dog_app

April 27, 2021

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

```
In [1]: import numpy as np
        from glob import glob
        import cv2
        import os
        from PIL import Image
        import matplotlib.pyplot as plt
        %matplotlib inline
        from tqdm import tqdm
        import torch
        import torch.nn as nn
        import torch.nn.functional as F
        import torch.optim as optim
        from torch.utils.data import Dataset, DataLoader
        from torchvision.models import vgg16, resnet101
        from torchvision import datasets, transforms

        # 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

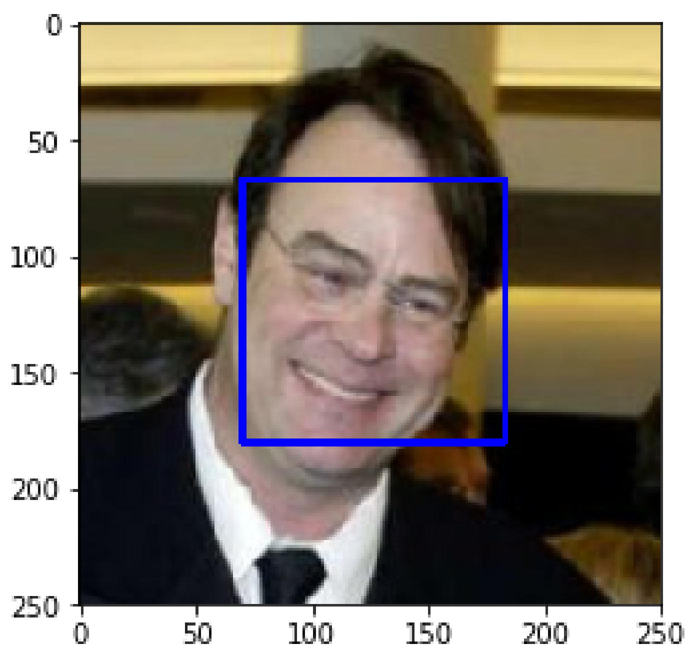
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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 [4]: # 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 [5]: 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.

        # humanDetection = tqdm(total=len(human_files_short), desc='Human Faces Detected in Huma
        # dogDetection = tqdm(total=len(dog_files_short), desc='Human Faces Detected in Dog File

        faceCount_HumanFiles = 0
```

```
faceCount_DogFiles = 0

for file in human_files_short:
    if face_detector(file):
        #humanDetection.update(1)
        faceCount_HumanFiles += 1

for file in dog_files_short:
    if face_detector(file):
        #dogDetection.update(1)
        faceCount_DogFiles += 1

print(f"Human Faces Detected in Human Files: {faceCount_HumanFiles / len(human_files_sho
print(f"Human Faces Detected in Dog Files: {faceCount_DogFiles / len(dog_files_short)*10
```

```
Human Faces Detected in Human Files: 98.0%
Human Faces Detected in Dog Files: 17.0%
```

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

In [6]:
```
### (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 [13]:
```
import torch
import torchvision.models as models

# define VGG16 model
VGG16 = models.vgg16(pretrained=True)

# check if CUDA is available
use_cuda = torch.cuda.is_available()
```

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

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

1.1.4 (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 [14]: 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
             data_transform = transforms.Compose([transforms.Resize(256),
                                                  transforms.CenterCrop(224),
                                                  transforms.ToTensor(),
                                                  transforms.Normalize(mean=[0.485, 0.456, 0.406
                                                                       std=[0.229, 0.224, 0.225]
             img = Image.open(img_path) # must use PIL library to read in image
             img = data_transform(img) # apply img resize and transform to tensor type
             img = img.unsqueeze(0) # add dimension for n_samples
             if use_cuda:
                 img = img.cuda()
             output = VGG16(img) # returns probability list for 1000 classes
             imageInd = int(torch.argmax(output)) # get index of max probability

             return imageInd  # predicted class index
```

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 [15]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.

