dog_app

December 27, 2018

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: * 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 [4]: 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

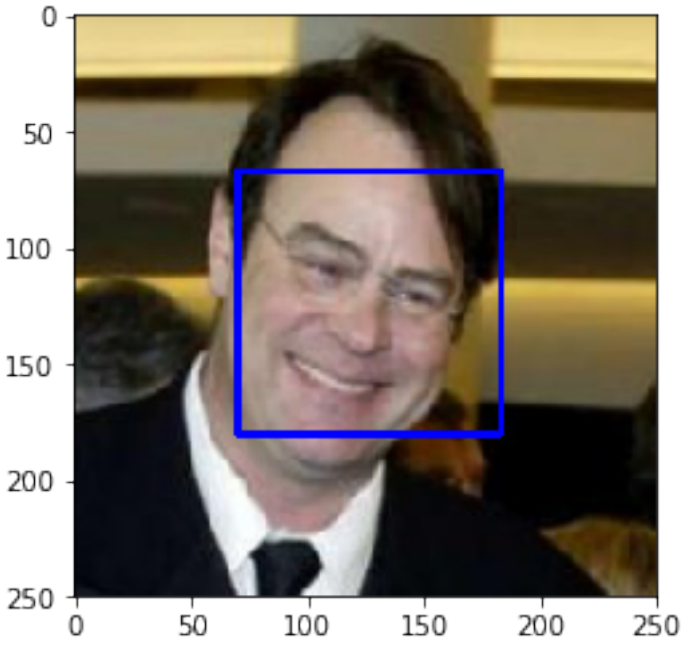
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [5]: 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 [6]: # 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: - percentage of the first 100 images in human_files have a detected human face 98.0%
- percentage of the first 100 images in dog_files have a detected human face 17.0%

In [24]:
```
from tqdm import tqdm

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

#-#-# Do NOT modify the code above this line. #-#-#
human_count = 0.0
dog_count = 0.0
for i in range(100):
    if face_detector(human_files_short[i]):
        human_count += 1
    if face_detector(dog_files_short[i]):
        dog_count += 1
print("percentage of the first 100 images in human_files have a detected human face {}%
print("percentage of the first 100 images in dog_files have a detected human face {}%".
```

```
percentage of the first 100 images in human_files have a detected human face 98.0%
percentage of the first 100 images in dog_files have a detected human face 17.0%
```

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

In [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 [7]: import torch
        import torchvision.models as models

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

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

        # 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 [00:05<00:00, 100433124.22it/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 [8]: from PIL import Image
        import torchvision.transforms as transforms

        VGG16 = VGG16.cuda()

        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
    '''
    image = Image.open(img_path)

    # Apply transforms
    in_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))])

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

    # Predictions using VVG16
    output =  VGG16(image)

    # Return the index with maximum probability
    _, preds_tensor = torch.max(output, 1)
    pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tensor

    return int(pred)
```

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 [9]: ### Returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            index = VGG16_predict(img_path)

            # Dogs classification b/n 151 - 268
            if index >= 151 and index <= 268:
                return True
            return False
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: - percentage of the first 100 images in human_files have a detected dog face 3.0 - percentage of the first 100 images in dog_files have a detected dog face 100.0

```
In [29]: ### Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
         human_count = 0.0
         dog_count = 0.0
         human_files_short[0]
         for i in range(100):
             if dog_detector(human_files_short[i]):
                 human_count += 1
             if dog_detector(dog_files_short[i]):
                 dog_count += 1
         print("percentage of the first 100 images in human_files have a detected dog face {}".f
         print("percentage of the first 100 images in dog_files have a detected dog face {}".for
```

```
percentage of the first 100 images in human_files have a detected dog face 3.0
percentage of the first 100 images in dog_files have a detected dog face 100.0
```

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

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

         # train data transforms
         data_transform = transforms.Compose([
             transforms.RandomRotation(10),
             transforms.RandomResizedCrop(224),
             transforms.ToTensor(),
             transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
         ])

         # test, valid data transforms
         test_data_transform = transforms.Compose([
```

```
        transforms.Resize(224),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
    ])


    train_data = datasets.ImageFolder(train_dir, transform=data_transform)
    valid_data = datasets.ImageFolder(valid_dir, transform=test_data_transform)
    test_data = datasets.ImageFolder(test_dir, transform=test_data_transform)

    ### data loaders for training, validation, and test sets
    ## Specify appropriate transforms, and batch_sizes
    batch_size = 25
    num_workers= 0

    # prepare data loaders
    train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                               num_workers=num_workers, shuffle=True)
    valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                               num_workers=num_workers, shuffle=True)
    test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                              num_workers=num_workers, shuffle=True)
    loaders_scratch ={'train': train_loader, 'valid': valid_loader, 'test': test_loader}
```

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

Answer: We use data loaders to load the data and resize the image to 224x224x3. Images can come in different sizes, so we need to resize the images to specific size. We have to consider the resolution needed for the images. So we choose bigger size than MMNIST data set size for dog classification. Too much large size can also result in high load to GPU. Most of the pretrained networks consider this 224x224x3 as input.

We apply different transformers for traning and testing data.

We use two transforms, one for train data and other for test and validation data. - train tranforms - Random Rotation, RandomResizedCrop to prevent overfitting - test/ vaildation transfors - Resize, centercrop to resize to 224x224x3 PIL image - we also apply normalization to help us work with decimals instead of large values.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [29]: import torch.nn as nn
         import torch.nn.functional as F

         # define the CNN architecture
```

```
        class Net(nn.Module):
            def __init__(self):
                super(Net, self).__init__()
                # Define CNN layers
                self.conv1 = nn.Conv2d(3, 64, 3, padding=1)
                self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
                self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
                self.maxPool2d = nn.MaxPool2d(2, 2)
                # Define FCU layer
                self.fc1 = nn.Linear(256 * 28 * 28, 512)
                self.fc2 = nn.Linear(512, 133)
                self.dropout = nn.Dropout(0.3)

            def forward(self, x):
                ## Define forward behavior
                x = self.maxPool2d(F.relu(self.conv1(x)))
                x = self.maxPool2d(F.relu(self.conv2(x)))
                x = self.maxPool2d(F.relu(self.conv3(x)))
                x = x.view(-1, 256 * 28 * 28)
                x = F.relu(self.fc1(self.dropout(x)))
                x = self.fc2(self.dropout(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
        use_cuda = torch.cuda.is_available()
        if use_cuda:
            model_scratch = model_scratch.cuda()

        def model_param_info(model):
            params = sum(p.numel() for p in model.parameters())
            trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
            print('Trainable params {}, Total params {}'.format(trainable_params, params))

        print(model_scratch)
        model_param_info(model_scratch)
```

```
Net(
  (conv1): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (maxPool2d): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=200704, out_features=512, bias=True)
  (fc2): Linear(in_features=512, out_features=133, bias=True)
```

```
  (dropout): Dropout(p=0.3)
)
Trainable params 103200005, Total params 103200005
```

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

Answer: I started using similar model used for vgg16. Since we don't want a complex model which need lot of training time, we limited the cnn layers to 3. I kept on increasing the depth by twice for cnn, also by adding maxpool layer to decrease height, width by half. Initially I used many cnn's and linear layers, which made the model endup with lot of parameters. This also resulted in huge training time. So limited the linear layers to 2 hidden layers.

- cnn1 224x224x3 --maxpool-> 112x112x64
- cnn2 112x112x64 --maxpool-> 56x56x128
- cnn3 56x56x128 --maxpool-> 28x28x256
- 28x28x256 -fc1-> 512
- 512 -fc2->133(output)

Since I couldn't see much change in validation accuracy after ~25 epoches, so limited epoches to 25 instead of initially given 100.

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 [3]:
```
import torch.optim as optim

