dog_app

October 8, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

## Step 0: Import Datasets

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

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

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

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

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

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

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

        # print number of images in each dataset

        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

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

## Step 1: Detect Humans

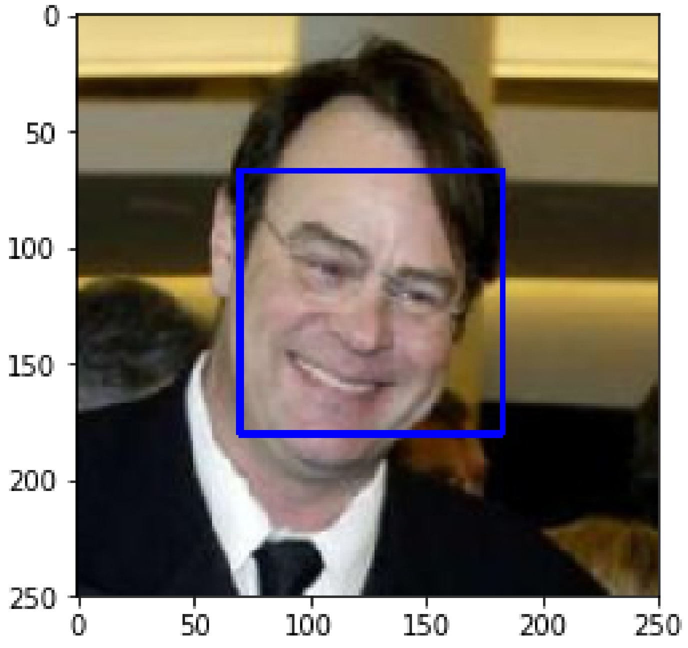
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

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

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

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

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

        # print number of faces detected in the image
```

```
print('Number of faces detected:', len(faces))

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

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

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

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

        ## TODO: Test the performance of the face_detector algorithm
        human_face_detection_cnt=0
        dog_face_detection_cnt=0
        for i in range(100):
            human_face_detection_cnt+= 1 if face_detector(human_files_short[i]) else 0
            dog_face_detection_cnt+= 1 if face_detector(dog_files_short[i]) else 0

        print("Accuracy of Human images with face Detected ",human_face_detection_cnt)
        print("Accuracy of Dog images with face Detected ",dog_face_detection_cnt)
        ## on the images in human_files_short and dog_files_short.
```

```
Accuracy of Human images with face Detected  98
Accuracy of Dog images with face Detected  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 [5]: ### (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 [6]: import torch
        import torchvision.models as models

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:32<00:00, 17250207.33it/s]
```

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

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

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

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

```
In [7]: from PIL import Image
        import torchvision.transforms as transforms


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

            Args:
                img_path: path to an image

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

            ## TODO: Complete the function.
            ## Load and pre-process an image from the given img_path
            data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                                    transforms.CenterCrop(224),
                                                 transforms.ToTensor(),
                                                 transforms.Normalize(mean=[0.485, 0.456, 0.406],st
            #image=datasets.ImageFolder(img_path, transform=data_transform)
            image=Image.open(img_path).convert('RGB')
            image = data_transform(image)
            image= image.unsqueeze_(0)
            if use_cuda:
                image = image.cuda()

            ## Return the *index* of the predicted class for that image
            output = VGG16(image)
            _, pred = torch.max(output, 1)
            #print("Output",pred)
            return pred # predicted class index

        #VGG16_predict(dog_files_short[0])
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
```

```
def dog_detector(img_path):
    ## TODO: Complete the function.
    predicted_dog_class=VGG16_predict(img_path)

    return True if ((151<=predicted_dog_class) and (predicted_dog_class<=268)) else Fals
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: - Percentage of images in human_files_short have detected dog 2 - percentage of the images in dog_files_short have a detected dog 100

```
In [20]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_face_detected_dog_cnt=0
         correct_dog_face_detected_cnt=0

         for i in tqdm(range(100)):
             human_face_detected_dog_cnt+= 1 if dog_detector(human_files_short[i]) else 0
             correct_dog_face_detected_cnt+= 1 if dog_detector(dog_files_short[i]) else 0

         print("Percentage of images in human_files_short have detected dog  ",human_face_detect
         print("percentage of the images in dog_files_short have a detected dog ",correct_dog_fa
```

```
100%|| 100/100 [00:07<00:00, 14.45it/s]

Percentage of images in human_files_short have detected dog   2
percentage of the images in dog_files_short have a detected dog  100
```

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 [10]: ### (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 [11]: import numpy as np
         from glob import glob
         from PIL import Image, ImageFile
         ImageFile.LOAD_TRUNCATED_IMAGES = True
         import torchvision.transforms as transforms
         import torch
         import torchvision.models as models

         import os
```

```
from torchvision import datasets
from torch.utils.data.sampler import SubsetRandomSampler


### TODO: Write data loaders for training, validation, and test sets
## Specify appropriate transforms, and batch_sizes
use_cuda = torch.cuda.is_available()
num_workers= 0
batch_size= 20
valid_size= 0.2

data_transform={}
data_transform['train'] = transforms.Compose([transforms.RandomResizedCrop(224),
                                             transforms.CenterCrop(224),
                                    transforms.RandomHorizontalFlip(),
                                    transforms.RandomRotation(10),
                                    transforms.ToTensor(),
                                   transforms.Normalize(mean=[0.485, 0.456, 0.406],std
data_transform['valid'] = transforms.Compose([transforms.RandomResizedCrop(224),
                                             transforms.CenterCrop(224),
                                             transforms.ToTensor(),
                                             transforms.Normalize(mean=[0.485, 0.456,
data_transform['test'] = transforms.Compose([transforms.RandomResizedCrop(224),
                                            transforms.CenterCrop(224),
                                            transforms.ToTensor(),
                                            transforms.Normalize(mean=[0.485, 0.456, 0

data_scratch={}
data_scratch['train'] = datasets.ImageFolder('/data/dog_images/train', transform=data_t
data_scratch['valid'] = datasets.ImageFolder('/data/dog_images/test', transform=data_tr
data_scratch['test']= datasets.ImageFolder('/data/dog_images/valid', transform=data_tra

n_classes=len(data_scratch['train'].classes)

loaders_scratch={}
# prepare data loaders (combine dataset and sampler)
loaders_scratch['train']= torch.utils.data.DataLoader(data_scratch['train'], batch_size

loaders_scratch['valid'] = torch.utils.data.DataLoader(data_scratch['valid'], batch_siz

loaders_scratch['test'] = torch.utils.data.DataLoader(data_scratch['test'], batch_size=
    num_workers=num_workers)
```

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: 1. I resized the images to size 224x224 to keep all the images of same size thus avoiding bias. Also, if the image is very large, resizing it to small size will save the memory 2. Yes, I decided to augment the data set. I have used `transforms.RandomHorizontalFlip()` for the model to be translation invariant, and `transforms.RandomRotation(10)` for the model to be rotation invariant.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [12]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
         class Net(nn.Module):
             ### TODO: choose an architecture, and complete the class
             def __init__(self):
                 super(Net, self).__init__()
                 self.conv1=nn.Conv2d(3,16,3, padding=1)
                 self.conv2 = nn.Conv2d(16, 32, 3, padding=1)
                 # convolutional layer (sees 8x8x32 tensor)
                 self.conv3 = nn.Conv2d(32, 64, 3, padding=1)
                 # max pooling layer
                 self.pool = nn.MaxPool2d(2, 2)
                 # linear layer (64 * 28 * 28 -> 500)
                 self.fc1 = nn.Linear(64*28*28, 500)
                 # linear layer (500 -> 10)
                 self.fc2 = nn.Linear(500, 133)
                 # dropout layer (p=0.25)
                 self.dropout = nn.Dropout(0.25)
                 ## Define layers of a CNN

             def forward(self, x):
                 ## Define forward behavior
                 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 image input
                 x = x.view(-1, 64*28*28)
                 #x=x.view(x.size(0), -1)
                 # add dropout layer
                 x = self.dropout(x)
                 # add 1st hidden layer, with relu activation function
                 x = F.relu(self.fc1(x))
                 # add dropout layer
                 x = self.dropout(x)
                 # add 2nd hidden layer, with relu activation function
                 x = self.fc2(x)
```

```
                return x

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

        # instantiate the CNN
        model_scratch = Net()

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

```
In [13]: print(model_scratch)
```

```
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=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

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

- As we have dataset of RGB images the depth of input image is 3. Then I have resized the image to 224x224. So, the input is of the shape 3x224x224
- The desired output is of the length of number of classes in the training dataset, which in this case is 133.
- The model_scratch has 3 convolutional layers, with filter of size 3x3 and padding=1 to cover the edges of the image
  - 1st convolutional layer - sees 32x32x3 image tensor
  - 2nd convolutional layer - sees 16x16x16 image tensor
  - 3rd convolutional layer - sees 8x8x32 image tensor Thus the algorithm takes input layer and gradually makes it much deeper. ReLu Activation function is applied to the layers
  - 1 max pooling layer with kernel_size= 2 and stride=2 was used to reduce the dimension of the image.
  - After max pooling, the images are flattend as we need to add linear layers at the end of the model, which accepts only 1D vectors.
  - 2 fully connected layers were added and a dropout to avoid overfitting.
    * first linear layer-(in_features= 64x28x28, out_features= 500)
    * second linear layer- (in_features= 500, output=133(number of classes))

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 [14]: 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 [15]: 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()
                     optimizer.zero_grad()
                     # forward pass: compute predicted outputs by passing inputs to the model
                     output = model(data)
                     # calculate the batch loss
                     loss = criterion(output, target)
                     # backward pass: compute gradient of the loss with respect to model paramet
                     loss.backward()
                     # perform a single optimization step (parameter update)
                     optimizer.step()
                     # update training loss
                     train_loss += loss.item()*data.size(0)
                     ## 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

            ######################
            # 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)
                # calculate the batch loss
                loss = criterion(output, target)
                # update average validation loss
                valid_loss += loss.item()*data.size(0)

            # calculate average losses
            train_loss = train_loss/len(loaders['train'].dataset)
            valid_loss = valid_loss/len(loaders['valid'].dataset)



            # 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('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 trained model
        return model
```

```
In [16]: # train the model
         model_scratch = train(50, loaders_scratch, model_scratch, optimizer_scratch,
                               criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.882032         Validation Loss: 4.855464
```

```
Validation loss decreased (inf --> 4.855464).  Saving model ...
Epoch: 2        Training Loss: 4.837151        Validation Loss: 4.783450
Validation loss decreased (4.855464 --> 4.783450).  Saving model ...
Epoch: 3        Training Loss: 4.741753        Validation Loss: 4.628017
Validation loss decreased (4.783450 --> 4.628017).  Saving model ...
Epoch: 4        Training Loss: 4.632483        Validation Loss: 4.584303
Validation loss decreased (4.628017 --> 4.584303).  Saving model ...
Epoch: 5        Training Loss: 4.581407        Validation Loss: 4.571840
Validation loss decreased (4.584303 --> 4.571840).  Saving model ...
Epoch: 6        Training Loss: 4.560260        Validation Loss: 4.555111
Validation loss decreased (4.571840 --> 4.555111).  Saving model ...
Epoch: 7        Training Loss: 4.518067        Validation Loss: 4.506117
Validation loss decreased (4.555111 --> 4.506117).  Saving model ...
Epoch: 8        Training Loss: 4.481464        Validation Loss: 4.453054
Validation loss decreased (4.506117 --> 4.453054).  Saving model ...
Epoch: 9        Training Loss: 4.451271        Validation Loss: 4.433606
Validation loss decreased (4.453054 --> 4.433606).  Saving model ...
Epoch: 10        Training Loss: 4.416217        Validation Loss: 4.454342
Epoch: 11        Training Loss: 4.401521        Validation Loss: 4.382602
Validation loss decreased (4.433606 --> 4.382602).  Saving model ...
Epoch: 12        Training Loss: 4.361248        Validation Loss: 4.393595
Epoch: 13        Training Loss: 4.343355        Validation Loss: 4.399085
Epoch: 14        Training Loss: 4.299307        Validation Loss: 4.333721
Validation loss decreased (4.382602 --> 4.333721).  Saving model ...
Epoch: 15        Training Loss: 4.277079        Validation Loss: 4.336075
Epoch: 16        Training Loss: 4.252675        Validation Loss: 4.305803
Validation loss decreased (4.333721 --> 4.305803).  Saving model ...
Epoch: 17        Training Loss: 4.232349        Validation Loss: 4.241857
Validation loss decreased (4.305803 --> 4.241857).  Saving model ...
Epoch: 18        Training Loss: 4.200402        Validation Loss: 4.264674
Epoch: 19        Training Loss: 4.167755        Validation Loss: 4.229602
Validation loss decreased (4.241857 --> 4.229602).  Saving model ...
Epoch: 20        Training Loss: 4.151293        Validation Loss: 4.164878
Validation loss decreased (4.229602 --> 4.164878).  Saving model ...
Epoch: 21        Training Loss: 4.116788        Validation Loss: 4.265063
Epoch: 22        Training Loss: 4.079938        Validation Loss: 4.240484
Epoch: 23        Training Loss: 4.049699        Validation Loss: 4.234517
Epoch: 24        Training Loss: 4.039917        Validation Loss: 4.153569
Validation loss decreased (4.164878 --> 4.153569).  Saving model ...
Epoch: 25        Training Loss: 4.013947        Validation Loss: 4.113070
Validation loss decreased (4.153569 --> 4.113070).  Saving model ...
Epoch: 26        Training Loss: 3.978993        Validation Loss: 4.124740
Epoch: 27        Training Loss: 3.953799        Validation Loss: 4.119269
Epoch: 28        Training Loss: 3.930572        Validation Loss: 4.106219
Validation loss decreased (4.113070 --> 4.106219).  Saving model ...
Epoch: 29        Training Loss: 3.904664        Validation Loss: 4.069811
Validation loss decreased (4.106219 --> 4.069811).  Saving model ...
Epoch: 30        Training Loss: 3.859413        Validation Loss: 4.100129
```

```
Epoch: 31         Training Loss: 3.870799        Validation Loss: 4.027703
Validation loss decreased (4.069811 --> 4.027703).  Saving model ...
Epoch: 32         Training Loss: 3.807363        Validation Loss: 4.070466
Epoch: 33         Training Loss: 3.799197        Validation Loss: 4.058980
Epoch: 34         Training Loss: 3.775463        Validation Loss: 4.049657
Epoch: 35         Training Loss: 3.753641        Validation Loss: 3.973460
Validation loss decreased (4.027703 --> 3.973460).  Saving model ...
Epoch: 36         Training Loss: 3.688764        Validation Loss: 4.087886
Epoch: 37         Training Loss: 3.705116        Validation Loss: 3.985007
Epoch: 38         Training Loss: 3.691021        Validation Loss: 4.183400
Epoch: 39         Training Loss: 3.641774        Validation Loss: 3.996159
Epoch: 40         Training Loss: 3.614490        Validation Loss: 4.023352
Epoch: 41         Training Loss: 3.636880        Validation Loss: 4.027124
Epoch: 42         Training Loss: 3.567318        Validation Loss: 4.012509
Epoch: 43         Training Loss: 3.583724        Validation Loss: 3.993360
Epoch: 44         Training Loss: 3.552387        Validation Loss: 4.035645
Epoch: 45         Training Loss: 3.524982        Validation Loss: 3.972685
Validation loss decreased (3.973460 --> 3.972685).  Saving model ...
Epoch: 46         Training Loss: 3.519021        Validation Loss: 3.948129
Validation loss decreased (3.972685 --> 3.948129).  Saving model ...
Epoch: 47         Training Loss: 3.452836        Validation Loss: 3.947002
Validation loss decreased (3.948129 --> 3.947002).  Saving model ...
Epoch: 48         Training Loss: 3.481260        Validation Loss: 3.978858
Epoch: 49         Training Loss: 3.442459        Validation Loss: 3.992816
Epoch: 50         Training Loss: 3.409627        Validation Loss: 4.030060
```

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 [17]: 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)

        test_loss = test_loss/len(loaders['test'].dataset)
        print('Test Loss: {:.6f}\n'.format(test_loss))



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

```
In [18]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 0.004782


Test Accuracy: 11% (98/835)
```

## 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 [35]: ## TODO: Specify data loaders
         loaders_transfer=loaders_scratch
```

```
In [36]: print(VGG16)
         print(VGG16.classifier[6].in_features)
         print(VGG16.classifier[6].out_features)
```

```
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=1000, bias=True)
  )
)
4096
1000
```

```
In [37]: # Freeze training for all "features" layers
         for param in VGG16.features.parameters():
             param.requires_grad = False
```

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

         ## TODO: Specify model architecture

         n_inputs = VGG16.classifier[6].in_features

         # add last linear layer (n_inputs -> 5 flower classes)
         # new layers automatically have requires_grad = True
         last_layer = nn.Linear(n_inputs, n_classes)

         VGG16.classifier[6] = last_layer
         model_transfer=VGG16

         print(model_transfer.classifier[6].out_features)

         # if GPU is available, move the model to GPU

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

```
133
```

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: - Transfer learning is a very efficient way of training models. I have used VGG16 pretrained model to train my model. Reason behind using VGG is that it has learned to distinguish between 1000 categories that are present in ImageNet dataset of which most of the categories are of animals, thus it best suits my project of dog breed classification. - We know that convolutional filters in the trained CNN are arranged in a kind of hierarchy, thus filters in the final layers are much specific. So, I have removed final layers of the network keeping the earlier layers by freezing them. - At the end I added a linear layers more specific to the project with out_features=133.

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 [39]: 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 [34]: # train the model
         model_transfer = train(40, loaders_transfer, model_transfer, optimizer_transfer, criter

         # 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: 0.991454        Validation Loss: 0.846310
Validation loss decreased (inf --> 0.846310).  Saving model ...
Epoch: 2         Training Loss: 0.959642        Validation Loss: 0.854284
Epoch: 3         Training Loss: 0.989277        Validation Loss: 0.857747
Epoch: 4         Training Loss: 0.949668        Validation Loss: 0.827528
Validation loss decreased (0.846310 --> 0.827528).  Saving model ...
Epoch: 5         Training Loss: 0.953344        Validation Loss: 0.909743
Epoch: 6         Training Loss: 0.975794        Validation Loss: 0.905414
Epoch: 7         Training Loss: 0.921718        Validation Loss: 0.841829
Epoch: 8         Training Loss: 0.950349        Validation Loss: 0.897987
Epoch: 9         Training Loss: 0.904691        Validation Loss: 0.841230
Epoch: 10         Training Loss: 0.912906        Validation Loss: 0.866729
Epoch: 11         Training Loss: 0.921879        Validation Loss: 0.813947
Validation loss decreased (0.827528 --> 0.813947).  Saving model ...
Epoch: 12         Training Loss: 0.895667        Validation Loss: 0.854545
Epoch: 13         Training Loss: 0.884975        Validation Loss: 0.794241
Validation loss decreased (0.813947 --> 0.794241).  Saving model ...
Epoch: 14         Training Loss: 0.916118        Validation Loss: 0.872169
Epoch: 15         Training Loss: 0.852115        Validation Loss: 0.869968
Epoch: 16         Training Loss: 0.874707        Validation Loss: 0.809286
Epoch: 17         Training Loss: 0.853899        Validation Loss: 0.813196
Epoch: 18         Training Loss: 0.873828        Validation Loss: 0.845228
Epoch: 19         Training Loss: 0.840503        Validation Loss: 0.741636
Validation loss decreased (0.794241 --> 0.741636).  Saving model ...
Epoch: 20         Training Loss: 0.872990        Validation Loss: 0.839961
Epoch: 21         Training Loss: 0.847500        Validation Loss: 0.800028
Epoch: 22         Training Loss: 0.829209        Validation Loss: 0.842830
Epoch: 23         Training Loss: 0.866158        Validation Loss: 0.847520
Epoch: 24         Training Loss: 0.866480        Validation Loss: 0.801686
Epoch: 25         Training Loss: 0.839776        Validation Loss: 0.831294
Epoch: 26         Training Loss: 0.857950        Validation Loss: 0.821162
Epoch: 27         Training Loss: 0.823474        Validation Loss: 0.811624
Epoch: 28         Training Loss: 0.811119        Validation Loss: 0.852893
Epoch: 29         Training Loss: 0.822378        Validation Loss: 0.871486
Epoch: 30         Training Loss: 0.820183        Validation Loss: 0.799038
Epoch: 31         Training Loss: 0.796115        Validation Loss: 0.795443
Epoch: 32         Training Loss: 0.793431        Validation Loss: 0.803689
Epoch: 33         Training Loss: 0.801650        Validation Loss: 0.805243
```

```
Epoch: 34         Training Loss: 0.814366         Validation Loss: 0.733682
Validation loss decreased (0.741636 --> 0.733682).  Saving model ...
Epoch: 35         Training Loss: 0.791968         Validation Loss: 0.785890
Epoch: 36         Training Loss: 0.799441         Validation Loss: 0.759192
Epoch: 37         Training Loss: 0.788896         Validation Loss: 0.858263
Epoch: 38         Training Loss: 0.782067         Validation Loss: 0.823034
Epoch: 39         Training Loss: 0.776136         Validation Loss: 0.758932
Epoch: 40         Training Loss: 0.772685         Validation Loss: 0.822122
```

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%.

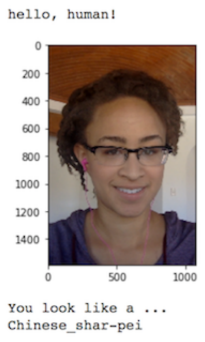
```
In [35]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.000939


Test Accuracy: 78% (653/835)
```

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 [40]: ### 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=data_scratch
         # 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]

         def predict_breed_transfer(img_path):
             model_transfer.eval()
             data_transform = transforms.Compose([transforms.RandomResizedCrop(224),
                                                  transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],s
             image=Image.open(img_path)

             image =data_transform(image)
             image= image.unsqueeze_(0)
             if use_cuda:
                 image = image.cuda()
             ## Return the *index* of the predicted class for that image
             output = model_transfer(image)
             _, pred = torch.max(output, 1)
```



```
hello, human!
```

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

Sample Human Output

```
    #print("Output",pred)
    return class_names[pred]
    # load the image and return the predicted breed
```

## 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 [41]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         def open_img(img_path):
             image=Image.open(img_path)
             plt.imshow(image)
             plt.show()

         def run_app(img_path):

             ## handle cases for a human face, dog, and neither
             Predicted_breed= predict_breed_transfer(img_path)
             if dog_detector(img_path):
                 print('\n Hey Doggy!')
                 open_img(img_path)
                 print('\n you look like',Predicted_breed)
             elif face_detector(img_path):
```

```
        print('\n Hey Human!')
        open_img(img_path)
        print('\n you look like',Predicted_breed)
    else:
        open_img(img_path)
        print('\n error: you are neither human nor a dog')
        print('\n you look like',Predicted_breed)
```

## 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!

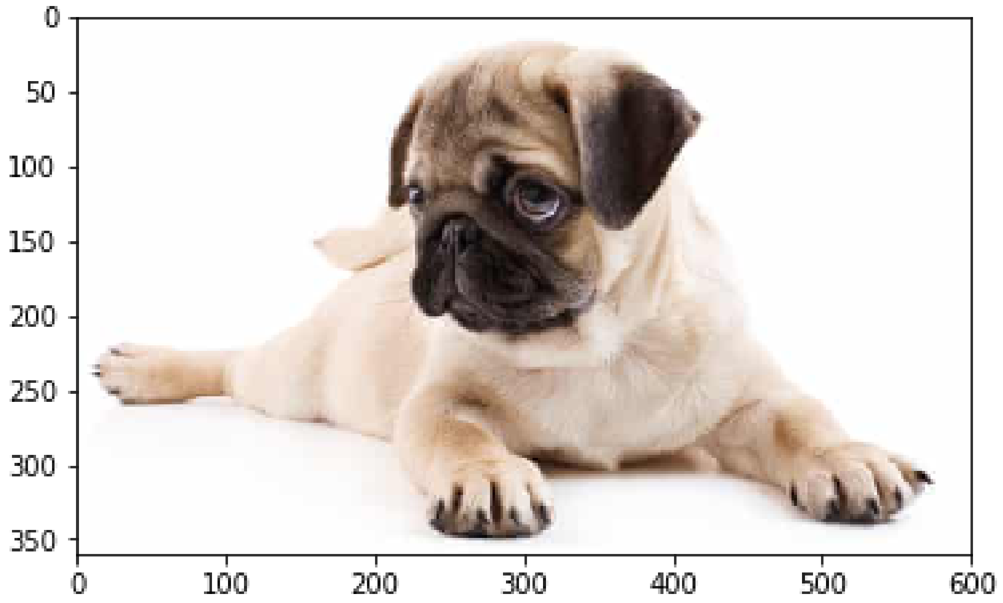
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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) - If we do more image augmentation using more transforms, the training datasel will be larger and we will get better results. - if we make more deeper layers it will increase the efficiency. - Also training with more number of epochs will increase the efficiency on the test data.

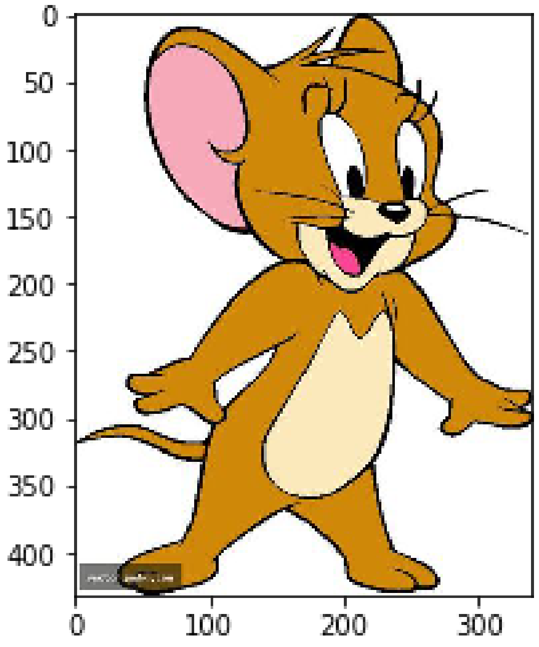
```
In [19]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

         ## suggested code, below
```

```
In [33]: #human_files = np.array(glob("/data/lfw/*/*"))
         my_pictures = glob("./pictures/*")
         for file in np.hstack((my_pictures)):
             run_app(file)
```

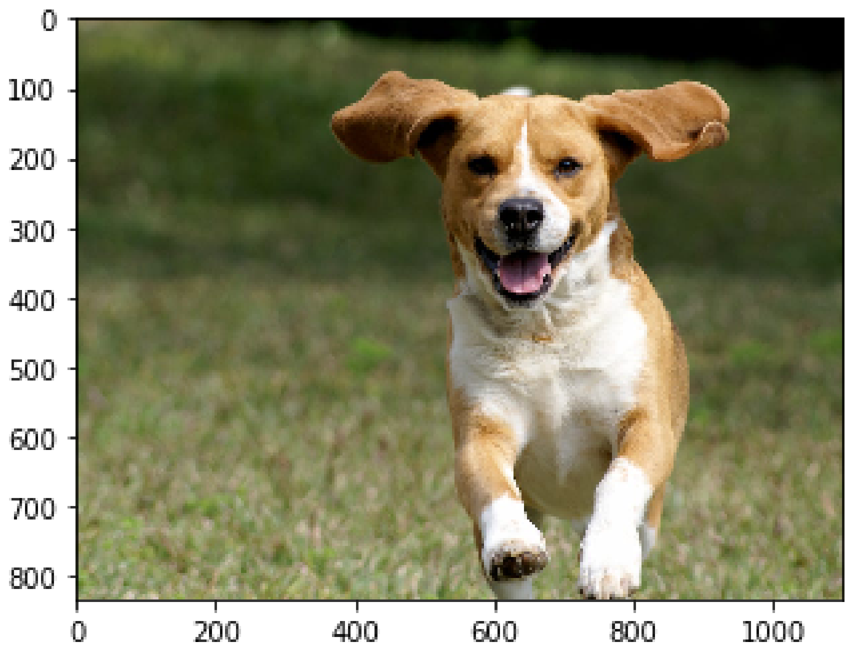


```
error: you are neither human nor a dog

you look like Bullmastiff

Hey Human!
```

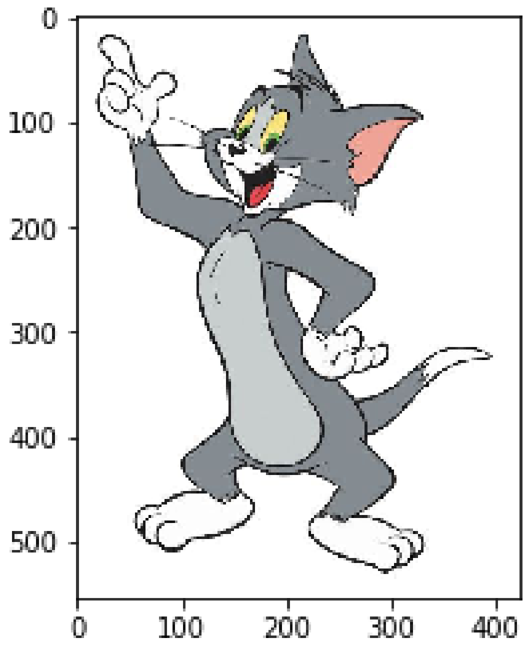
```
you look like Norwich terrier
```



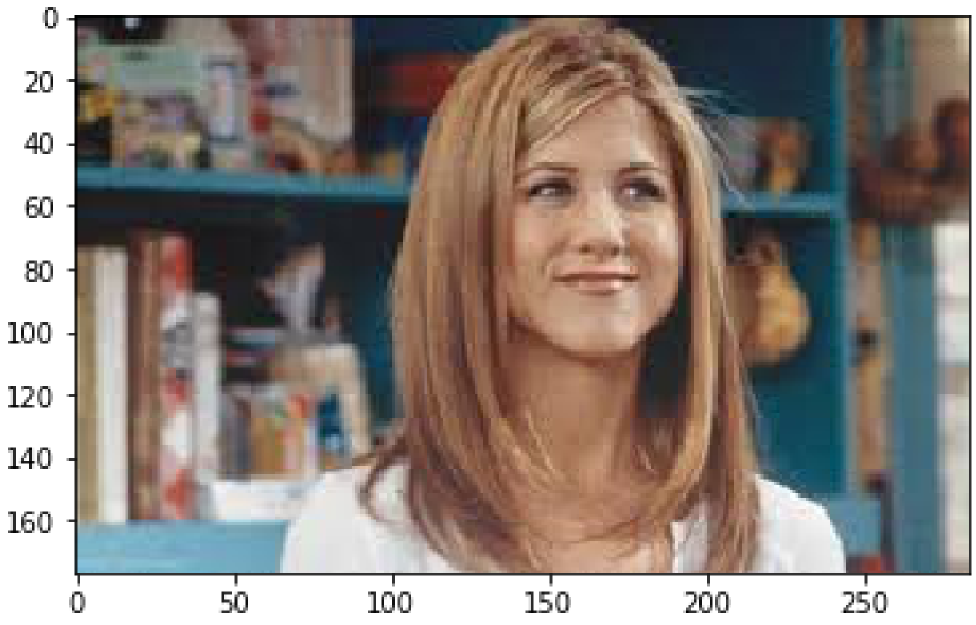
```
error: you are neither human nor a dog

you look like Dalmatian
```



```
error: you are neither human nor a dog

you look like Parson russell terrier
```



```
error: you are neither human nor a dog

you look like German shepherd dog

Hey Human!
```



```
you look like Dachshund
```

```
In [ ]:
```