             index = VGG16_predict(img_path)

             return (index >= 151 and index <= 268) # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [16]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         # humanDetection = tqdm(total=len(human_files_short), desc='Human Faces Detected in Dog
         # dogDetection = tqdm(total=len(dog_files_short), desc='Dog Faces Detected in Dog Detec

         humanFaceCount = 0
         dogFaceCount = 0

         for file in human_files_short:
             if dog_detector(file):
         #         humanDetection.update(1)
                 humanFaceCount += 1

         for file in dog_files_short:
             if dog_detector(file):
         #         dogDetection.update(1)
                 dogFaceCount += 1

         print(f"Dog Faces Detected in Dog Detector: {dogFaceCount / len(dog_files_short)*100}%"
         print(f"Human Faces Detected in Dog Detector: {humanFaceCount / len(human_files_short)*

Dog Faces Detected in Dog Detector: 100.0%
Human Faces Detected in Dog Detector: 0.0%
```

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 [17]: ### (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 [18]:
```
import os
from torchvision import datasets

### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
data_dir = '/data/dog_images'

# set data transform types
dataTransforms = {"train": transforms.Compose([transforms.RandomResizedCrop(224),
                                               transforms.RandomRotation(30),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.40
                                                                     [0.229, 0.224, 0.22

                  "valid": transforms.Compose([transforms.Resize(256),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.40
                                                                     [0.229, 0.224, 0.22

                  "test": transforms.Compose([transforms.Resize(256),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.485, 0.456, 0.406
                                                                     [0.229, 0.224, 0.22
# get data w/ transforms
dataSets = {set: datasets.ImageFolder(os.path.join(data_dir, set), transform=dataTransf
            for set in ["train", "valid", "test"]}

# load data with given batch size
dataLoaders = {set: DataLoader(dataSets[set], batch_size=32, shuffle=True) for set in [
```

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:

-For validation and testing data sets, I resize the image to 256x256 for uniformity and center crop of 224x224 from that to grab the most relevant image data and remove background noise.

-For training data, random 224x224 image cropping as well as random 30 degree rotations, and random horizontal flipping is performed.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [19]: import torch.nn as nn
         import torch.nn.functional as F

         # check if CUDA is available
         use_cuda = torch.cuda.is_available()

         # 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
                 # Input Images: 224x224
                 self.convLayer1 = nn.Conv2d(3, 16, 3, stride=1, padding=1)
                 self.convLayer2 = nn.Conv2d(16, 32, 3, stride=1, padding=1)
                 self.convLayer3 = nn.Conv2d(32, 64, 3, stride=1, padding=1)
                 self.convLayer4 = nn.Conv2d(64, 128, 3, stride=1, padding=1)

                 self.maxPool = nn.MaxPool2d(2,2)
                 self.dropout = nn.Dropout2d(0.25)

                 # 133 dog breeds to classify between
                 # 4 convolutional layers -> 4 pool layers: 224x224 -> 112x112 ->  56x56 -> 28x2
                 self.fullyConnected1 = nn.Linear(128*14*14, 512) # depth of 128 filters with a
                 self.fullyConnected2 = nn.Linear(512, 256)
                 self.fullyConnected3 = nn.Linear(256, 133)

             def forward(self, x):
                 ## Define forward behavior
                 x = self.maxPool(F.relu(self.convLayer1(x))) # 224x224 -> 128x128
                 x = self.maxPool(F.relu(self.convLayer2(x))) # 128x128 -> 56x56
                 x = self.maxPool(F.relu(self.convLayer3(x))) # 56x56 -> 28x28
                 x = self.maxPool(F.relu(self.convLayer4(x))) # 28x28 -> 14x14
                 x = x.view(-1, 128*14*14) # flatten to 1D vector
                 x = self.dropout(x) # define dropout for 1st fully connected layer
                 x = F.relu(self.fullyConnected1(x))
                 x = self.dropout(x) # define dropout for 2nd fully connected layer
                 x = F.relu(self.fullyConnected2(x))
                 x = self.fullyConnected3(x)

                 return x
```

```
#-#-# You so NOT have to modify the code below this line. #-#-#
use_cuda = True
# 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:

-I used 4 convolutional layers with a ReLu activation function, starting with 16 feature maps and doubling filter depth with each layer

-Max pooling was used between each convolutional layer to reduce image size and prevent overfitting

-After convolutional layers, the output was flattened into a 1D vector and passed into 3 fully connected layers

-Dropout was used to randomly deactivate nodes within each fully connected layer during training to prevent overfitting

-The training set consists of 133 dog breeds to classify between. Therefore, fully connected layers downsize the output to a final size of 133.

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 [20]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.Adam(model_scratch.parameters(), lr=0.001)
```

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 [23]: from PIL import ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True

         def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
             """returns trained model"""
             # initialize tracker for minimum validation loss
             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']):
        #print("batch: " + str(batch_idx))
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

        optimizer.zero_grad() # zero out gradients before starting training
        outputs = model(data) # get model predictions
        #print(outputs)
        _,predictions = torch.max(outputs, 1)
        loss = criterion(outputs,target)
        loss.backward()
        optimizer.step()

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


    ######################
    # 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
        ######HERE
        # with torch.no_grad():
        outputs = model(data)
        loss = criterion(outputs,target)
        valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss)) # average

    # 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:
                print(f"validation loss decreased from {valid_loss_min} to {valid_loss}")
                valid_loss_min = valid_loss
                print("saving trained model")
                torch.save(model.state_dict(), save_path)

        # return trained model
        return model


    # train the model
    model_scratch = train(10, dataLoaders, model_scratch, optimizer_scratch,
                          criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.121214         Validation Loss: 3.958159
validation loss decreased from inf to 3.9581592082977295
saving trained model
Epoch: 2         Training Loss: 4.075752         Validation Loss: 3.856663
validation loss decreased from 3.9581592082977295 to 3.856663465499878
saving trained model
Epoch: 3         Training Loss: 4.052187         Validation Loss: 3.790237
validation loss decreased from 3.856663465499878 to 3.7902369499206543
saving trained model
Epoch: 4         Training Loss: 3.989966         Validation Loss: 3.821360
Epoch: 5         Training Loss: 3.960958         Validation Loss: 3.842998
Epoch: 6         Training Loss: 3.933561         Validation Loss: 3.727980
validation loss decreased from 3.7902369499206543 to 3.727980375289917
saving trained model
Epoch: 7         Training Loss: 3.888061         Validation Loss: 3.853142
Epoch: 8         Training Loss: 3.885008         Validation Loss: 3.664903
validation loss decreased from 3.727980375289917 to 3.664902687072754
saving trained model
Epoch: 9         Training Loss: 3.808436         Validation Loss: 3.622870
validation loss decreased from 3.664902687072754 to 3.6228697299957275
saving trained model
Epoch: 10         Training Loss: 3.809955         Validation Loss: 3.654021
```

In [28]:
```
!pip install torchsummary
```

```
Collecting torchsummary
```

```
  Downloading https://files.pythonhosted.org/packages/7d/18/1474d06f721b86e6a9b9d7392ad68bed711a
Installing collected packages: torchsummary
Successfully installed torchsummary-1.5.1
```

```
In [29]: from torchsummary import summary
         summary(model_scratch, (3, 224, 224))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 16, 224, 224]             448
         MaxPool2d-2        [-1, 16, 112, 112]               0
            Conv2d-3        [-1, 32, 112, 112]           4,640
         MaxPool2d-4          [-1, 32, 56, 56]               0
            Conv2d-5          [-1, 64, 56, 56]          18,496
         MaxPool2d-6          [-1, 64, 28, 28]               0
            Conv2d-7         [-1, 128, 28, 28]          73,856
         MaxPool2d-8         [-1, 128, 14, 14]               0
         Dropout2d-9               [-1, 25088]               0
           Linear-10                 [-1, 512]      12,845,568
        Dropout2d-11                 [-1, 512]               0
           Linear-12                 [-1, 256]         131,328
           Linear-13                 [-1, 133]          34,181
================================================================
Total params: 13,108,517
Trainable params: 13,108,517
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 14.56
Params size (MB): 50.01
Estimated Total Size (MB): 65.14
----------------------------------------------------------------
```

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 [24]: 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(dataLoaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.703083


Test Accuracy: 13% (114/836)
```

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 [30]: "using same data loaders and transforms"
```

```
Out[30]: 'using same data loaders and transforms'
```

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

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

         # Freeze parameters of transfer model
         for parameter in model_transfer.parameters():
             parameter.requires_grad = False

         # Keep convolutional layers and replace fully connected layer with the required 133 out
         print(model_transfer.classifier[6].in_features)
         model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[6].in_features, 133)

         if use_cuda:
             model_transfer = model_transfer.cuda()

         print(model_transfer)
```

```
4096
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 used the vgg16 model that was trained on the ImageNet data set and achieved 92.7% accuracy in classifying images belonging to 1,000 different classes. As this dataset related to dogs is smaller and pretty similar to images that this network was already trained on, I determined that nearly all of the network weights could be kept as-is, and only the last bit of the fully connected layer would need to be trained more specifically on this dog breed dataset.

-To do this, I froze all the model parameters by turning off changes to gradients. However, I changed the last fully connected layer in the model to output 133 outputs which is the size of the dog breed classes used in this dataset. This turned on gradient changes only for the final fully connected layer, and the model was then trained specifically on this dog breed dataset to achieve more relevant and accurate results.

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 [32]: criterion_transfer = nn.CrossEntropyLoss()
         model_transfer_gradient_params = filter(lambda p: p.requires_grad, model_transfer.param
         optimizer_transfer = optim.SGD(model_transfer_gradient_params, lr=0.01)
```

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 [33]: n_epochs = 10

         # train the model
         model_transfer = train(n_epochs, dataLoaders, model_transfer, optimizer_transfer, crite

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

```
Epoch: 1         Training Loss: 3.270490         Validation Loss: 1.378800
validation loss decreased from inf to 1.3788001537322998
saving trained model
Epoch: 2         Training Loss: 2.052136         Validation Loss: 0.940526
validation loss decreased from 1.3788001537322998 to 0.9405256509780884
saving trained model
Epoch: 3         Training Loss: 1.755057         Validation Loss: 0.747723
validation loss decreased from 0.9405256509780884 to 0.7477234601974487
saving trained model
Epoch: 4         Training Loss: 1.622306         Validation Loss: 0.644539
validation loss decreased from 0.7477234601974487 to 0.644538938999176
saving trained model
Epoch: 5         Training Loss: 1.493649         Validation Loss: 0.614104
validation loss decreased from 0.644538938999176 to 0.6141043305397034
saving trained model
Epoch: 6         Training Loss: 1.431479         Validation Loss: 0.553629
validation loss decreased from 0.6141043305397034 to 0.5536288022994995
saving trained model
Epoch: 7         Training Loss: 1.418994         Validation Loss: 0.533381
validation loss decreased from 0.5536288022994995 to 0.533380925655365
saving trained model
Epoch: 8         Training Loss: 1.412342         Validation Loss: 0.513573
validation loss decreased from 0.533380925655365 to 0.5135729908943176
saving trained model
Epoch: 9         Training Loss: 1.366600         Validation Loss: 0.534014
Epoch: 10         Training Loss: 1.359814         Validation Loss: 0.482132
validation loss decreased from 0.5135729908943176 to 0.48213186860084534
saving trained model
```

1.1.16 (IMPLEMENTATION) Test the Model

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

```
In [34]: test(dataLoaders, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.527881
```

```
Test Accuracy: 85% (713/836)
```

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

         # list of class names by index, i.e. a name can be accessed like class_names[0]
         class_names = [item[4:].replace("_", " ") for item in dataSets['train'].classes]

         data_transform = dataTransforms['valid']

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             img = Image.open(img_path) # must use PIL library to read in image
             plt.imshow(img)
             plt.show()
             img = data_transform(img) # apply img resize and transform to tensor type
             img = img.unsqueeze(0) # add dimension for n_samples
             if use_cuda:
                 img = img.cuda()
             output = model_transfer(img) # returns probability list for 1000 classes
             imageInd = int(torch.argmax(output)) # get index of max probability
             return class_names[imageInd]
```

Step 5: Write your Algorithm

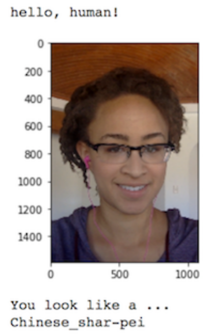
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is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

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [36]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             if dog_detector(img_path):
```



Sample Human Output

```
        print(f"This dog seems to be a {predict_breed_transfer(img_path)}")
    elif face_detector(img_path):
        print(f"I think this human looks like a {predict_breed_transfer(img_path)}")
    else:
        print("I actually don't know what this is...")
```

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 dog breed detection of actual dog images is spot on, more accurate than I had expected. After viewing images of the dog breeds that it thought the human images looked like, I could see how certain features like hair style could lead the network to coming up with that result.

3 Possible Points of Improvement:

-Different transfer models could be used to determine the most accurate predictor for this particular dog breed classification problem

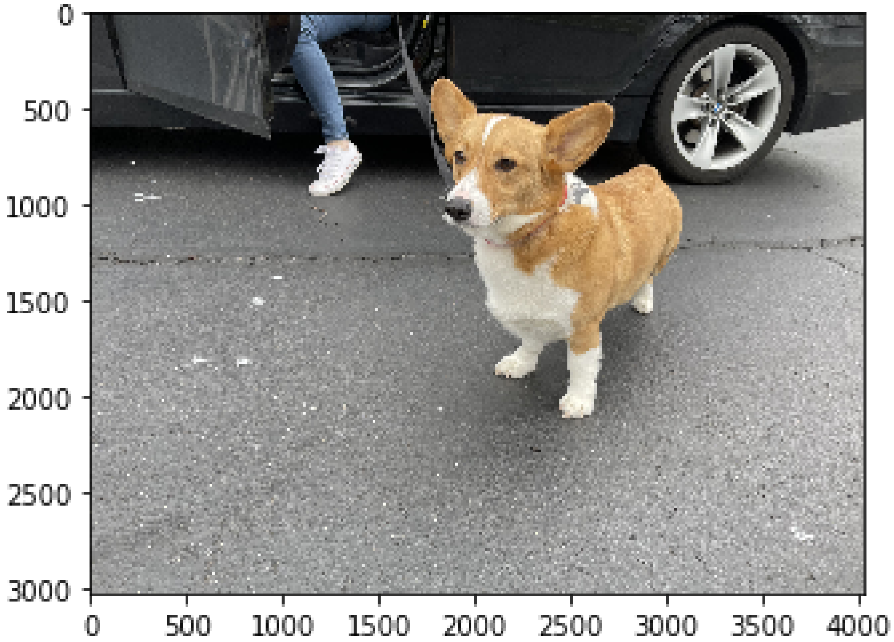
-A more extensive data set with a bigger variety dog images for each breed could help increase accuracy

-Hyperparameters like initial weights or learn rate could be further tuned

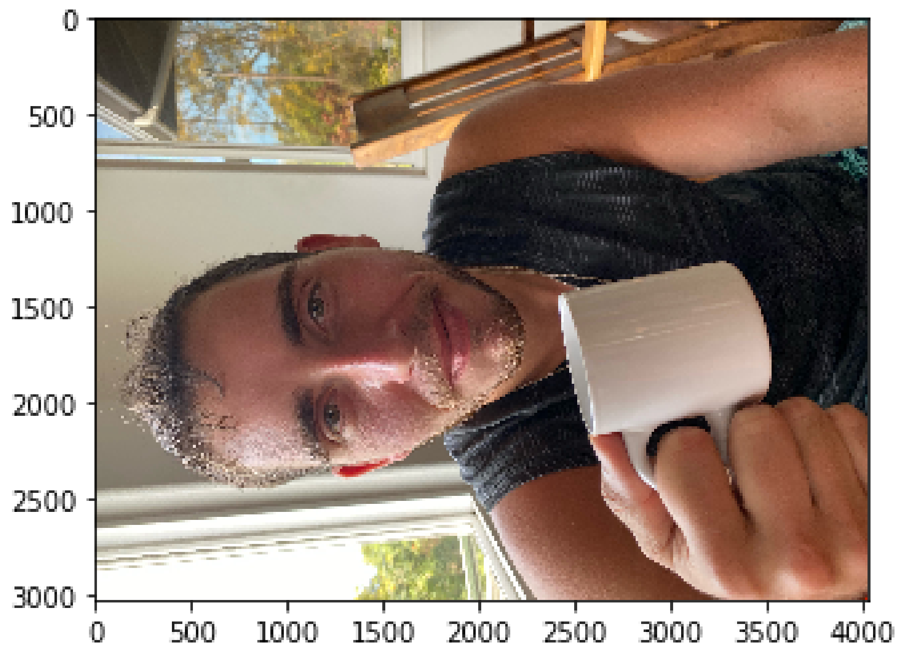
-The transfer model could be trained on more epochs, and a learn rate scheduler could be used to determine optimal parameters to minimize training loss

```
In [37]: ## 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.
```

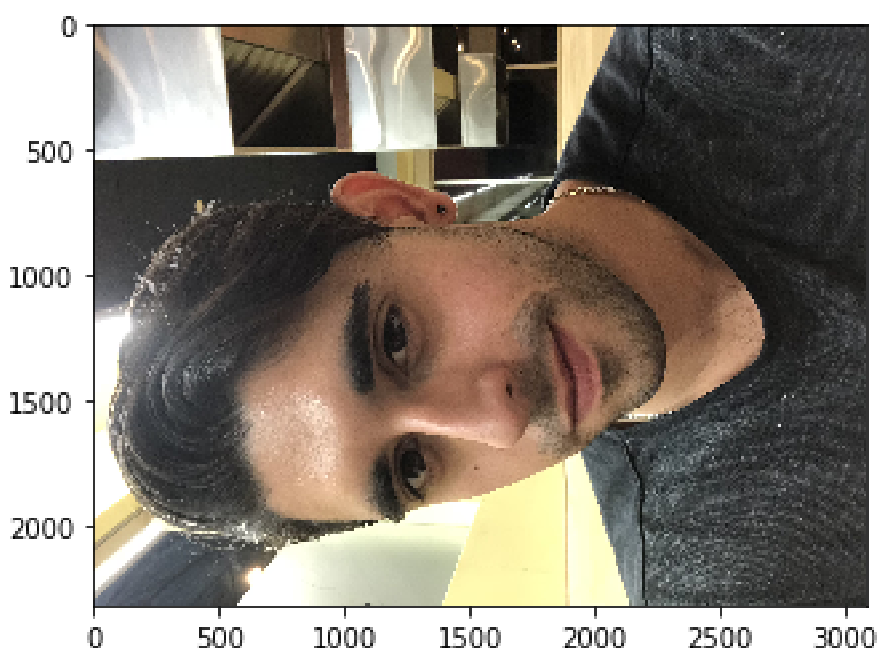
```
import glob
for filename in glob.glob('Test_Images/*.jpeg'): #assuming jpg
    run_app(filename)

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



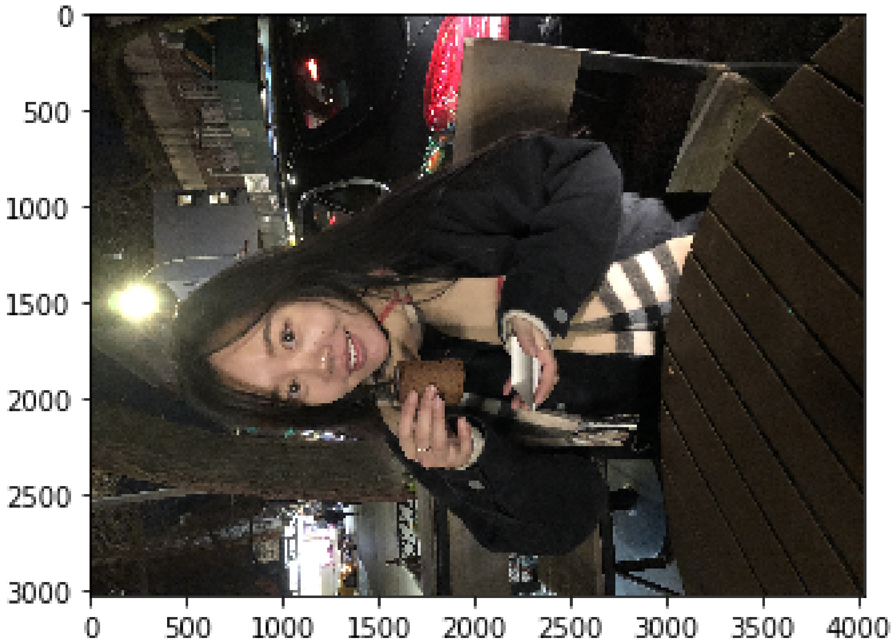

```
This dog seems to be a Pembroke welsh corgi
```



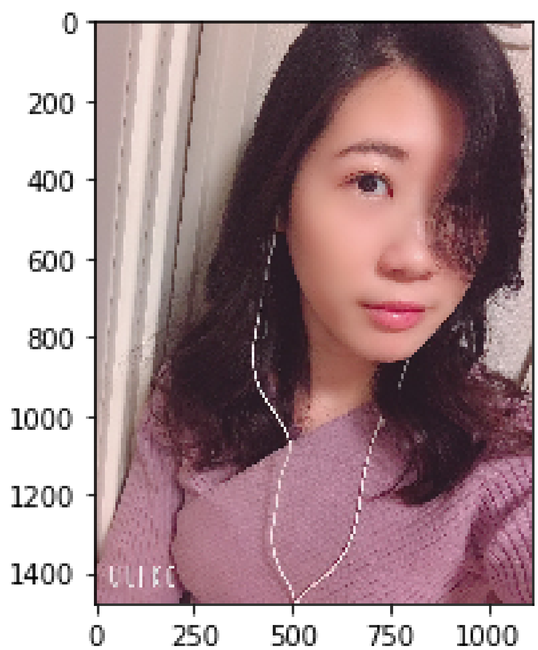

```
I think this human looks like a Pharaoh hound
I actually don't know what this is...
```



I think this human looks like a Italian greyhound



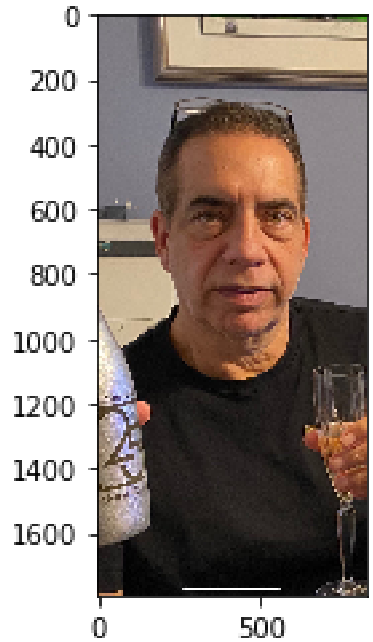
I think this human looks like a Neapolitan mastiff



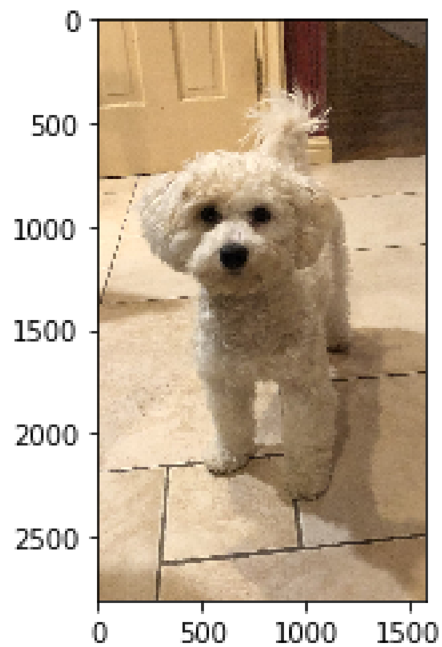
```
I think this human looks like a Boykin spaniel
```


```
This dog seems to be a Pomeranian
```

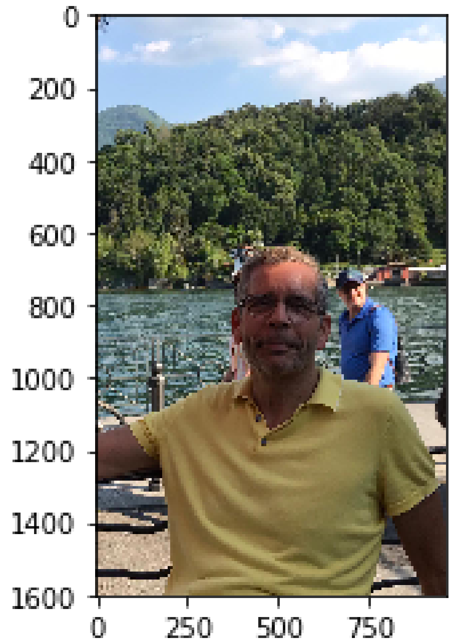
```
I think this human looks like a Dalmatian
```



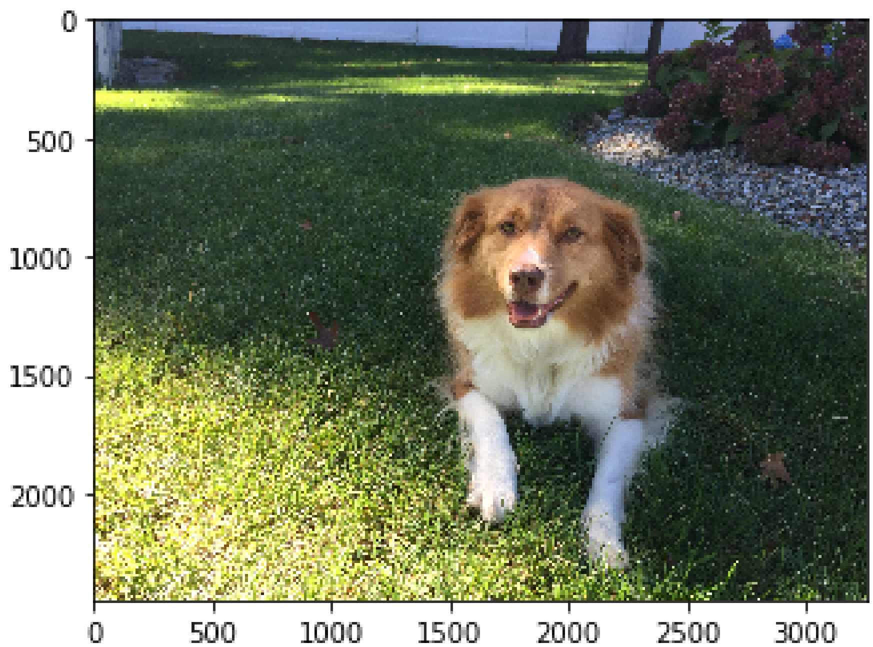

```
I think this human looks like a Chinese crested
```



```
This dog seems to be a Bichon frise
```

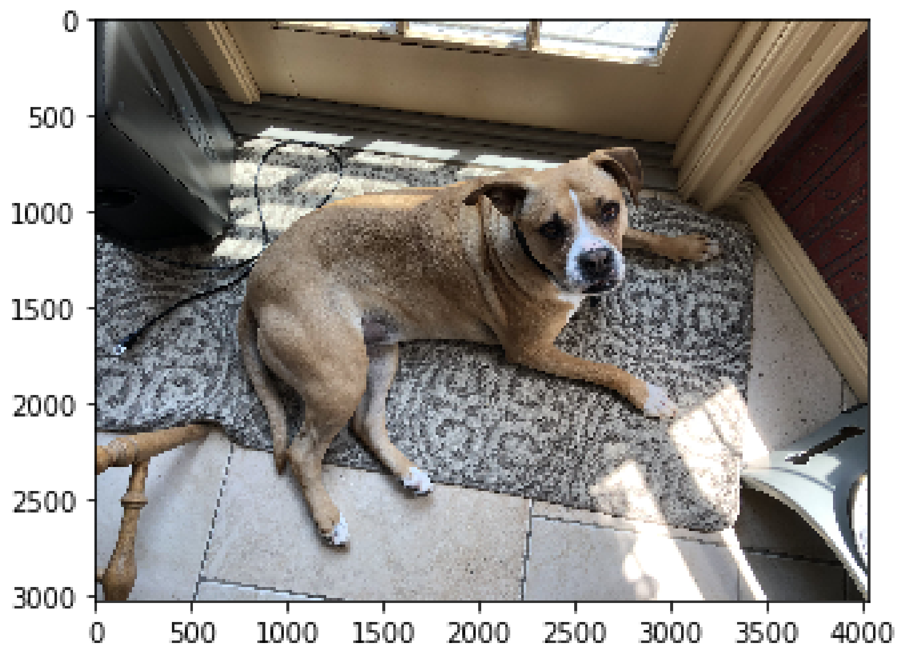



```
I think this human looks like a Pharaoh hound
```



```
This dog seems to be a Nova scotia duck tolling retriever
```


```
This dog seems to be a Greyhound
```



```
This dog seems to be a Boxer
```

```
In [ ]:
```