### loss function
criterion_scratch = nn.CrossEntropyLoss()

### 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 [4]:
```
import numpy as np
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

        ###################
        # train the model #
        ###################
        model.train()
        for batch_idx, (data, target) in enumerate(loaders['train']):
            # move to GPU
            if use_cuda:
                data, target = data.cuda(), target.cuda()
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            # update training 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)
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

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

    # return trained model
    return model

# train the model
from workspace_utils import keep_awake
```

```
        for i in keep_awake(range(1)):
            model_scratch = train(30, loaders_scratch, model_scratch, optimizer_scratch, criteri

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

```
Epoch: 1         Training Loss: 4.882747         Validation Loss: 4.853453
Validation loss decreased (inf --> 4.853453).  Saving model ...
Epoch: 2         Training Loss: 4.831620         Validation Loss: 4.755530
Validation loss decreased (4.853453 --> 4.755530).  Saving model ...
Epoch: 3         Training Loss: 4.720798         Validation Loss: 4.610343
Validation loss decreased (4.755530 --> 4.610343).  Saving model ...
Epoch: 4         Training Loss: 4.629630         Validation Loss: 4.511194
Validation loss decreased (4.610343 --> 4.511194).  Saving model ...
Epoch: 5         Training Loss: 4.575688         Validation Loss: 4.456190
Validation loss decreased (4.511194 --> 4.456190).  Saving model ...
Epoch: 6         Training Loss: 4.525138         Validation Loss: 4.464795
Epoch: 7         Training Loss: 4.467952         Validation Loss: 4.376814
Validation loss decreased (4.456190 --> 4.376814).  Saving model ...
Epoch: 8         Training Loss: 4.442785         Validation Loss: 4.307696
Validation loss decreased (4.376814 --> 4.307696).  Saving model ...
Epoch: 9         Training Loss: 4.411265         Validation Loss: 4.326186
Epoch: 10         Training Loss: 4.373693         Validation Loss: 4.257763
Validation loss decreased (4.307696 --> 4.257763).  Saving model ...
Epoch: 11         Training Loss: 4.326352         Validation Loss: 4.188396
Validation loss decreased (4.257763 --> 4.188396).  Saving model ...
Epoch: 12         Training Loss: 4.298756         Validation Loss: 4.244473
Epoch: 13         Training Loss: 4.276258         Validation Loss: 4.232751
Epoch: 14         Training Loss: 4.245475         Validation Loss: 4.216269
Epoch: 15         Training Loss: 4.205585         Validation Loss: 4.105568
Validation loss decreased (4.188396 --> 4.105568).  Saving model ...
Epoch: 16         Training Loss: 4.171382         Validation Loss: 4.078811
Validation loss decreased (4.105568 --> 4.078811).  Saving model ...
Epoch: 17         Training Loss: 4.146917         Validation Loss: 4.186315
Epoch: 18         Training Loss: 4.102564         Validation Loss: 4.225090
Epoch: 19         Training Loss: 4.080231         Validation Loss: 4.044093
Validation loss decreased (4.078811 --> 4.044093).  Saving model ...
Epoch: 20         Training Loss: 4.035469         Validation Loss: 4.118454
Epoch: 21         Training Loss: 4.014609         Validation Loss: 3.962836
Validation loss decreased (4.044093 --> 3.962836).  Saving model ...
Epoch: 22         Training Loss: 3.976301         Validation Loss: 4.064164
Epoch: 23         Training Loss: 3.975634         Validation Loss: 3.994351
Epoch: 24         Training Loss: 3.938764         Validation Loss: 3.992113
Epoch: 25         Training Loss: 3.895207         Validation Loss: 3.980756
Epoch: 26         Training Loss: 3.867775         Validation Loss: 4.886085
Epoch: 27         Training Loss: 3.834727         Validation Loss: 4.031542
Epoch: 28         Training Loss: 3.816405         Validation Loss: 3.897043
Validation loss decreased (3.962836 --> 3.897043).  Saving model ...
```

```
Epoch: 29         Training Loss: 3.803554         Validation Loss: 3.859910
Validation loss decreased (3.897043 --> 3.859910).  Saving model ...
Epoch: 30         Training Loss: 3.756050         Validation Loss: 4.112590
```

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 [5]: import numpy as np
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        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))
        model_scratch.load_state_dict(torch.load('model_scratch.pt'))
        # call test function
        test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.806852
```

```
Test Accuracy: 12% (105/836)
```

Step 4: Create a CNN to Classify Dog Breeds (using Transfer Learning)

You will now use transfer learning to create a CNN that can identify dog breed from images. Your CNN must attain at least 60% accuracy on the test set.

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively).

If you like, **you are welcome to use the same data loaders from the previous step**, when you created a CNN from scratch.

```
In [7]: import os
        from torchvision import datasets
        import torchvision.transforms as transforms
        import torch

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

        # transforms for training data
        data_transform = transforms.Compose([
            transforms.RandomRotation(10),
            transforms.RandomResizedCrop(224),
            transforms.ToTensor(),
            transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
        ])

        # transforms for valid, test data
        test_data_transform = transforms.Compose([
            transforms.Resize(224),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
        ])


        train_data = datasets.ImageFolder(train_dir, transform=data_transform)
        valid_data = datasets.ImageFolder(valid_dir, transform=test_data_transform)
        test_data = datasets.ImageFolder(test_dir, transform=test_data_transform)
```

```
        # data loaders for training, validation, and test sets
        # transforms, and batch_sizes
        batch_size = 25
        num_workers= 0

        # prepare data loaders
        train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
        valid_loader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size,
                                                   num_workers=num_workers, shuffle=True)
        test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,
                                                  num_workers=num_workers, shuffle=True)

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

In [8]:
```
        import numpy as np

        # image file is truncated (150 bytes not processed) -- To prevent this error we make LOA
        from PIL import ImageFile
        ImageFile.LOAD_TRUNCATED_IMAGES = True

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

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

                ###################
                # train the model #
                ###################
                model.train()
                for batch_idx, (data, target) in enumerate(loaders['train']):
                    # move to GPU
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()
                    optimizer.zero_grad()
                    output = model(data)
                    loss = criterion(output, target)
                    loss.backward()
                    optimizer.step()
                    # update training loss
                    train_loss += loss.item()*data.size(0)

                ######################
```

```
            # 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)
                valid_loss += loss.item()*data.size(0)
            train_loss = train_loss / len(loaders['train'].dataset)
            valid_loss = valid_loss / len(loaders['valid'].dataset)

            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(epoch

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

        # return trained model
        return model

    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))
```

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

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

         for param in model_transfer.features.parameters():
             param.requires_grad = False

         model_transfer.classifier[6] = nn.Linear(model_transfer.classifier[3].in_features, 133)
         print(model_transfer)
         model_param_info(model_transfer)

         use_cuda = torch.cuda.is_available()

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

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
```

```
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
Trainable params 120090757, Total params 134805445
```

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:

Here we use VGG16 model for transfer learning. This pretrained network deals with similar image and we only have small dataset to train the network. This initial layer will already be able to recognise patterns. So we will freeze the initial layer, training the classifier layer.

We can use the same pretrained network with only changing the last classifier layer to match 133 target dog breed classifier.

So we changed model.classifier[6] : Linear(in_features=4096, out_features=133, bias=True)

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 [15]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.01)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters at filepath `'model_transfer.pt'`.

In [16]:
```
# train the model

        import workspace_utils
        from workspace_utils import keep_awake

        for i in keep_awake(range(1)):
            model_transfer = train(25, loaders_transfer, model_transfer, optimizer_transfer, cr
            # 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: 2.146579         Validation Loss: 0.648617
Validation loss decreased (inf --> 0.648617).  Saving model ...
Epoch: 2         Training Loss: 1.167975         Validation Loss: 0.659335
Epoch: 3         Training Loss: 1.069724         Validation Loss: 0.551857
Validation loss decreased (0.648617 --> 0.551857).  Saving model ...
Epoch: 4         Training Loss: 0.956475         Validation Loss: 0.598258
Epoch: 5         Training Loss: 0.914650         Validation Loss: 0.416889
Validation loss decreased (0.551857 --> 0.416889).  Saving model ...
Epoch: 6         Training Loss: 0.893721         Validation Loss: 0.408209
Validation loss decreased (0.416889 --> 0.408209).  Saving model ...
Epoch: 7         Training Loss: 0.834280         Validation Loss: 0.443306
Epoch: 8         Training Loss: 0.809749         Validation Loss: 0.404010
Validation loss decreased (0.408209 --> 0.404010).  Saving model ...
Epoch: 9         Training Loss: 0.804581         Validation Loss: 0.405045
Epoch: 10         Training Loss: 0.749564         Validation Loss: 0.446651
Epoch: 11         Training Loss: 0.752444         Validation Loss: 0.448430
Epoch: 12         Training Loss: 0.739907         Validation Loss: 0.389570
Validation loss decreased (0.404010 --> 0.389570).  Saving model ...
Epoch: 13         Training Loss: 0.745170         Validation Loss: 0.394089
Epoch: 14         Training Loss: 0.710237         Validation Loss: 0.389230
Validation loss decreased (0.389570 --> 0.389230).  Saving model ...
Epoch: 15         Training Loss: 0.705683         Validation Loss: 0.385887
Validation loss decreased (0.389230 --> 0.385887).  Saving model ...
Epoch: 16         Training Loss: 0.665164         Validation Loss: 0.375585
Validation loss decreased (0.385887 --> 0.375585).  Saving model ...
Epoch: 17         Training Loss: 0.668961         Validation Loss: 0.371569
Validation loss decreased (0.375585 --> 0.371569).  Saving model ...
Epoch: 18         Training Loss: 0.661517         Validation Loss: 0.529800
Epoch: 19         Training Loss: 0.665169         Validation Loss: 0.372799
Epoch: 20         Training Loss: 0.636196         Validation Loss: 0.390249
Epoch: 21         Training Loss: 0.627854         Validation Loss: 0.371041
Validation loss decreased (0.371569 --> 0.371041).  Saving model ...
Epoch: 22         Training Loss: 0.593255         Validation Loss: 0.367549
Validation loss decreased (0.371041 --> 0.367549).  Saving model ...
```

```
Epoch: 23         Training Loss: 0.586197         Validation Loss: 0.388484
Epoch: 24         Training Loss: 0.588669         Validation Loss: 0.368553
Epoch: 25         Training Loss: 0.602355         Validation Loss: 0.395607
```

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%.

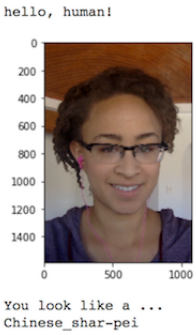
```
In [17]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.471380


Test Accuracy: 87% (734/836)
```

1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

Write a function that takes an image path as input and returns the dog breed (`Affenpinscher`, `Afghan hound`, etc) that is predicted by your model.

```
In [12]: import random
         from PIL import Image
         import matplotlib.pyplot as plt

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

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

         def predict_breed_transfer(img_path):
             # Get image tensor for given image path
             image = Image.open(img_path)
             data_transform = transforms.Compose([
                 transforms.Resize(224),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize((0.485, 0.456, 0.406), (0.229, 0.224, 0.225))
             ])
             image = data_transform(image)[:3,:,:].unsqueeze(0)

             if use_cuda:
                 image = image.cuda()

             # Return the class name with highest probability
             output = model_transfer(image)
```



Sample Human Output

```
    _, preds_tensor = torch.max(output, 1)
    pred = np.squeeze(preds_tensor.numpy()) if not use_cuda else np.squeeze(preds_tenso
    return class_names[pred]
```

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 [13]:
```
### Algorithm that predicts the given image.

def show_img(img_path):
    image = Image.open(img_path)
    plt.imshow(image)
    plt.show()

def run_app(img_path):
    msg = "The image is neither human nor dog."
    if (dog_detector(img_path)):
        predicted_breed = predict_breed_transfer(img_path)
        msg = "hello dog! you look like a " + predicted_breed
    elif (face_detector(img_path)):
        predicted_breed = predict_breed_transfer(img_path)
        msg = "hello human! you look like a " + predicted_breed
    else:
        print(msg)
```

```
        show_img(img_path)
        return
    print(msg)
    show_img(img_path)
```

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.

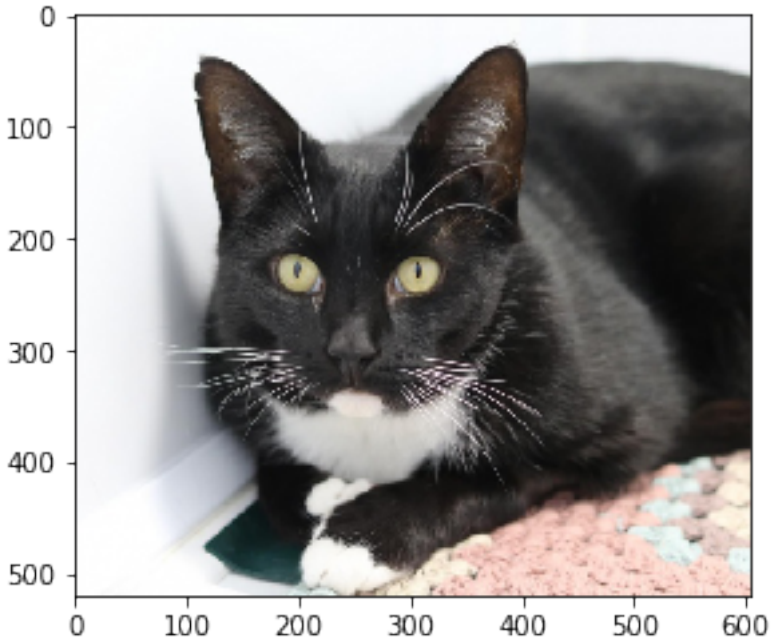
Answer:

The model gave a test accuracy of 87%. This is actually better output than expected. Most of the images tested are accurately classified. For cat image, it classified as neither human or dog correctly. For human faces one images, one image with side face posture is classified as neither. For dogs most of them are classified correctly. We can try the following to improve the algorithm. - Train dog classification model with more varying training data. This can let the algorithm detect images with other disturbances - Add one more Linear layer to classifier, increasing the number of parameters and also increasing the training time to train these parameters. - We can also tune our hyperparams to fine tune fully connected layers, that increases the accuracy.

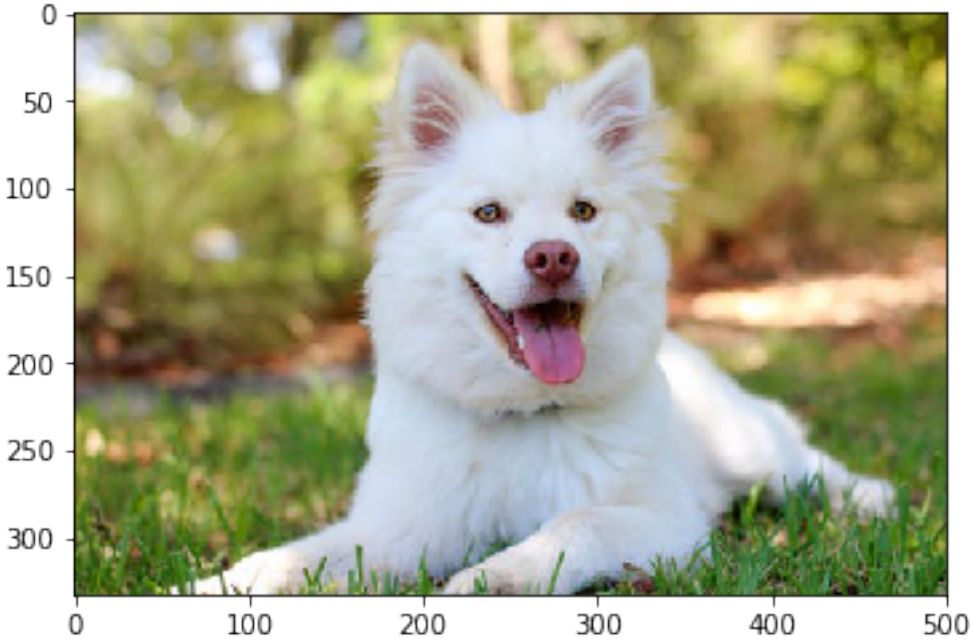
In [38]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
import cv2

## suggested code, below
import numpy as np
from glob import glob
model_transfer.load_state_dict(torch.load('model_transfer.pt'))

# load filenames for human and dog images
run_app("test_images/cat.jpg")
run_app("test_images/dog1.jpeg")
run_app("test_images/dog2.jpg")
run_app("test_images/dog3.jpg")
run_app("test_images/dog4.jpg")
run_app("test_images/dog5.jpg")
run_app("test_images/dog6.jpg")
run_app("test_images/dog7.jpg")
```

```
run_app("test_images/dog8.jpg")
run_app("test_images/dog9.jpg")
run_app("test_images/dog10.jpg")
run_app("test_images/dog11.jpg")
run_app("test_images/dog12.jpg")
run_app("test_images/dog13.jpg")
run_app("test_images/dog14.jpg")
run_app("test_images/dog15.jpg")
run_app("test_images/dog15.png")
run_app("test_images/dog16.jpg")
run_app("test_images/dog17.jpg")
run_app("test_images/dog18.jpg")
run_app("test_images/dog19.jpg")
run_app("test_images/dog20.jpg")
run_app("test_images/dog21.jpg")
run_app("test_images/dog22.jpg")
run_app("test_images/dog23.jpg")
run_app("test_images/dog25.jpeg")
run_app("test_images/human1.jpg")
run_app("test_images/human4.jpg")
run_app("test_images/human2.jpg")
run_app("test_images/human3.jpg")
# for file in np.hstack((human_files[20:23], dog_files[100:103])):
#     run_app(file)
```

```
The image is neither human nor dog.
```

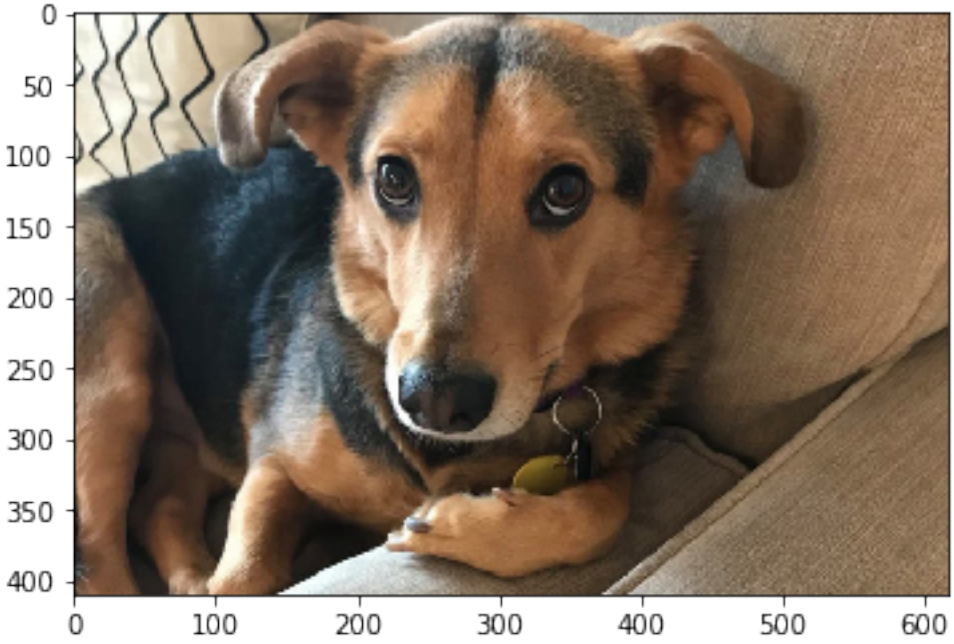


```
hello dog! you look like a American eskimo dog
```

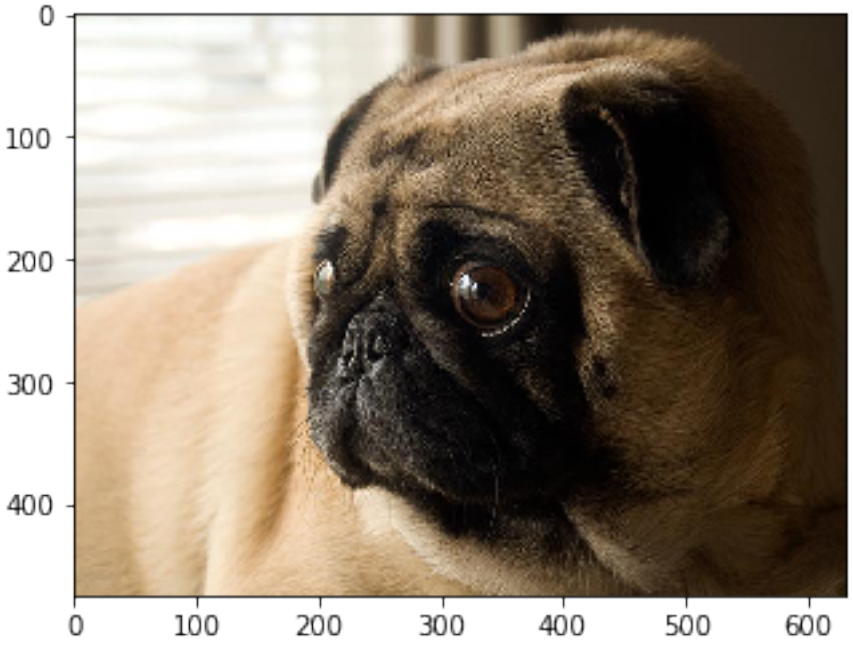



```
hello dog! you look like a American eskimo dog
```

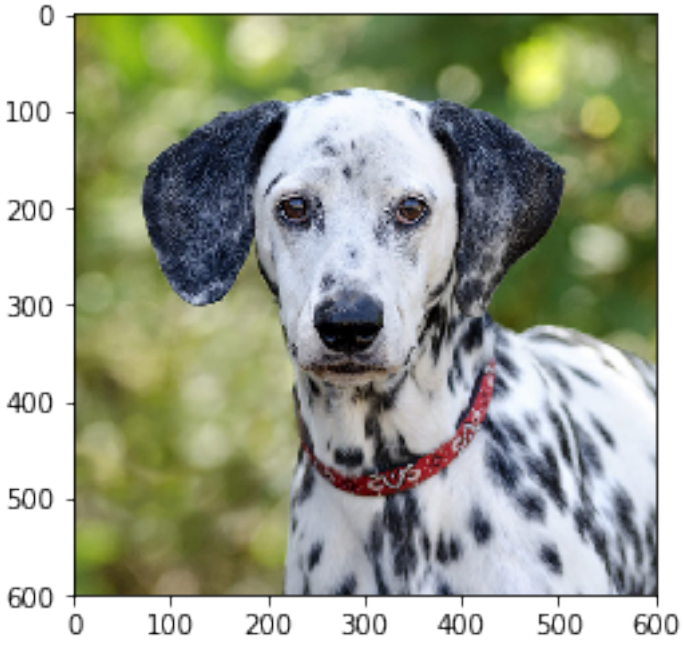
```
hello dog! you look like a Dachshund
```



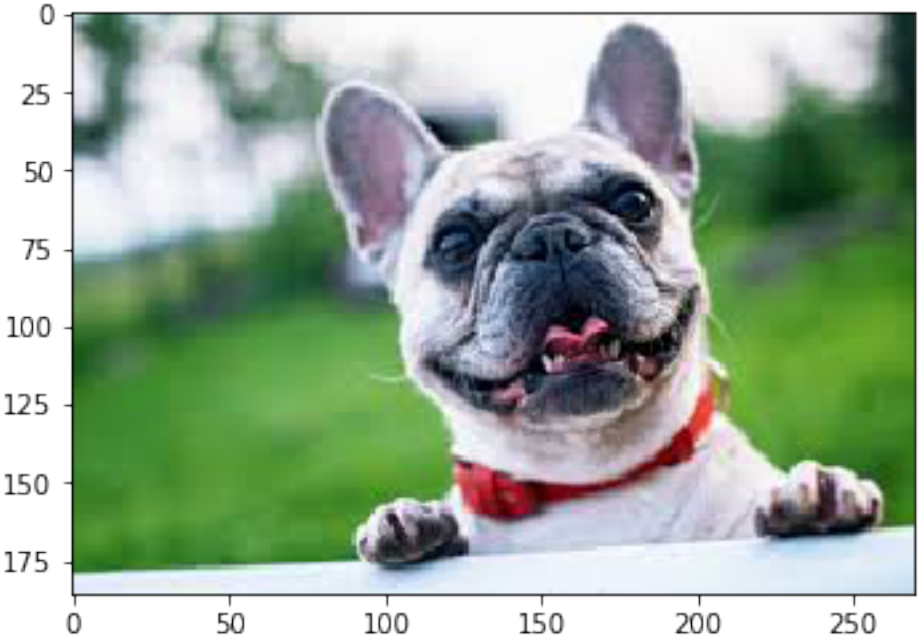

```
hello dog! you look like a Mastiff
```



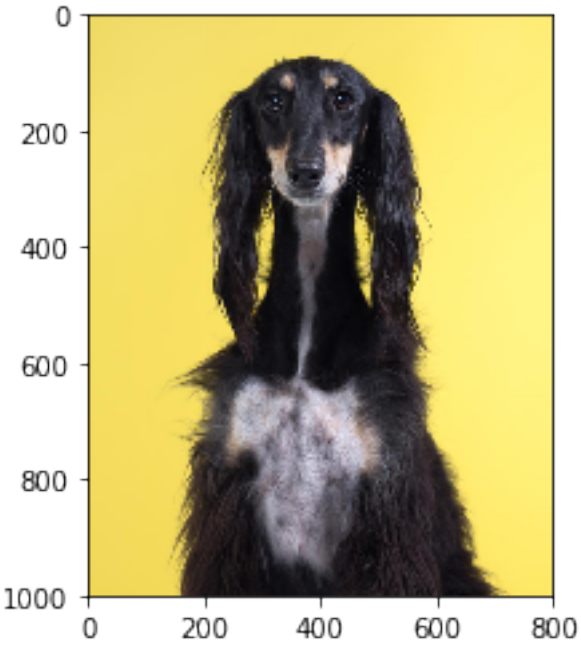
```
hello dog! you look like a Dalmatian
```



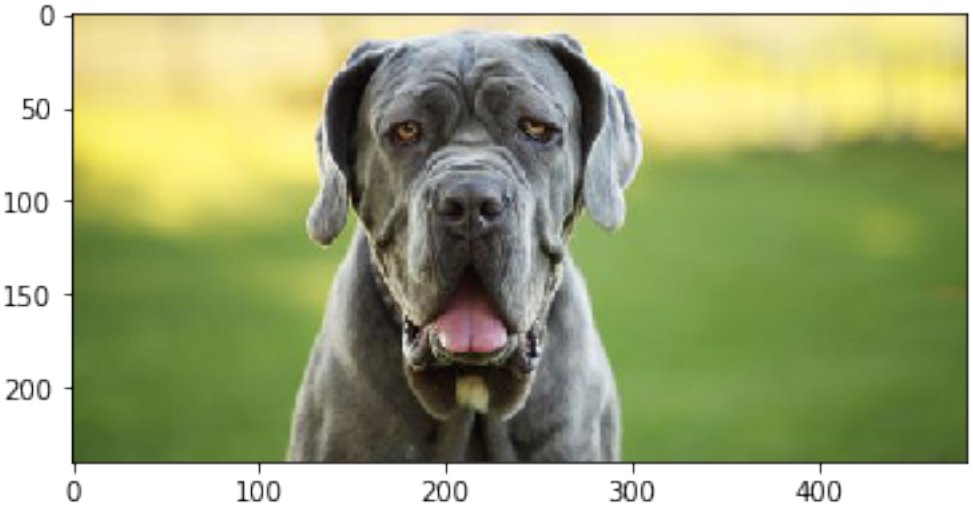

```
hello dog! you look like a French bulldog
```



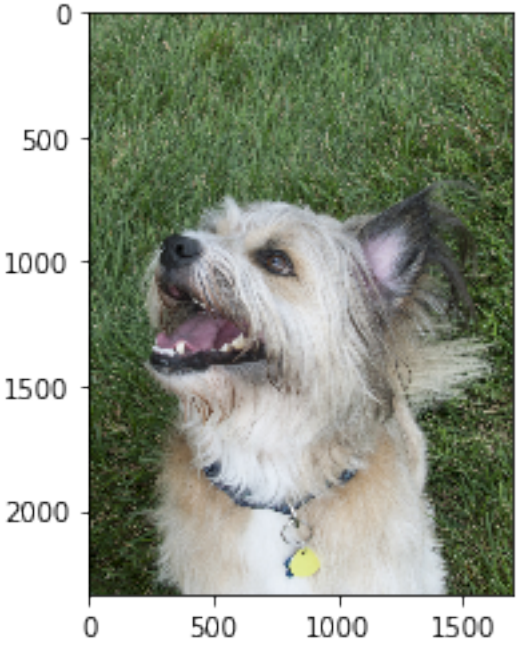
```
hello dog! you look like a Afghan hound
```



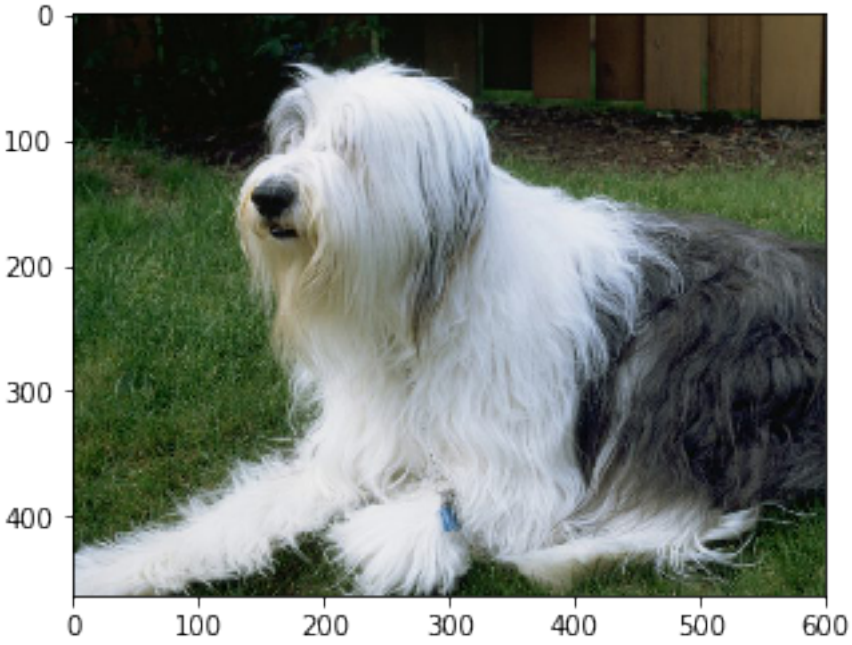

```
hello dog! you look like a Neapolitan mastiff
```



```
hello dog! you look like a Cairn terrier
```



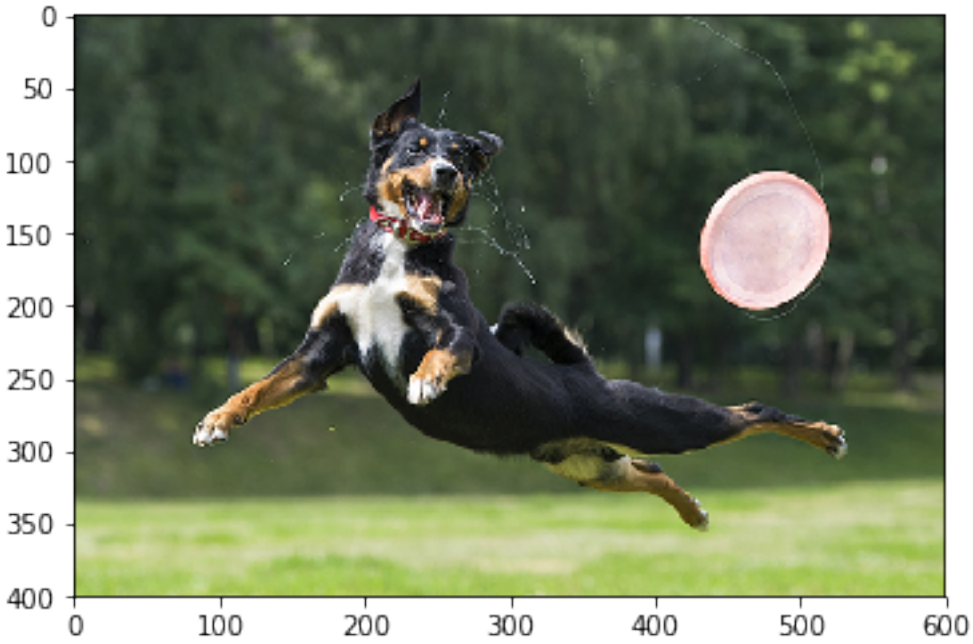

```
hello dog! you look like a Bearded collie
```



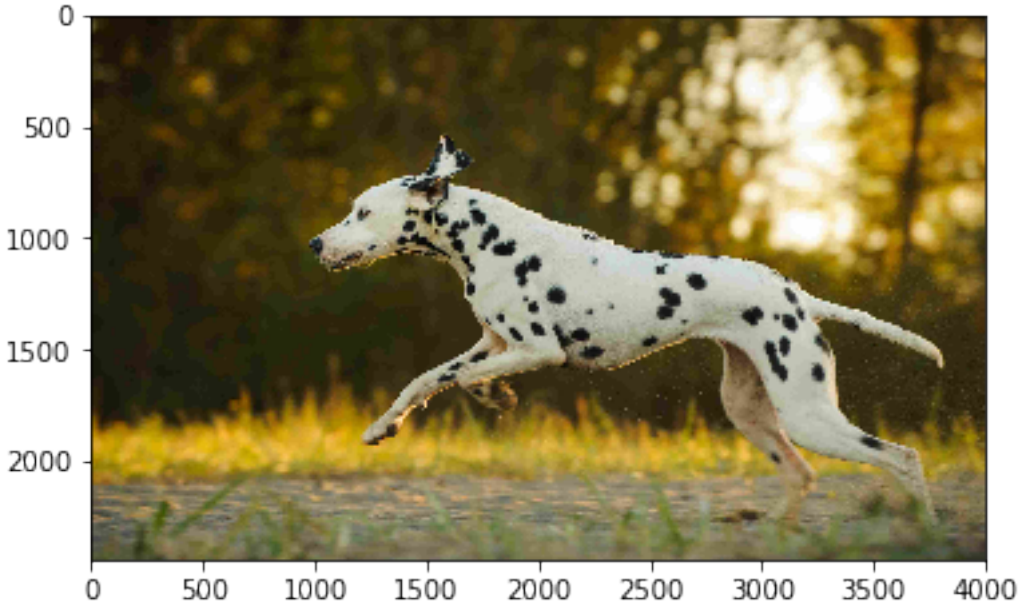
```
hello dog! you look like a Collie
```


```
hello dog! you look like a Collie
```

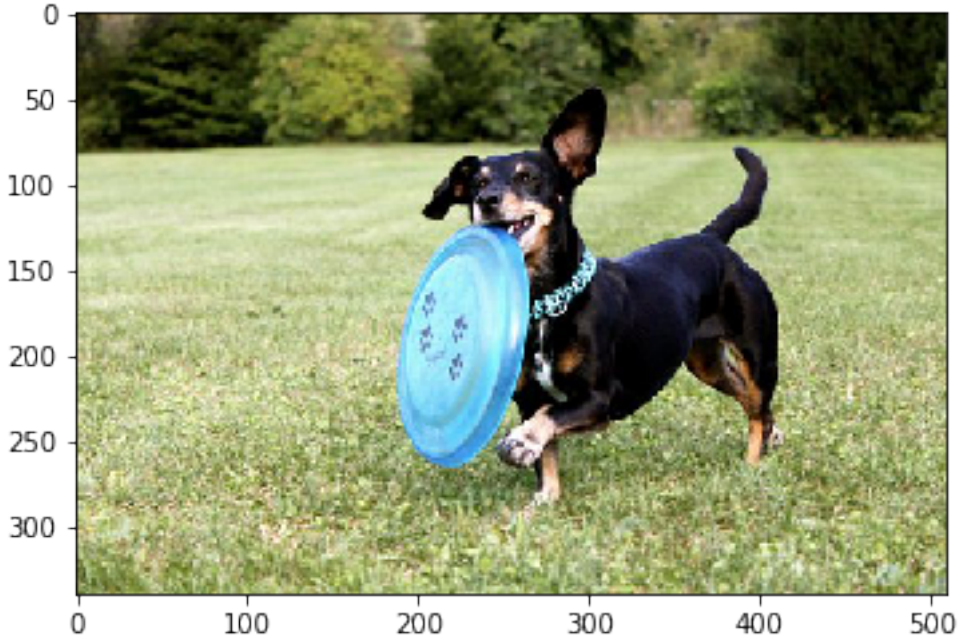
```
hello dog! you look like a German pinscher
```




```
hello dog! you look like a Dalmatian
```

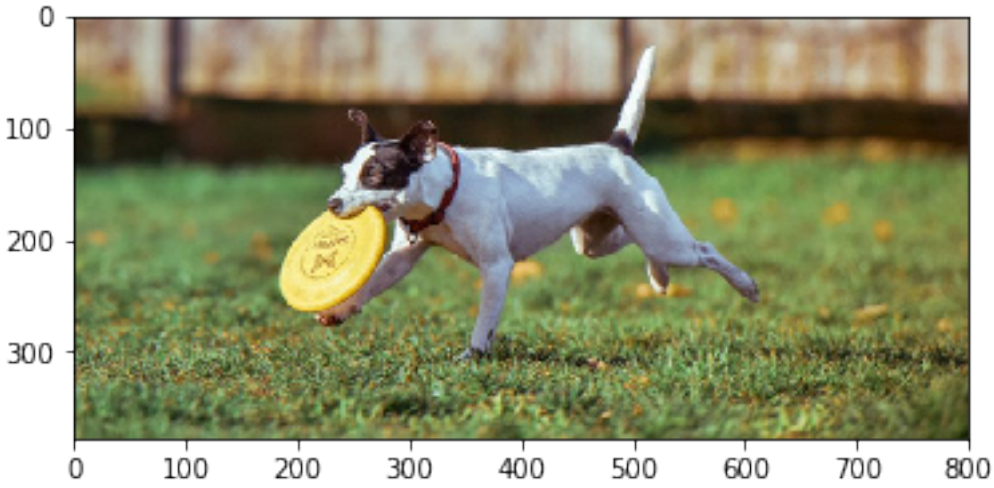


```
hello dog! you look like a Dachshund
```

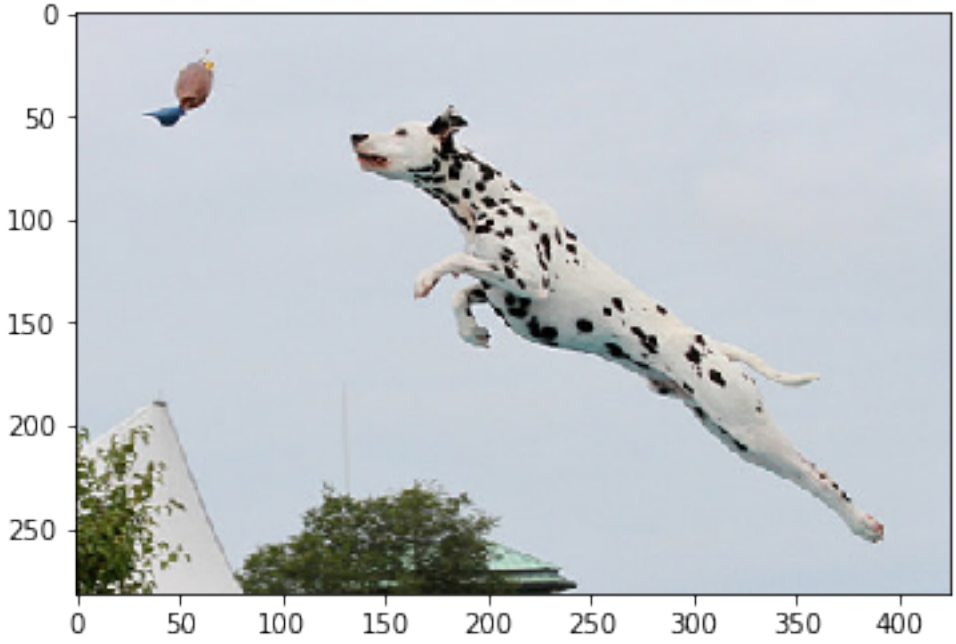



```
hello dog! you look like a Belgian malinois
```

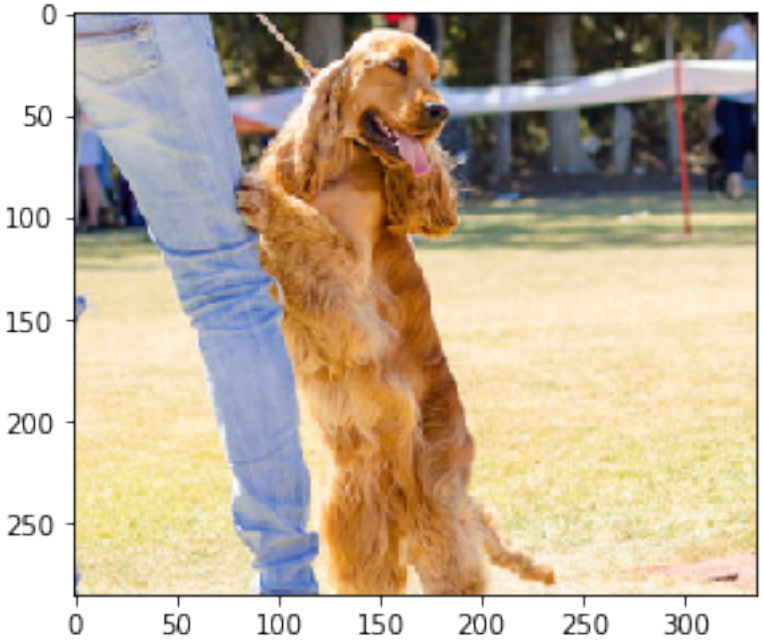
```
hello dog! you look like a Italian greyhound
```



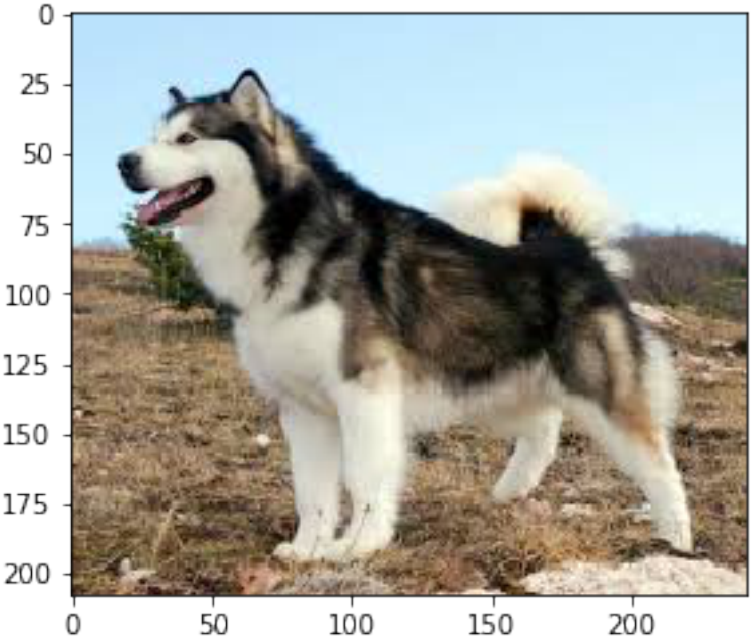

```
hello dog! you look like a Dalmatian
```



```
hello dog! you look like a English cocker spaniel
```

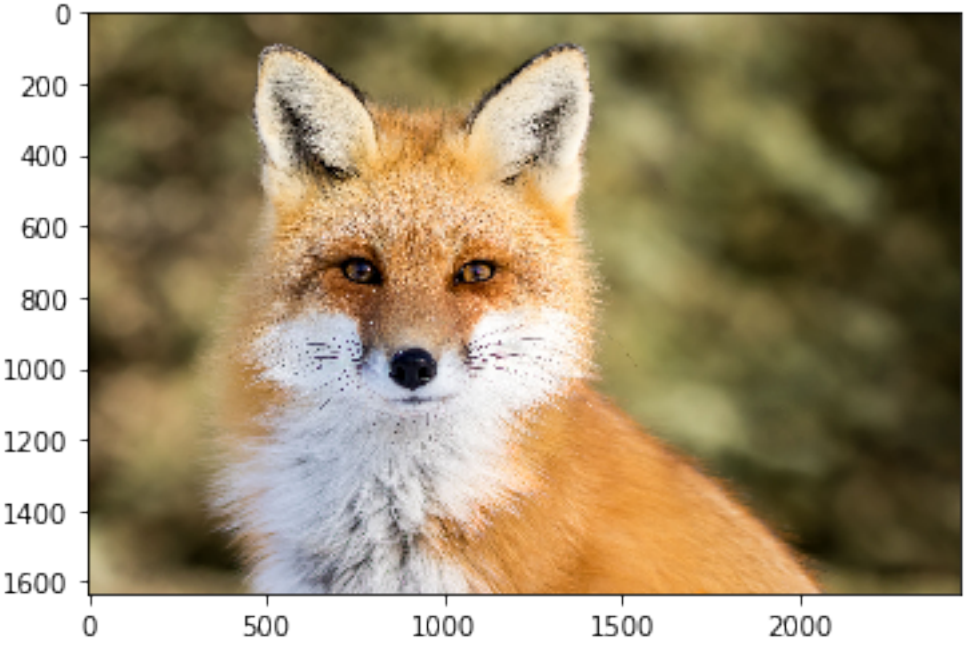



```
hello dog! you look like a Alaskan malamute
```



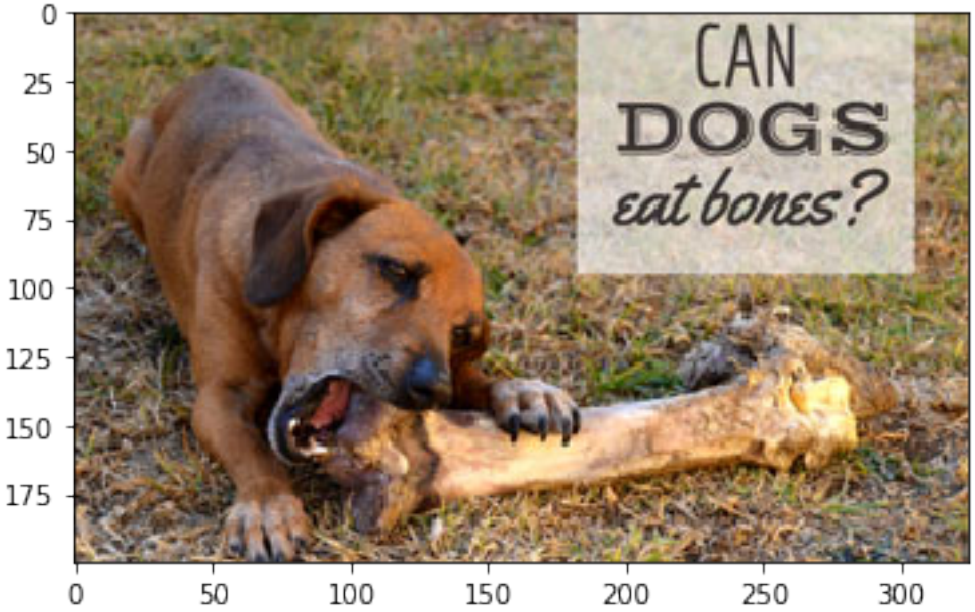
```
hello dog! you look like a Alaskan malamute
```


```
The image is neither human nor dog.
```

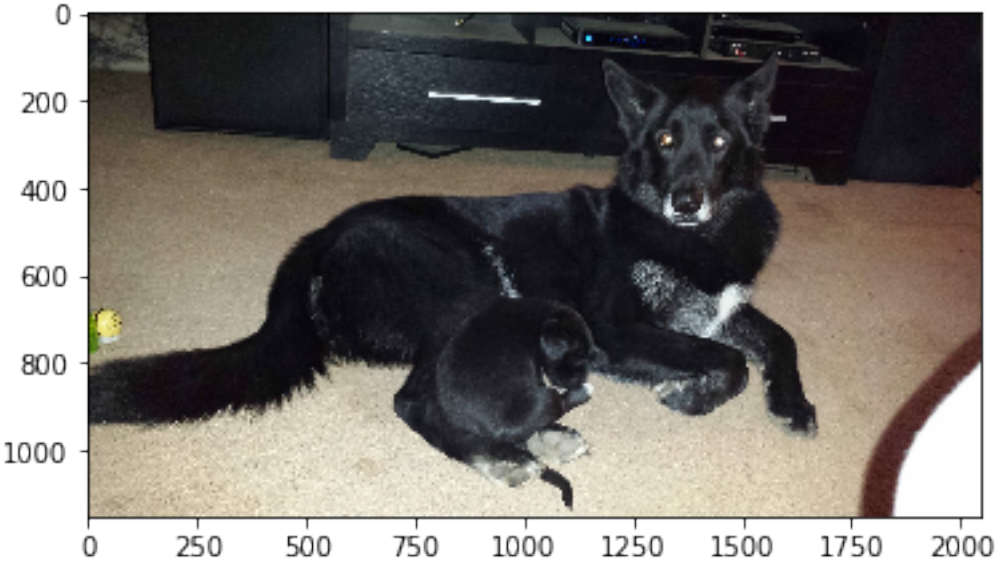


```
hello dog! you look like a Bulldog
```

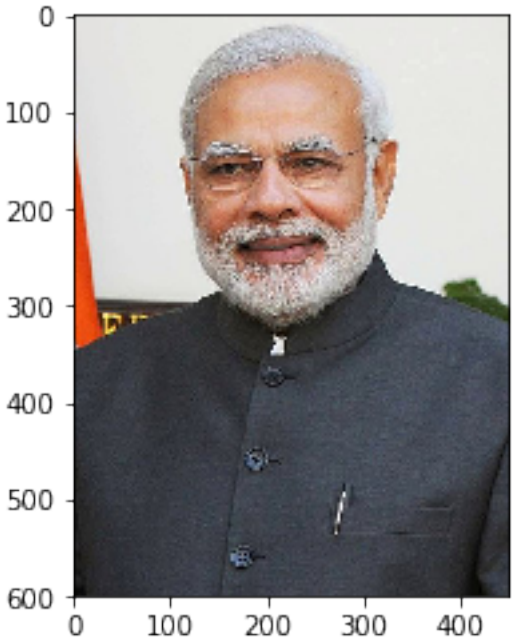

```
hello dog! you look like a Bloodhound
```



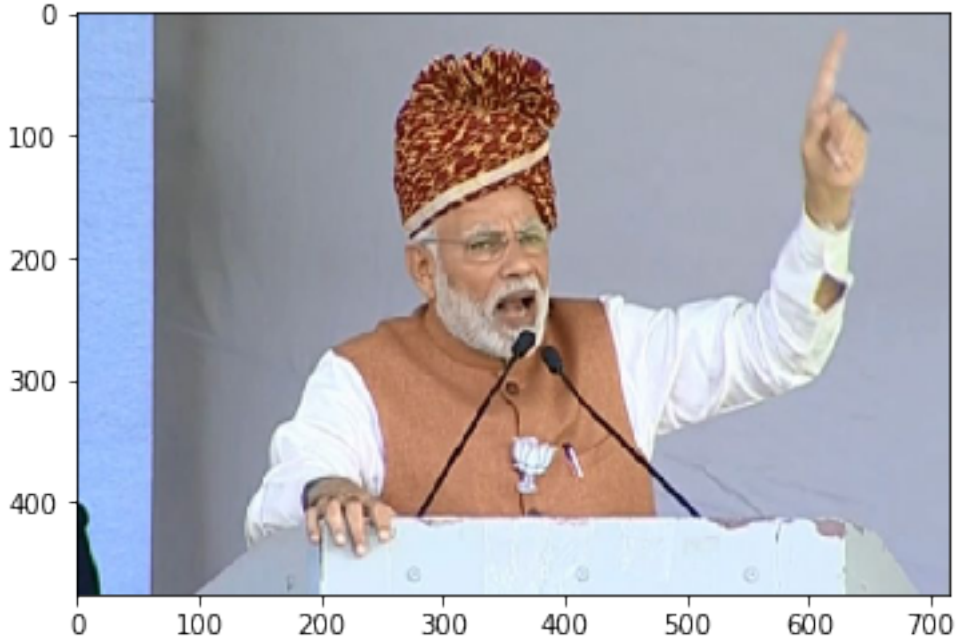
```
hello dog! you look like a Belgian sheepdog
```




```
hello human! you look like a Pharaoh hound
```

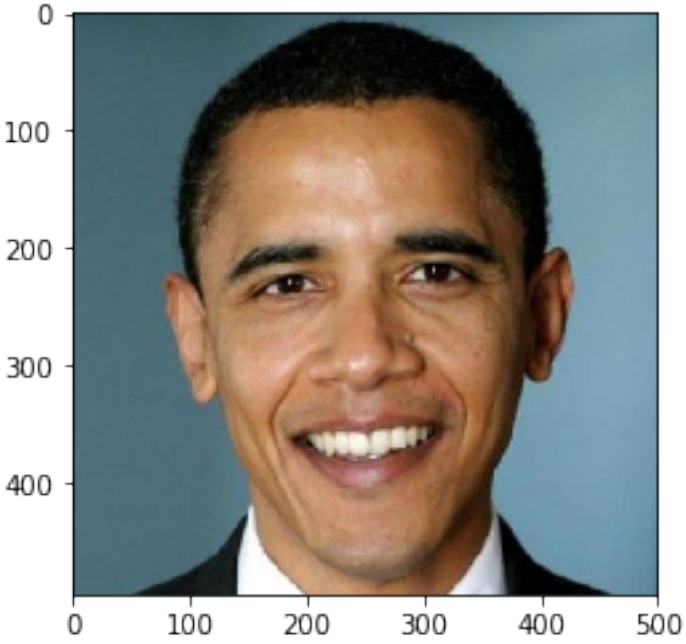


```
hello human! you look like a Poodle
```




```
The image is neither human nor dog.
```

```
hello human! you look like a Akita
```



In [ ]: