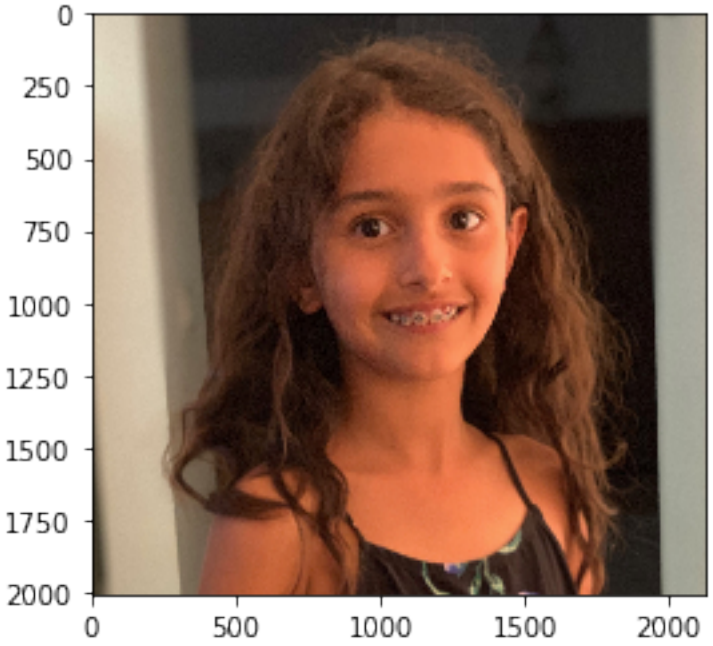
```
In [74]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.
         my_human_files = ['./images/boy.png', './images/IMG_0822.jpg',  './images/IMG_0509_1.jp

         my_dog_files = ['./images/saint-bernard.jpg', './images/pomeranian.jpg', './images/bass

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

```
Hello, human!
```

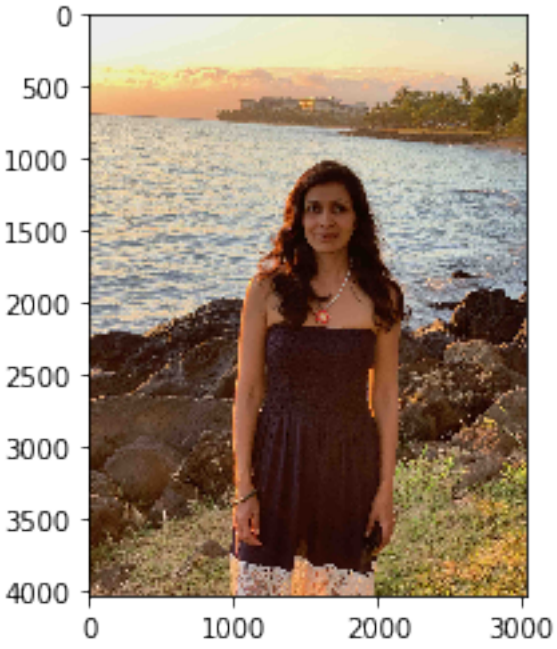



```
You look like a Maltese!
```

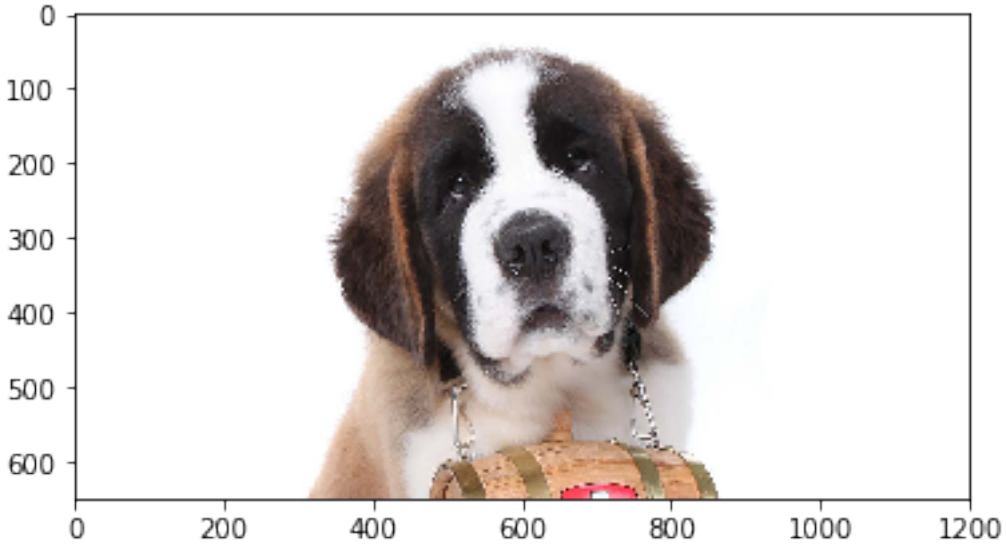
```
Hello, human!
```



```
You look like a Chinese crested!


Hello, human!
```

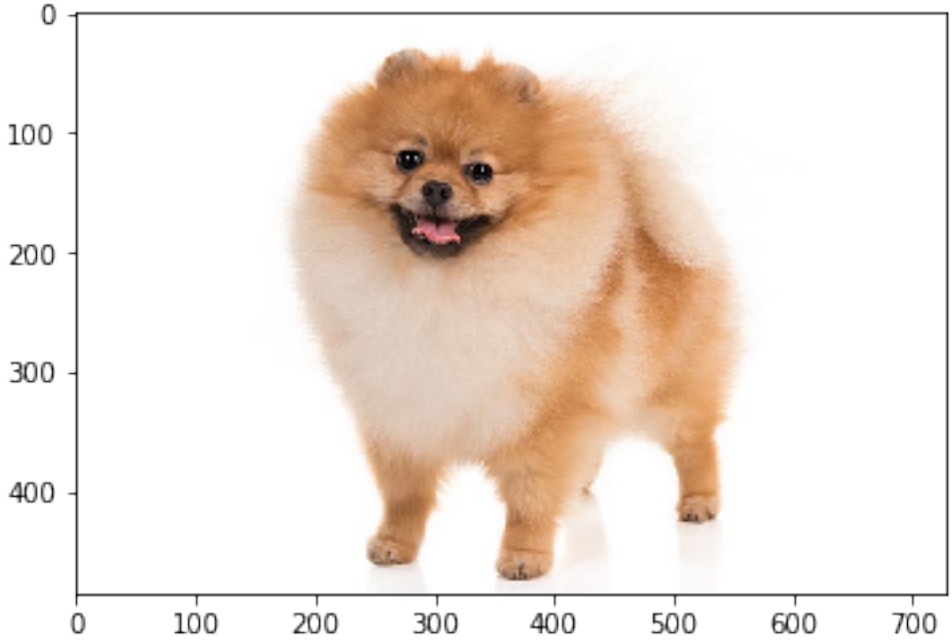



```
You look like a Bullmastiff!


Dog detected
```



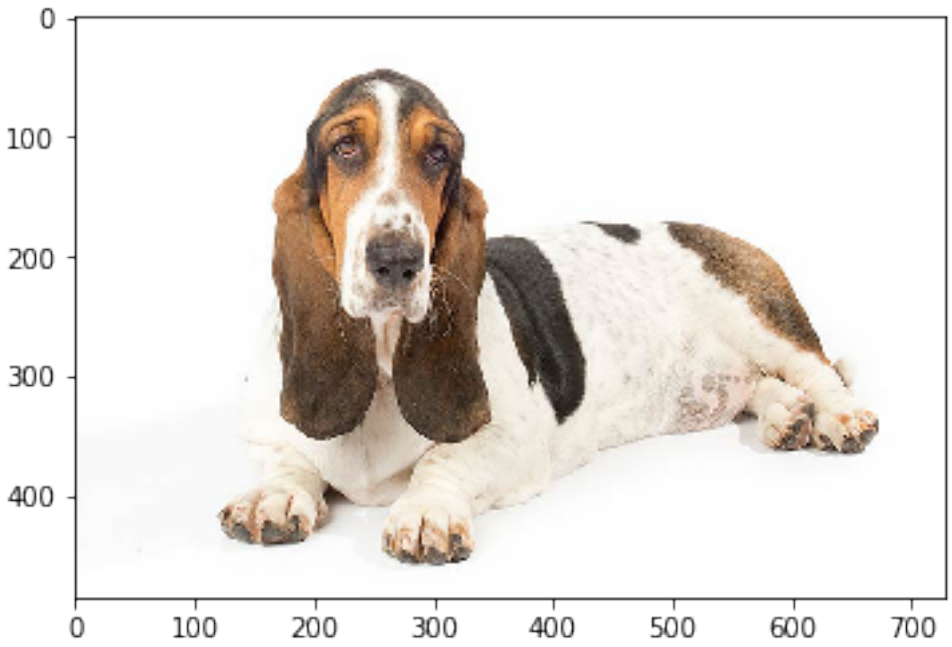

```
What a fine Saint bernard!
```

```
Dog detected
```



```
What a fine Pomeranian!
```

```
Dog detected
```




```
What a fine Basset hound!
```

In [ ]: