dog_app

September 2, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this Jupyter notebook.
Step 0: Import Datasets
Make sure that you've downloaded the required human and dog datasets: * Download the dog dataset. Unzip the folder and place it in this project's home directory, at the location `/dogImages`.

- 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
        import requests
        from glob import glob

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

        # 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

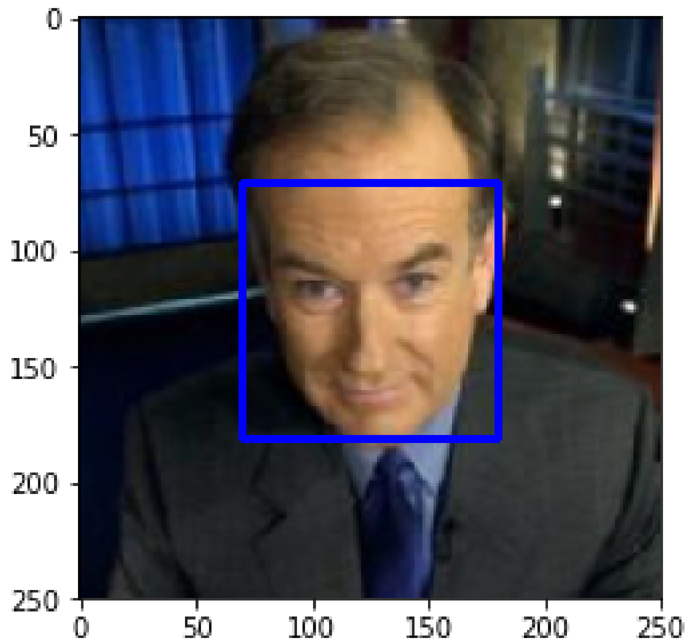
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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[9])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

        # print number of faces detected in the image
        print('Number of faces detected:', len(faces))

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

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

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

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



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
```

```
        img = cv2.imread(img_path)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        faces = face_cascade.detectMultiScale(gray)
        return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

Answer: (You can print out your results and/or write your percentages in this cell)
Human files - detected human faces in 98% of the first 100 images
Dog files - detected human faces in 11% of the first 100 images

In [ ]:

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

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

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

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

        def detected_face_counter(file_lst):
            count = 0
            for img in tqdm(file_lst):
                count += face_detector(img)
            return count

        print("Human files - detected {} human faces in the first {} images".format(
            detected_face_counter(human_files_short), len(human_files_short)))

        print("Dog files - detected {} human faces in the first {} images".format(
            detected_face_counter(dog_files_short), len(dog_files_short)))
```

```
100%|| 100/100 [00:02<00:00, 37.26it/s]
  1%|          | 1/100 [00:00<00:10,  9.36it/s]

Human files - detected 98 human faces in the first 100 images


100%|| 100/100 [00:18<00:00, 10.04it/s]
```

```
Dog files - detected 11 human faces in the first 100 images
```

In [ ]:

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 another face detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs
In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet, a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories.

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:05<00:00, 97892458.97it/s]
```

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

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

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

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

In [7]:
```
from PIL import Image

from torch.autograd import Variable
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

# From PyTorch documentation:
# All pre-trained models expect input images normalized in the same
# way, i.e. mini-batches of 3-channel RGB images of shape (3 x H x W),
# where H and W are expected to be at least 224. The images have to
# be loaded in to a range of [0, 1] and then normalized using
# mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

def preprocess_img(img_path):
    normalize = transforms.Normalize(
        mean=[0.485, 0.456, 0.406],
        std=[0.229, 0.224, 0.225])

    transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        normalize])

    img = Image.open(img_path)
    img = img.convert('RGB')
    img = transform(img)

    # From: https://gist.github.com/jkarimi91/d393688c4d4cdb9251e3f939f138876e
    # PyTorch pretrained models expect the Tensor dims to be (num input imgs, num color
    # Currently however, we have (num color channels, height, width); let's fix this by
    img = img.unsqueeze(0)  # Insert the new axis at index 0

    return Variable(img)
```

In [8]:
```
def VGG16_predict(img_path):
    '''
    Use pre-trained VGG-16 model to obtain index corresponding to
```

```
        predicted ImageNet class for image at specified path

        Args:
            img_path: path to an image

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

        ## TODO: Complete the function.
        ## Load and pre-process an image from the given img_path
        ## Return the *index* of the predicted class for that image

        img = preprocess_img(img_path)

        if use_cuda:
            img = img.cuda()
        output = VGG16(img)
        return torch.argmax(output).data.item() # predicted class index
```

