dog_app

May 7, 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

        # 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

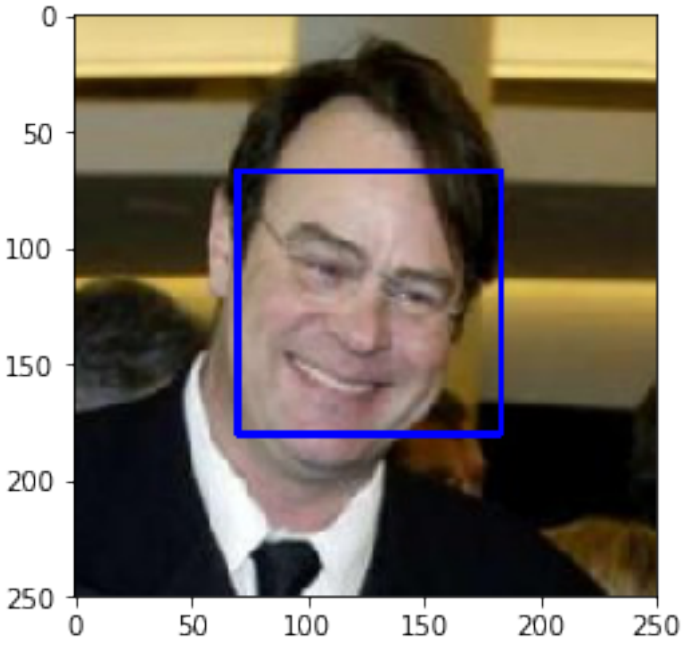
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 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:

- Humans: 98%
- Dogs: 17%

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

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

        ## Test the performance of the face_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        num_humans = 0
        num_dogs = 0

        for _, human in tqdm(enumerate(human_files_short)):
            detected = face_detector(human)
            if detected:
                num_humans += 1

        for _, dog in tqdm(enumerate(dog_files_short)):
            detected = face_detector(dog)
            if detected:
                num_dogs += 1

        print('Percentage of humans detected: {}% \tPecentage of dogs detected: {}%'
```

```
            .format(num_humans / len(human_files_short) * 100, num_dogs / len(dog_files_short)
        )
```

```
100it [00:02, 34.83it/s]
100it [00:29,  3.34it/s]

Percentage of humans detected: 98.0%         Pecentage of dogs detected: 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`.

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 [5]: 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, 17025510.05it/s]
```

```
In [6]: from PIL import Image
        import torchvision.transforms as transforms
```

```
# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

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 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]: 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
            '''
            # Load the image
            image = load_image(img_path)

            if use_cuda:
                image = image.cuda()

            out = VGG16(image)
```

```
    _, index = torch.max(out, 1)

    return index.item() # predicted class index
```

1.1.5 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):

            dog_range = range(151, 268)

            predicted_class = VGG16_predict(img_path)

            is_dog = predicted_class in dog_range

            return is_dog # true/false
```

1.1.6 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 [9]: from tqdm import tqdm

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

        ## Test the performance of the dog_detector algorithm
        ## on the images in human_files_short and dog_files_short.

        num_humans = 0
        num_dogs = 0

        for _, human in tqdm(enumerate(human_files_short)):
            detected = dog_detector(human)
            if detected:
                num_humans += 1
```

```
        for _, dog in tqdm(enumerate(dog_files_short)):
            detected = dog_detector(dog)
            if detected:
                num_dogs += 1

        print('Percentage of humans detected: {}% \tPecentage of dogs detected: {}%'
              .format(num_humans / len(human_files_short) * 100, num_dogs / len(dog_files_short)
             )
```

```
100it [00:04, 24.90it/s]
100it [00:05, 19.33it/s]

Percentage of humans detected: 0.0%         Pecentage of dogs detected: 96.0%
```

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 [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 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 os
         import torch
         from torchvision import datasets
         import torchvision.transforms as transforms

         # how many samples per batch to load
         batch_size = 20

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

         data_transforms = {'train': transforms.Compose([transforms.RandomResizedCrop(224),
                                                transforms.RandomHorizontalFlip(),
                                                transforms.RandomRotation(20),
                                                transforms.ToTensor(),
                                                standard_normalization]),
                            'valid': transforms.Compose([transforms.Resize(256),
                                                transforms.CenterCrop(224),
                                                transforms.ToTensor(),
                                                standard_normalization]),
                            'test': transforms.Compose([transforms.Resize(size=(224,224)),
                                                transforms.ToTensor(),
                                                standard_normalization])
                           }

         # choose the training, validating and test datasets
         train_data = datasets.ImageFolder('/data/dog_images/train', transform=data_transforms['
```

```
valid_data = datasets.ImageFolder('/data/dog_images/valid', transform=data_transforms['
test_data = datasets.ImageFolder('/data/dog_images/test', transform=data_transforms['te


# prepare data loaders (combine dataset and sampler)
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffle=T
valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size)

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

Question 3: Describe your chosen procedure for preprocessing the data. - How does your code resize the images (by cropping, stretching, etc)? What size did you pick for the input tensor, and why? - Did you decide to augment the dataset? If so, how (through translations, flips, rotations, etc)? If not, why not?

Answer:

- The input tensor will have a size of 228 * 228, I picked this size so it can be decreased to a reasonable size in the last maxpooling layer of my model archeticture. The image will be transformed as follows:
  - Training images:
    - Randomly resize and crop to size 228 * 228
    - Randomly flip the image horizontally
  - Validation images: since this is used for validation, I just resized & cropped it.
    - Resize to 256 * 256
    - Center crop to 228 * 228
  - Test images: for testing no need to do anything by resizing.
    - Resize to 228 * 228 Then, all transforms will turn the images into tensors and normalize the pixel values.}
- I argumented the training data set, to avoid overfitting, and let the model see different angles of the images so it generalizes better.

1.1.8 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__()

        ## Define layers of a CNN (features)

        # 1st conv layer: 224 (h) x 224 (w) x 3 (depth)
        self.conv1 = nn.Conv2d(3, 16, 3, padding=1)

        # 2nd conv layer: 112 x 112 x 32
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1)

        # 3rd conv layer: 56 x 56 x 64
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1)

        # 4th conv layer: 28 x 28 x 64
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1)

        # 5th conv layer: 14 x 14 x 128
        self.conv5 = nn.Conv2d(128, 256, 3, padding=1)

        # Maxpooling: 2 x 2 => will decrease dim by 2
        self.pool = nn.MaxPool2d(2, 2)

        # Fully connected layers for the classifier
        self.fc1 = nn.Linear(7 * 7 * 256, 133)

        self.dropout = nn.Dropout(0.2)

    def forward(self, x):

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))

        x = x.view(-1 , 7 * 7 * 256)

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

        return x


# instantiate the CNN
model_scratch = Net()

# move tensors to GPU if CUDA is available
```

```
        if use_cuda:
            model_scratch.cuda()
```

```
In [13]: model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
        model_scratch
```

```
Out[13]: 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))
           (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
           (conv5): Conv2d(128, 256, 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=12544, out_features=133, bias=True)
           (dropout): Dropout(p=0.2)
         )
```

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

Answer:

Features Extractor:

1. Conv: in => 3, out => 16
2. MaxPool: decrease size to 112 * 112
3. Conv: in => 16, out => 32
4. MaxPool: deacrease size to 56 * 56
5. Conv: in => 32, out => 64
6. MaxPool: decrease size to 28 * 28
7. Conv: in => 64, out => 128
8. MaxPool: decrease size to 14 * 14
9. Conv: in => 128, out => 256
10. MaxPool: decrease size to 7 * 7

All convolutional layers are acctivated with **ReLU**.

Classifier

1. Dropout by 0.2
2. Fully connected: in => (77256), out => 133

A **dropout** applied to avoid overfitting. No activation is applied since the classifier has only one layer to classify the final output.

1.1.9 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 [16]: import torch.optim as optim

         criterion_scratch = nn.CrossEntropyLoss()

         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.03)
```

1.1.10 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 [100]: def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):
              """returns trained model"""
              # initialize tracker for minimum validation loss
              valid_loss_min = np.Inf

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

                  ###################
                  # train the model #
                  ###################
                  model.train()
                  for batch_idx, (data, target) in enumerate(loaders['train']):
                      # move to GPU
                      if use_cuda:
                          data, target = data.cuda(), target.cuda()
                      ## find the loss and update the model parameters accordingly
                      optimizer.zero_grad()
                      output = model(data)
                      loss = criterion(output, target)
                      loss.backward()
                      optimizer.step()

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

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

```
            output = model(data)
            loss = criterion(output, target)

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

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

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

    # return trained model
    return model
```

```
In [101]: # load the model that got the best validation accuracy
          model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [102]: # train the model
          model_scratch = train(5, loaders, model_scratch, optimizer_scratch,
                                criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 2.839589         Validation Loss: 3.379555
Validation loss decreased (inf --> 3.379555).  Saving model ...
Epoch: 2         Training Loss: 2.795582         Validation Loss: 3.155316
Validation loss decreased (3.379555 --> 3.155316).  Saving model ...
Epoch: 3         Training Loss: 2.787287         Validation Loss: 3.212044
Epoch: 4         Training Loss: 2.766299         Validation Loss: 3.140712
Validation loss decreased (3.155316 --> 3.140712).  Saving model ...
Epoch: 5         Training Loss: 2.722117         Validation Loss: 3.079833
Validation loss decreased (3.140712 --> 3.079833).  Saving model ...
```

1.1.11 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 [103]: def test(loaders, model, criterion, use_cuda):
```

```
            # monitor test loss and accuracy
            test_loss = 0.
            correct = 0.
            total = 0.

            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['test']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                # forward pass: compute predicted outputs by passing inputs to the model
                output = model(data)
                # calculate the loss
                loss = criterion(output, target)
                # update average test loss
                test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))
                # convert output probabilities to predicted class
                pred = output.data.max(1, keepdim=True)[1]
                # compare predictions to true label
                correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy()
                total += data.size(0)

            print('Test Loss: {:.6f}\n'.format(test_loss))

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

```
In [104]: # call test function
          test(loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.268972


Test Accuracy: 27% (229/836)
```

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 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.

Loaders are same as the loaders from last step

1.1.13 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 [19]: import torchvision.models as models
         import torch.nn as nn

         ## TODO: Specify model architecture
         model_transfer = models.vgg19(pretrained=True)

         # Freeze training for all "features" layers
         for param in model_transfer.features.parameters():
             param.requires_grad = False
```

```
Downloading: "https://download.pytorch.org/models/vgg19-dcbb9e9d.pth" to /root/.torch/models/vgg
100%|| 574673361/574673361 [00:32<00:00, 17831225.53it/s]
```

```
In [24]: classifier = nn.Linear(in_features=model_transfer.classifier[6].in_features,
                                                 out_features=133)

         model_transfer.classifier[6] = classifier
```

```
In [25]: if use_cuda:
             model_transfer = model_transfer.cuda()
```

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:

The features parameters in vgg19 model was freezed, so they will have the weights from the downloaded trained model and won't be affected by the training in this project. Then, the last layer in the classifier, which has index **6** was assigned another layer with same in_features, and out_features set to the number of classes in our dataset.

Why I think this archeticutre is suitable for this problem?

The problem is to detect a dog's breed, and since the VGG19 was trained on ImageNet, which already has a number of classes for dogs, and our dataset is small and similar to ImageNet's, the transfer learning can be done with only changing the last fully connected layer to match our dataset classes.

1.1.14 Specify Loss Function and Optimizer

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

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

1.1.15 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 [105]: # train the model
          model_transfer.load_state_dict(torch.load('model_transfer.pt'))

          model_transfer = train(2, loaders, model_transfer,
                                 optimizer_transfer, criterion_transfer,
                                 use_cuda, 'model_transfer.pt')
```

```
Epoch: 1         Training Loss: 1.088682         Validation Loss: 0.401385
Validation loss decreased (inf --> 0.401385).  Saving model ...
Epoch: 2         Training Loss: 1.111587         Validation Loss: 0.388575
Validation loss decreased (0.401385 --> 0.388575).  Saving model ...
```

1.1.16 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%.

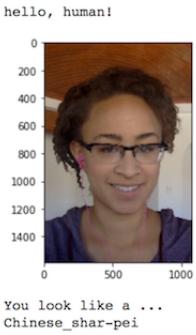
```
In [106]: # load the model that got the best validation accuracy
          model_transfer.load_state_dict(torch.load('model_transfer.pt'))

In [107]: test(loaders, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.485844


Test Accuracy: 84% (704/836)
```

1.1.17 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 [92]: class_names = [item[4:].replace("_", " ") for item in loaders['train'].dataset.classes]

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

             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                                        transforms.ToTensor(), standard_normalization])

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



Sample Human Output

```
    model_transfer.eval()
    idx = torch.argmax(model_transfer(image))
    return class_names[idx]
```

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.

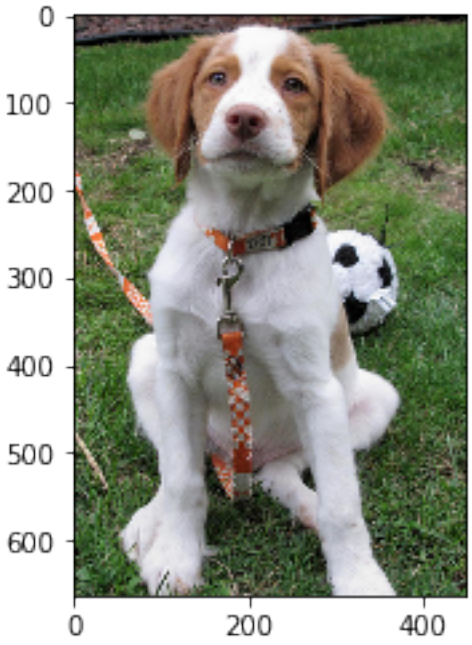
Some sample output for our algorithm is provided below, but feel free to design your own user experience!

1.1.18 Write your Algorithm

```
In [114]: def run_app(img_path):
              ## handle cases for a human face, dog, and neither
              image = Image.open(img_path).convert('RGB')
              plt.imshow(image)
              plt.show()
              if dog_detector(img_path) is True:
                  prediction = predict_breed_transfer(img_path)
                  print("Dogs Detected!\nIt looks like a {0}".format(prediction))
              elif face_detector(img_path) > 0:
                  prediction = predict_breed_transfer(img_path)
                  print("Hello, human!\nIf you were a dog..You may look like a {0}".format(predi
              else:
                  print("Error! Can't detect anything..")

              print('-----------------------------------------')
```

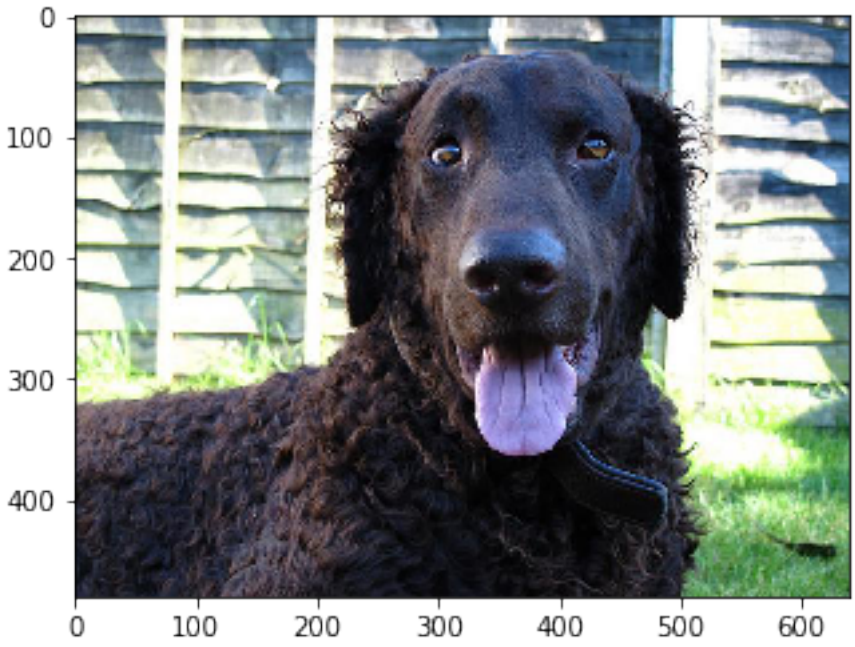
```
In [115]: for img_path in np.array(glob("images/*")):
              run_app(img_path)
```


```
Dogs Detected!
It looks like a Irish red and white setter
----------------------------------------
```

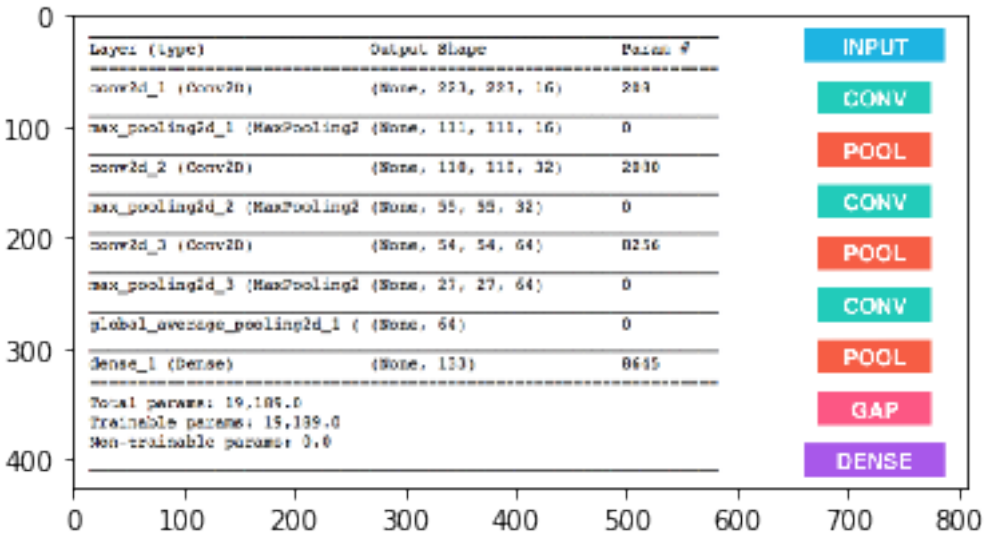


```
Dogs Detected!
It looks like a Brittany
----------------------------------------
```

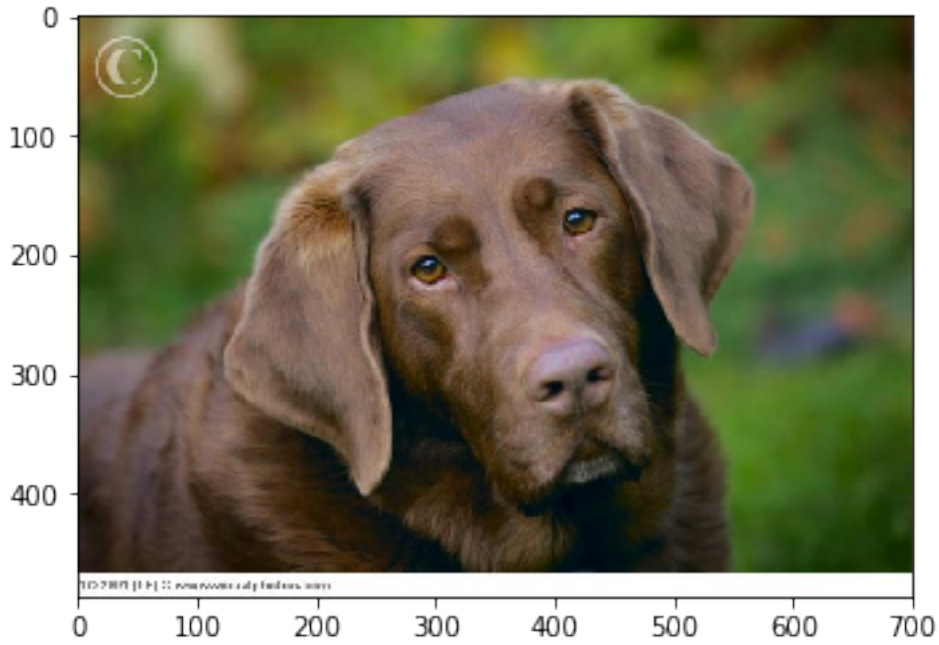

```
Dogs Detected!
It looks like a Labrador retriever
----------------------------------------
```



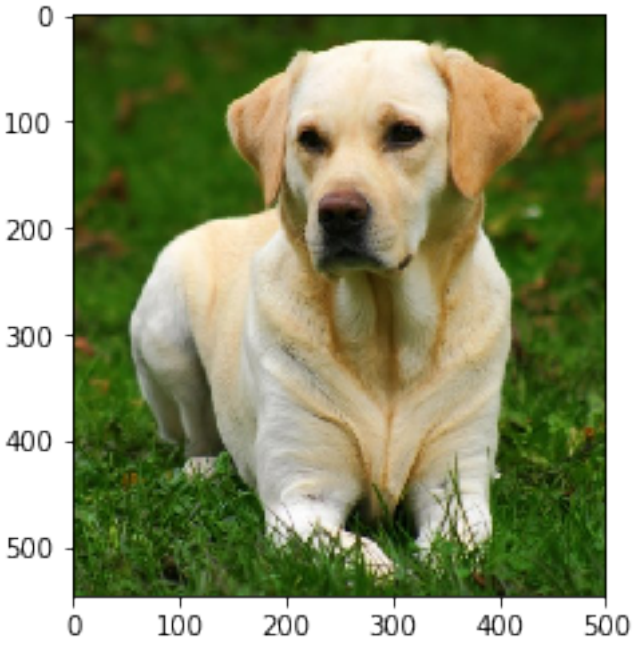

```
Dogs Detected!
It looks like a Curly-coated retriever
----------------------------------------
```



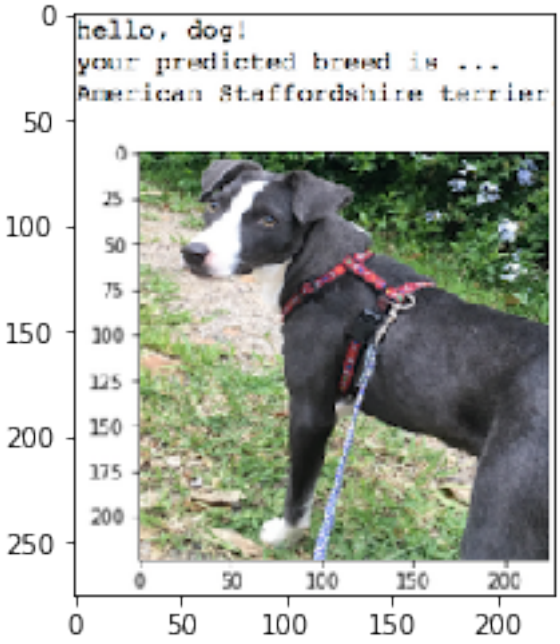

```
Error! Can't detect anything..
----------------------------------------
```



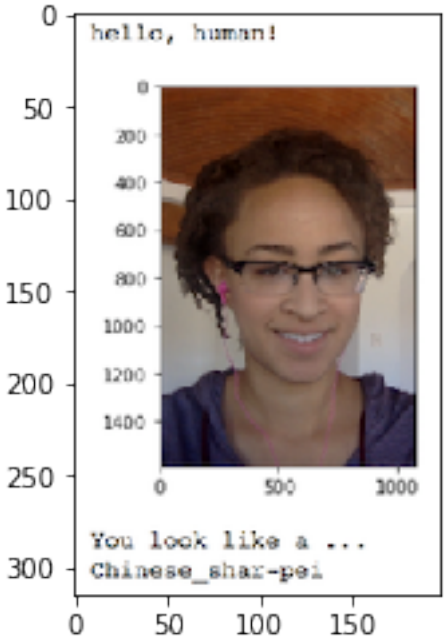

```
Dogs Detected!
It looks like a German shorthaired pointer
-----------------------------------------
```



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




```
Dogs Detected!
It looks like a Great dane
----------------------------------------
```




```
Hello, human!
If you were a dog..You may look like a Chesapeake bay retriever
----------------------------------------
```

```
Dogs Detected!
It looks like a Boykin spaniel
-------------------------------------------
```

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:

The output isn't so bad, but it can be improved.

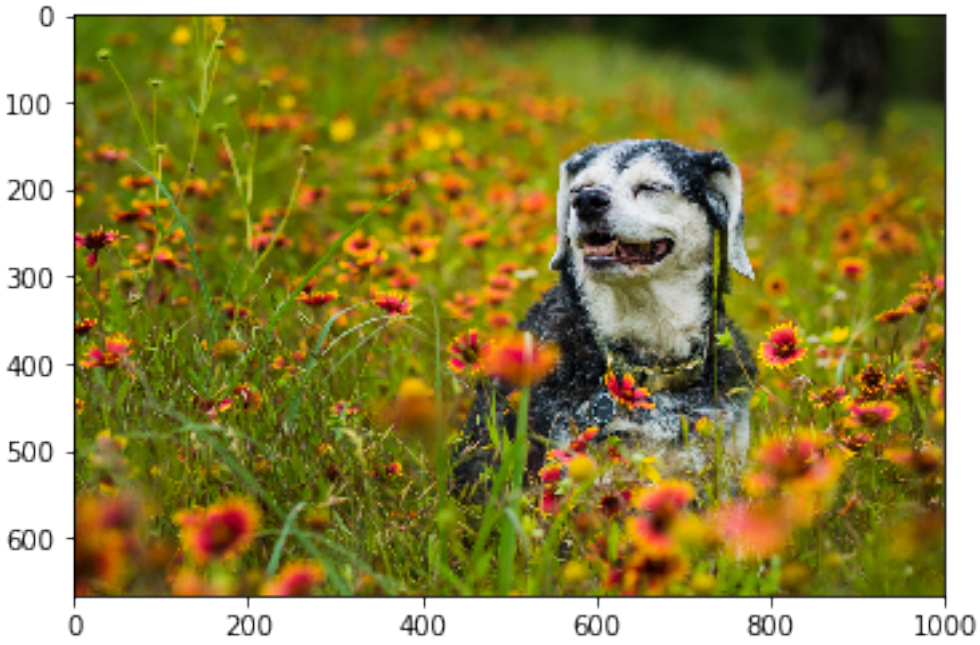
Possible ways of improvments: 1. Train the transfer learning model for more epochs. 2. Try more learning rates. 3. Use more transforms on the dataset, and possibly more images.

```
In [111]: test_images = np.array(glob("./my_images/*"))

          print('There are %d total images.' % len(test_images))
```

```
There are 11 total images.
```

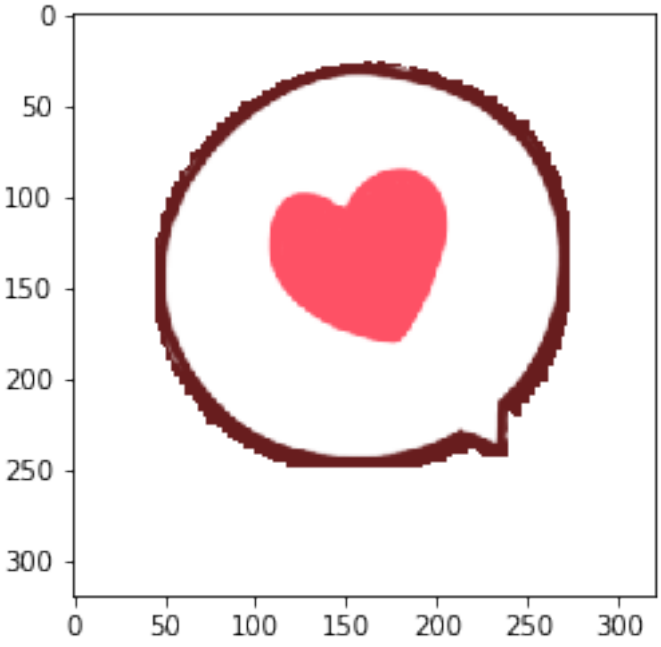
```
In [113]: for file in np.hstack((test_images)):
              run_app(file)
```

```
Hello, human!
If you were a dog..You may look like a Japanese chin
-----------------------------------------
```




```
Dogs Detected!
It looks like a English setter
-----------------------------------------
```

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

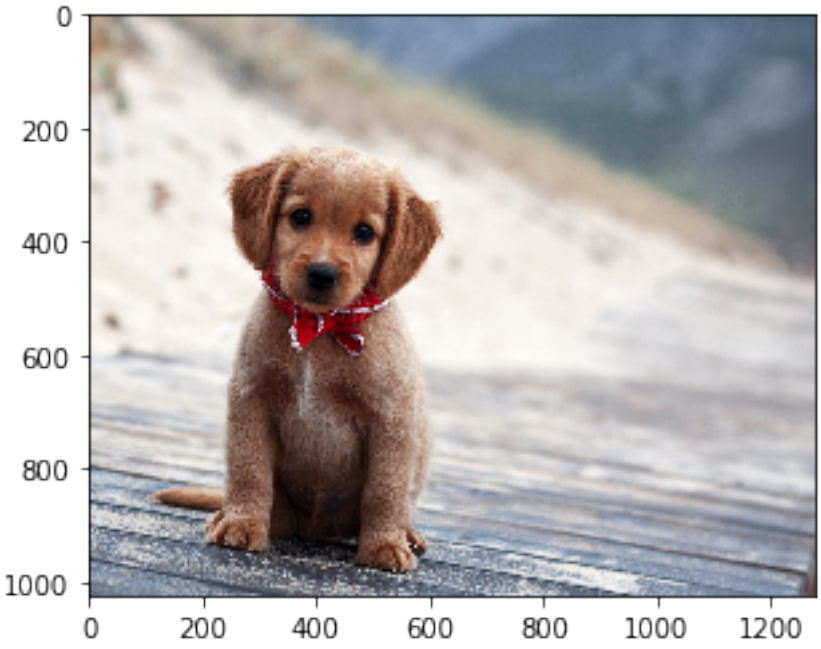



```
Error! Can't detect anything..
----------------------------------------
```

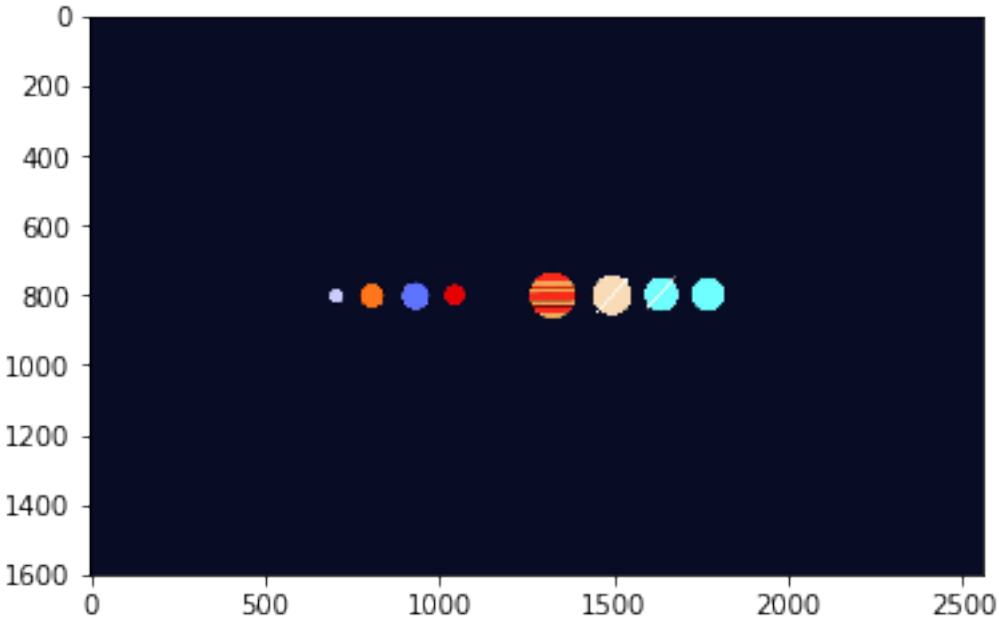
```
Error! Can't detect anything..
----------------------------------------
```


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

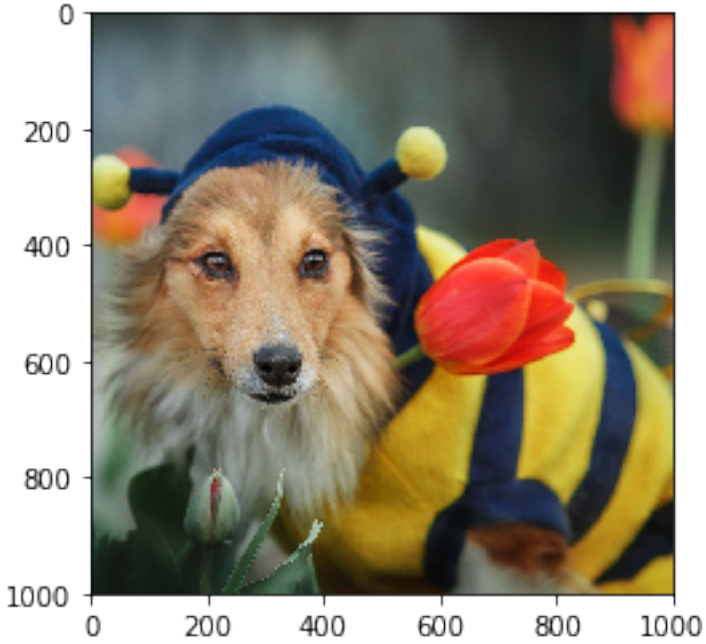
```
Dogs Detected!
It looks like a Collie
----------------------------------------
```



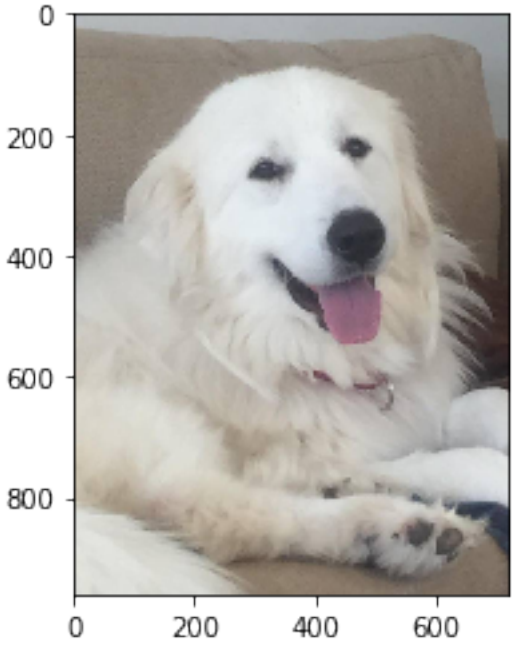

```
Dogs Detected!
It looks like a Golden retriever
----------------------------------------
```




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



```
Dogs Detected!
It looks like a Nova scotia duck tolling retriever
-------------------------------------------
```




```
Dogs Detected!
It looks like a Great pyrenees
-------------------------------------------


In [ ]:
```