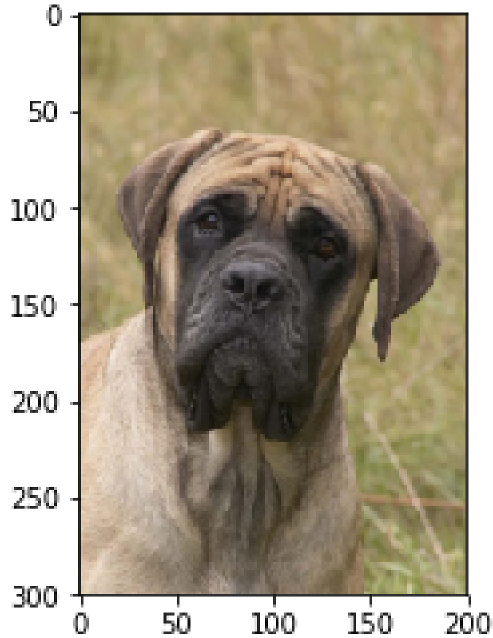
```
In [9]: # output.data.max(1, keepdim=True)[1]
```

1.1.5 Testing VGG16 model

```
In [10]: # Download imagenet labels
         LABELS_URL = 'https://s3.amazonaws.com/mlpipes/pytorch-quick-start/labels.json'

         labels = {int(key):value for (key, value)
                   in requests.get(LABELS_URL).json().items()}
```

```
In [12]: img_path = dog_files_short[0]
         print(img_path)

         pred_idx = VGG16_predict(img_path)
         print(labels[pred_idx])

         img = Image.open(img_path)
         img.thumbnail((300,300))
         plt.imshow(img)
         plt.show()
```

```
dogImages/train/103.Mastiff/Mastiff_06839.jpg
bull mastiff
```



1.1.6 (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 [13]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             pred_idx = VGG16_predict(img_path)
             return 151 <= pred_idx <= 268
```

1.1.7 (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:
Human files - detected dog in 1% of the first 100 images
Dog files - detected dog in 100% of the first 100 images

```
In [14]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         def detection_counter(file_lst, classifier):
             count = 0
             for img in tqdm(file_lst):
                 count += classifier(img)
             return count

         print("Human files - detected {} dogs in the first {} images".format(
             detection_counter(human_files_short, dog_detector),
             len(human_files_short)))

         print("Dog files - detected {} dogs in the first {} images".format(
             detection_counter(dog_files_short, dog_detector),
             len(dog_files_short)))
```

```
100%|| 100/100 [00:03<00:00, 30.45it/s]
  3%|          | 3/100 [00:00<00:03, 27.06it/s]

Human files - detected 0 dogs in the first 100 images


100%|| 100/100 [00:04<00:00, 24.93it/s]

Dog files - detected 100 dogs in the first 100 images
```

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 [15]: ### (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!

1.1.8 (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). 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 [16]: import os
         from torchvision import datasets

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

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

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

         data_dir = 'dogImages'
```

```python
train_dir = os.path.join(data_dir, 'train')
valid_dir = os.path.join(data_dir, 'valid')
test_dir = os.path.join(data_dir, 'test')

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

# Transforms
transform_dct = {
    'train': transforms.Compose([transforms.RandomRotation(25),
                                 transforms.Resize(256),
                                 transforms.RandomResizedCrop(224, scale=(0.8, 1.0)),
                                 transforms.RandomHorizontalFlip(),
                                 transforms.ToTensor(),
                                 normalize]),
    'valid': transforms.Compose([transforms.Resize(256),
                                 transforms.CenterCrop(224),
                                 transforms.ToTensor(),
                                 normalize]),
    'test': transforms.Compose([transforms.Resize(256),
                                transforms.CenterCrop(224),
                                transforms.ToTensor(),
                                normalize])
}

# # Data sets
train_data = datasets.ImageFolder(train_dir, transform=transform_dct['train'])
valid_data = datasets.ImageFolder(valid_dir, transform=transform_dct['valid'])
test_data = datasets.ImageFolder(test_dir, transform=transform_dct['test'])

# 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=False)

# Data loaders dict
loaders_scratch = {
    'train': train_loader,
    'valid': valid_loader,
    'test': test_loader
}
```

In [ ]:

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

Answer:

As prescribed in the PyTorch documentation, we crop the images in the train, validation and test datasets to 224x224 and normalize the images using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225].

The training set is augmented using RandomRotation (rotation), RandomResizedCrop (translation and scale) and RandomHorizontalFlip (flip). The aim of the augmentation is to increase the prediction accuracy by making the model more invariant to differences in object rotation, translation and scale.

1.1.9 (IMPLEMENTATION) Model Architecture

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

In [17]:
```
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
        self.conv1 = nn.Conv2d(3, 16, 3, stride=2, padding=1) # /4
        self.conv2 = nn.Conv2d(16, 32, 3, padding=1) # /2
        self.conv3 = nn.Conv2d(32, 64, 3, padding=1) # /2
        self.conv4 = nn.Conv2d(64, 128, 3, padding=1) # /2
        # Max pooling layer
        self.pool = nn.MaxPool2d(2, 2)
        # Linear layer
        self.fc1 = nn.Linear(7 * 7 * 128, 500)
        self.fc2 = nn.Linear(500, 133)
        # Dropout
        self.dropout = nn.Dropout(0.25)

    def forward(self, x):
        ## Define forward behavior
        x = self.pool(F.relu(self.conv1(x))) # receives 224 x x 3
        x = self.pool(F.relu(self.conv2(x))) # receives 56 x x 16
        x = self.pool(F.relu(self.conv3(x))) # receives 28 x x 32
        x = self.pool(F.relu(self.conv4(x))) # receives 14 x x 64

        x = x.view(-1, 7 * 7 * 128)
```

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

            x = self.dropout(x)
            x = self.fc2(x)
            return x

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

    # instantiate the CNN
    model_scratch = Net()
    print(model_scratch)

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

```
Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), 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))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=6272, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.25)
)
```

In [ ]:

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

Answer:

Initially, my design started as the CNN architecture described in the Udacity classroom. Starting from that architecture, I

- Preserved the (3, 3) kernels and (1, 1) padding for all convolutional layers, because that works fine for the VGGNET model described in the Udacity classroom, and makes it easy to keep track of the down-sampling in the x-y dimensions.
- Preserved the ReLu activations, since no real alternatives were discussed in the classroom and they appear to work well for most architectures.
- Preserved the (2, 2) pooling layers after each convolution layer to reduce the input's x-y dimensions.
- Added a (2, 2) stride in the first convolutional layer to increase the down-sampling in x-y dimensions.

- Added a fourth convolution layer, because this data set and objective (dog breed classification) appear more complex than the data set and objective in the Udacity classroom (MNIST digits).
- Preserved the 2 linear layers, ReLu activations and the (p=0.25) dropout.

1.1.10 (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 [18]: import torch.optim as optim

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

         ### TODO: select optimizer
         optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.05)
```

1.1.11 (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 [19]: # the following import is required for training to be robust to truncated images
         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()
                     ## TODO:
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
```

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

            # clear the gradients of all optimized variables
            optimizer.zero_grad()
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # backward pass: compute gradient of the loss with respect to model paramet
            loss.backward()
            # perform a single optimization step (parameter update)
            optimizer.step()
            # update training loss
            train_loss += ((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()
            ## TODO: update the average validation loss
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model(data)
            # calculate the batch loss
            loss = criterion(output, target)
            # update average validation loss
            valid_loss += ((1 / (batch_idx + 1)) * (loss.data - valid_loss))

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

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

    # return trained model
    return model
```

In [ ]:

In [19]:
```
# train the model
model_scratch = train(20, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```
Epoch: 1         Training Loss: 4.874717        Validation Loss: 4.817525
Validation loss decreased (inf --> 4.817525).  Saving model ...
Epoch: 2         Training Loss: 4.723743        Validation Loss: 4.692999
Validation loss decreased (4.817525 --> 4.692999).  Saving model ...
Epoch: 3         Training Loss: 4.579119        Validation Loss: 4.587366
Validation loss decreased (4.692999 --> 4.587366).  Saving model ...
Epoch: 4         Training Loss: 4.400685        Validation Loss: 4.344599
Validation loss decreased (4.587366 --> 4.344599).  Saving model ...
Epoch: 5         Training Loss: 4.261161        Validation Loss: 4.377175
Epoch: 6         Training Loss: 4.159474        Validation Loss: 4.158762
Validation loss decreased (4.344599 --> 4.158762).  Saving model ...
Epoch: 7         Training Loss: 4.053485        Validation Loss: 4.112990
Validation loss decreased (4.158762 --> 4.112990).  Saving model ...
Epoch: 8         Training Loss: 3.949791        Validation Loss: 4.068662
Validation loss decreased (4.112990 --> 4.068662).  Saving model ...
Epoch: 9         Training Loss: 3.864335        Validation Loss: 4.038600
Validation loss decreased (4.068662 --> 4.038600).  Saving model ...
Epoch: 10         Training Loss: 3.775687        Validation Loss: 4.102492
Epoch: 11         Training Loss: 3.659664        Validation Loss: 3.900174
Validation loss decreased (4.038600 --> 3.900174).  Saving model ...
Epoch: 12         Training Loss: 3.583799        Validation Loss: 3.921677
Epoch: 13         Training Loss: 3.473362        Validation Loss: 3.820579
Validation loss decreased (3.900174 --> 3.820579).  Saving model ...
Epoch: 14         Training Loss: 3.371543        Validation Loss: 3.771619
Validation loss decreased (3.820579 --> 3.771619).  Saving model ...
Epoch: 15         Training Loss: 3.268238        Validation Loss: 3.971390
Epoch: 16         Training Loss: 3.153470        Validation Loss: 3.776736
Epoch: 17         Training Loss: 3.061381        Validation Loss: 3.798499
Epoch: 18         Training Loss: 2.954467        Validation Loss: 3.752123
Validation loss decreased (3.771619 --> 3.752123).  Saving model ...
Epoch: 19         Training Loss: 2.847306        Validation Loss: 3.689503
Validation loss decreased (3.752123 --> 3.689503).  Saving model ...
Epoch: 20         Training Loss: 2.734679        Validation Loss: 3.776585
```

1.1.12 (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 [20]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

```
In [21]: 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 [22]: # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.713265


Test Accuracy: 14% (119/836)
```

```
In [ ]:
```

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.

1.1.13 (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 [23]:
```
## TODO: Specify data loaders

# EXACT COPY FROM SECTION 3

loaders_transfer = loaders_scratch.copy()
```

1.1.14 (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 [24]:
```
import torchvision.models as models
import torch.nn as nn

n_classes = 133

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

# Freeze the pre-trained feature weights
for param in model_transfer.features.parameters():
    param.requires_grad = False

# add last linear layer (n_inputs -> 133 dog breed classes)
# new layers automatically have requires_grad = True
n_inputs = model_transfer.classifier[6].in_features
last_layer = nn.Linear(n_inputs, n_classes)

model_transfer.classifier[6] = last_layer

# check to see that your last layer produces the expected number of outputs
print(model_transfer.classifier[6].out_features)

# print out the model structure
print(model_transfer)

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

```
133
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)
  )
)
```

```
In [ ]:
```

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:

Note that: - The new data set is reasonably small (8,351 records) - The new data is very similar to the original training data of the VGG16 model

Because the data sets are similar, it seems reasonable to expect that the high level features of the images will be similar. Hence: - We keep all layers except the last layer of the network intact - We freeze the pre-trained weights in order to prevent overfitting on the small data set - For the last layer, we add a new linear layer with randomized weights and 133 output features, which is the number of dog breed classes in our data set

1.1.15 (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 [25]: import torch.optim as optim

         criterion_transfer = nn.CrossEntropyLoss()
         optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(), lr=0.001)
```

1.1.16 (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 [27]: # train the model
         n_epochs = 20
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,

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

```
Epoch: 1         Training Loss: 3.880616         Validation Loss: 2.362100
Validation loss decreased (inf --> 2.362100).  Saving model ...
Epoch: 2         Training Loss: 1.878302         Validation Loss: 1.014218
Validation loss decreased (2.362100 --> 1.014218).  Saving model ...
Epoch: 3         Training Loss: 1.118724         Validation Loss: 0.691332
Validation loss decreased (1.014218 --> 0.691332).  Saving model ...
Epoch: 4         Training Loss: 0.884951         Validation Loss: 0.570887
Validation loss decreased (0.691332 --> 0.570887).  Saving model ...
Epoch: 5         Training Loss: 0.740057         Validation Loss: 0.510260
Validation loss decreased (0.570887 --> 0.510260).  Saving model ...
Epoch: 6         Training Loss: 0.661237         Validation Loss: 0.481228
Validation loss decreased (0.510260 --> 0.481228).  Saving model ...
Epoch: 7         Training Loss: 0.621918         Validation Loss: 0.452288
Validation loss decreased (0.481228 --> 0.452288).  Saving model ...
Epoch: 8         Training Loss: 0.559375         Validation Loss: 0.423670
Validation loss decreased (0.452288 --> 0.423670).  Saving model ...
Epoch: 9         Training Loss: 0.532043         Validation Loss: 0.414652
Validation loss decreased (0.423670 --> 0.414652).  Saving model ...
```

```
Epoch: 10 	Training Loss: 0.489670 	Validation Loss: 0.397938
Validation loss decreased (0.414652 --> 0.397938).  Saving model ...
Epoch: 11 	Training Loss: 0.479517 	Validation Loss: 0.400576
Epoch: 12 	Training Loss: 0.442066 	Validation Loss: 0.399123
Epoch: 13 	Training Loss: 0.430490 	Validation Loss: 0.387156
Validation loss decreased (0.397938 --> 0.387156).  Saving model ...
Epoch: 14 	Training Loss: 0.425088 	Validation Loss: 0.374426
Validation loss decreased (0.387156 --> 0.374426).  Saving model ...
Epoch: 15 	Training Loss: 0.399616 	Validation Loss: 0.375839
Epoch: 16 	Training Loss: 0.390003 	Validation Loss: 0.377780
Epoch: 17 	Training Loss: 0.367822 	Validation Loss: 0.363450
Validation loss decreased (0.374426 --> 0.363450).  Saving model ...
Epoch: 18 	Training Loss: 0.366486 	Validation Loss: 0.374237
Epoch: 19 	Training Loss: 0.333833 	Validation Loss: 0.360343
Validation loss decreased (0.363450 --> 0.360343).  Saving model ...
Epoch: 20 	Training Loss: 0.325299 	Validation Loss: 0.363793
```

In [ ]:

In [28]:
```python
# load the model that got the best validation accuracy (uncomment the line below)
model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.17 (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 [29]:
```python
test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.422128


Test Accuracy: 87% (729/836)
```

1.1.18 (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 [30]:
```python
# list of class names by index, i.e. a name can be accessed like class_names[0]
# class_names = [item[4:].replace("_", " ") for item in data_transfer['train'].classes]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]
```

In [31]:
```python
class_names[:5]
```

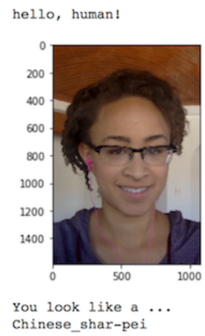
Out[31]:
```
['Affenpinscher',
 'Afghan hound',
```

```
 'Airedale terrier',
 'Akita',
 'Alaskan malamute']
```

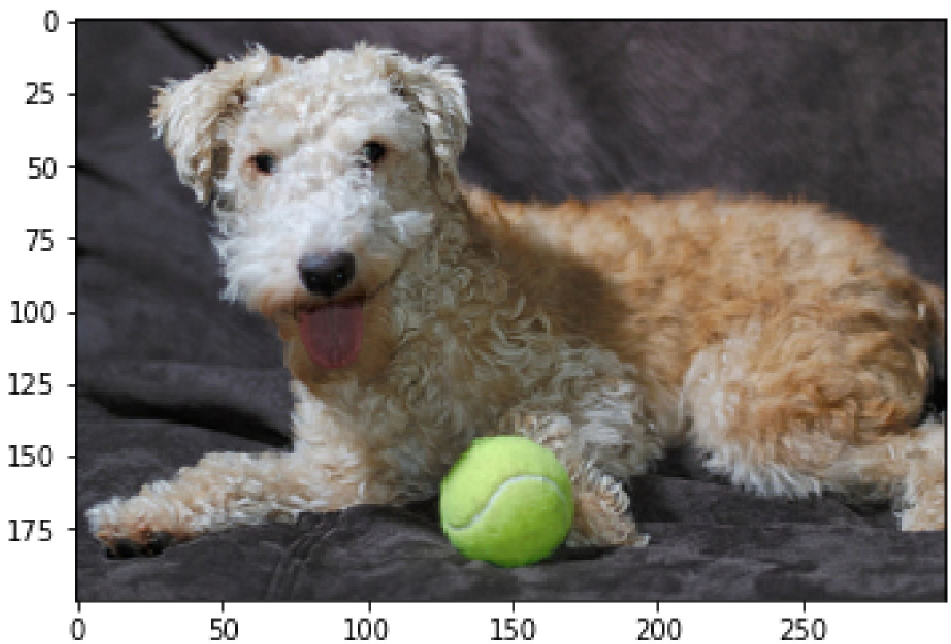
In [32]:
```python
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

def preprocess_img_transfer(img_path):
    img = Image.open(img_path).convert('RGB')

    transform = transform_dct['test']
    img = transform(img).unsqueeze(0)

    return Variable(img)

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed
    img = preprocess_img_transfer(img_path)
    if use_cuda:
        img = img.cuda()

    output = model_transfer(img)
    pred_idx = torch.argmax(output).data.item() # predicted class index
    return class_names[pred_idx], pred_idx
```

In [33]:
```python
human_filepath = human_files_short[10]
dog_filepath = dog_files_short[99]

img_path = dog_filepath

print("Predicted dog breed: {}".format(predict_breed_transfer(img_path)[0]))
print(img_path)

img = Image.open(img_path)
img.thumbnail((300,300))
plt.imshow(img)
plt.show()
```

```
Predicted dog breed: Lakeland terrier
dogImages/train/097.Lakeland_terrier/Lakeland_terrier_06501.jpg
```



Sample Human Output



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 `dog_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!

1.1.19 (IMPLEMENTATION) Write your Algorithm

```
In [34]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

         def run_app(img_path):
             img = Image.open(img_path)
             img.thumbnail((300,300))
             plt.imshow(img)
             plt.show()

             if dog_detector(img_path) or face_detector(img_path):
                 pred_breed, pred_idx = predict_breed_transfer(img_path)
                 # Show a random image of the predicted breed (from the train data set)
                 pred_breed_folder_path = os.path.join('dogImages', 'train', train_data.classes[
                 pred_breed_img_path = np.random.choice(np.array(glob(pred_breed_folder_path)))

                 if dog_detector(img_path):
                     print("Woof-Woof! \nThis dog looks like a... {}!".format(pred_breed))
                 else:
                     print("Hello there! \nThis human is most similar to a... {}!".format(pred_b
                 pred_breed_img = Image.open(pred_breed_img_path)
                 pred_breed_img.thumbnail((300,300))
                 plt.imshow(pred_breed_img)
                 plt.show()
             else:
                 print("No dog or human was detected! \nPlease provide a new image and try again

             print("**********************************************************************")
             ## handle cases for a human face, dog, and neither
             return None

In [ ]:
```

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?

1.1.20 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: (Three possible points for improvement)

The dog breed classification accuracy on the test set (87%) is a lot better than I expected! The algorithm also classifies 5/5 dog breeds correct for the dog images that I added.

The 'face_detector' does misqualify a monkey as a human. However, given that we are closely related to monkeys, that is not totally unexpected.

When investigating the behavior of the algorithm on images of humans, it was interesting to learn that **A LOT of politicians look like Dogue de Bordeaux dogs!**

Finally, here are some avenues that can be explored to further improve the performance of the algorithm:

1. Hyperparameter optimization (optimizer, learning rate, weight initialization, architecture choices)
2. Find more dog pictures to increase the size of the data set
3. Use augmentation techniques to increase the size of the data set
4. Try different classification models. Investigate if using an ensemble increases performance.

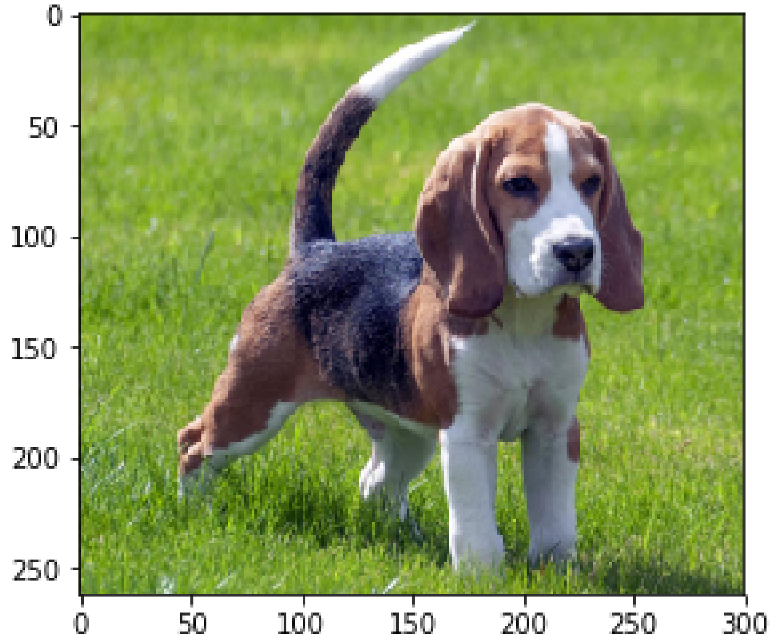
In [ ]:

In [36]:
```
# ## TODO: Execute your algorithm from Step 6 on
# ## at least 6 images on your computer.
# ## Feel free to use as many code cells as needed.

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

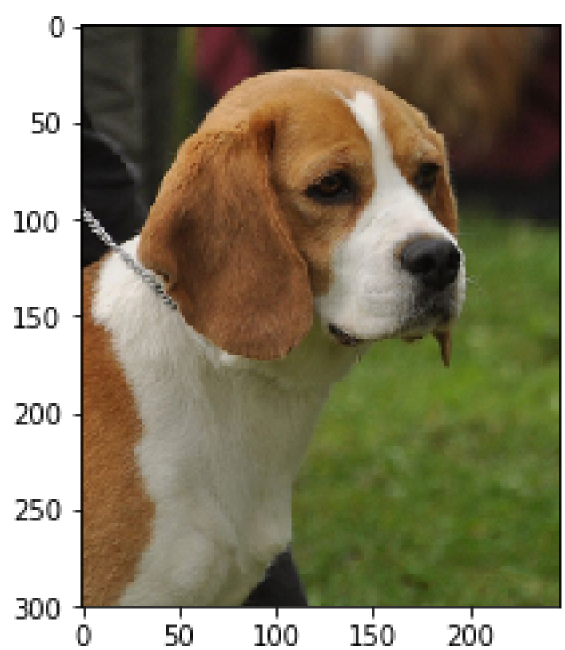
In [ ]:
```
# for img_file in os.listdir('./images'):
#     img_path = os.path.join('./images', img_file)
#     try:
#         run_app(img_path)
#     except:
#         next # catch .ipynb_checkpoints file
```

In [35]:
```
dir_path = './my_images/dogs'

for img_file in os.listdir(dir_path):
    if img_file == '.ipynb_checkpoints':
        next
    else:
        print(img_file)
        img_path = os.path.join(dir_path, img_file)
        run_app(img_path)
```

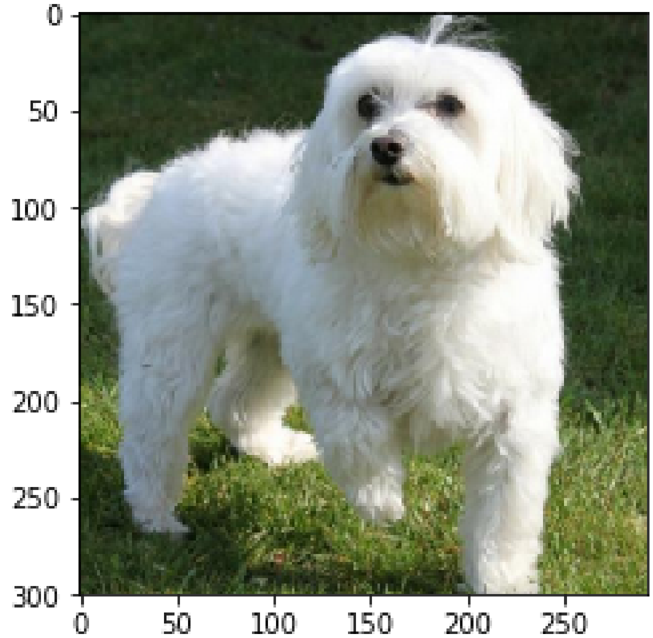
```
beagle.png
```



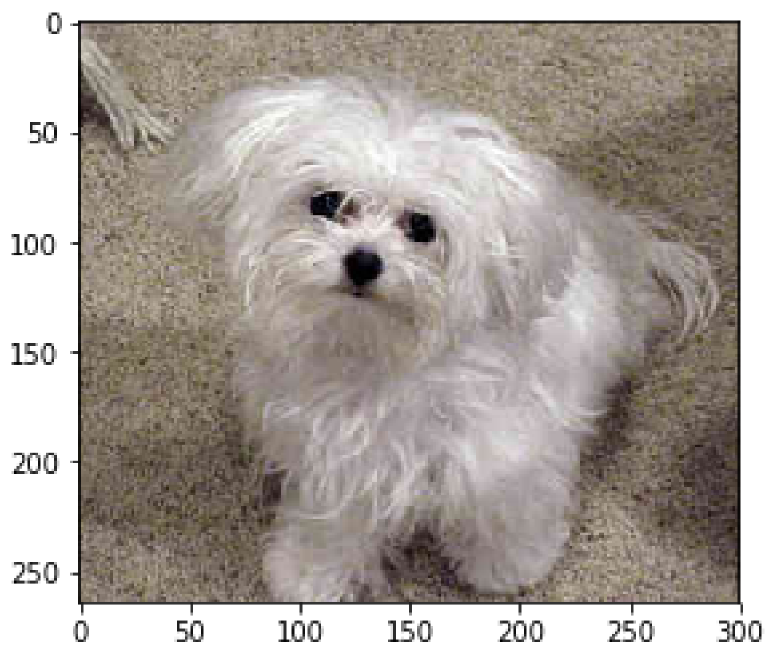

```
Woof-Woof!
This dog looks like a... Beagle!
```



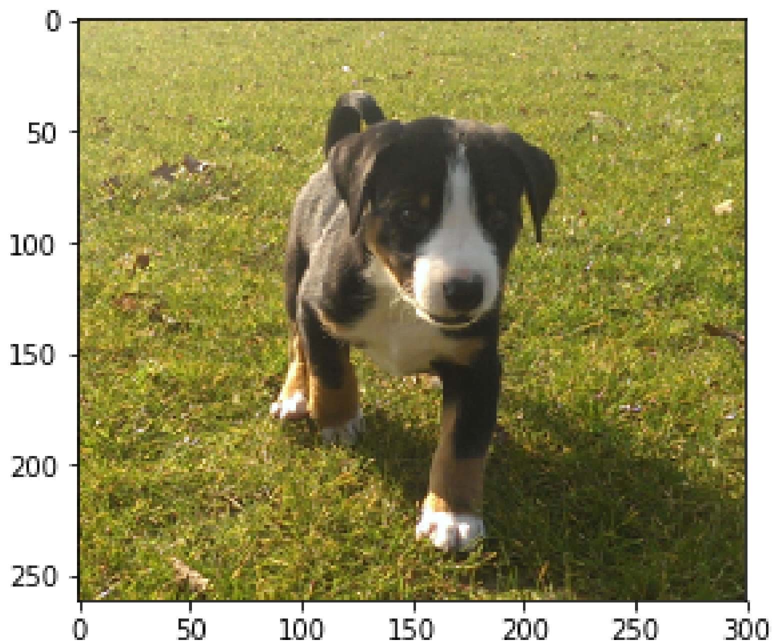
****
maltese.png



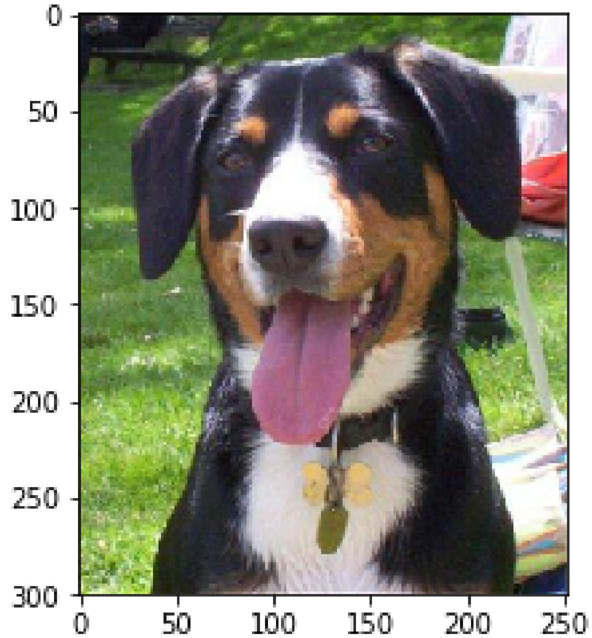
Woof-Woof!
This dog looks like a... Maltese!



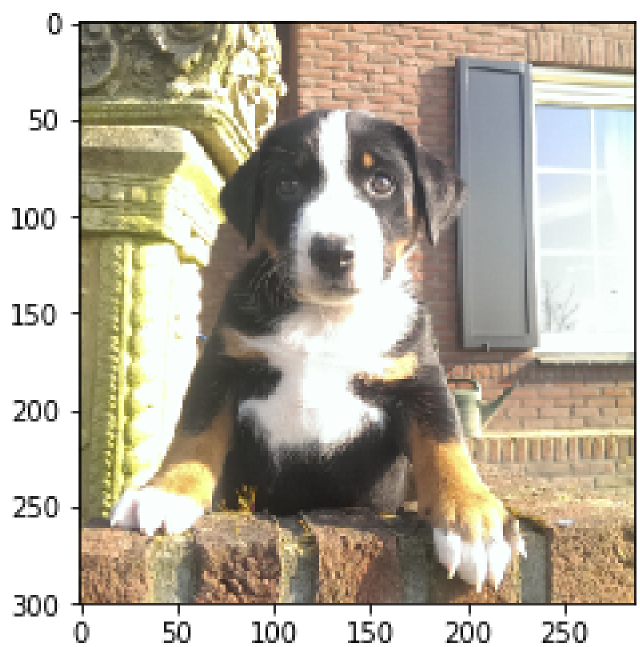
```
*********************************************************************
my_image_14.png
```



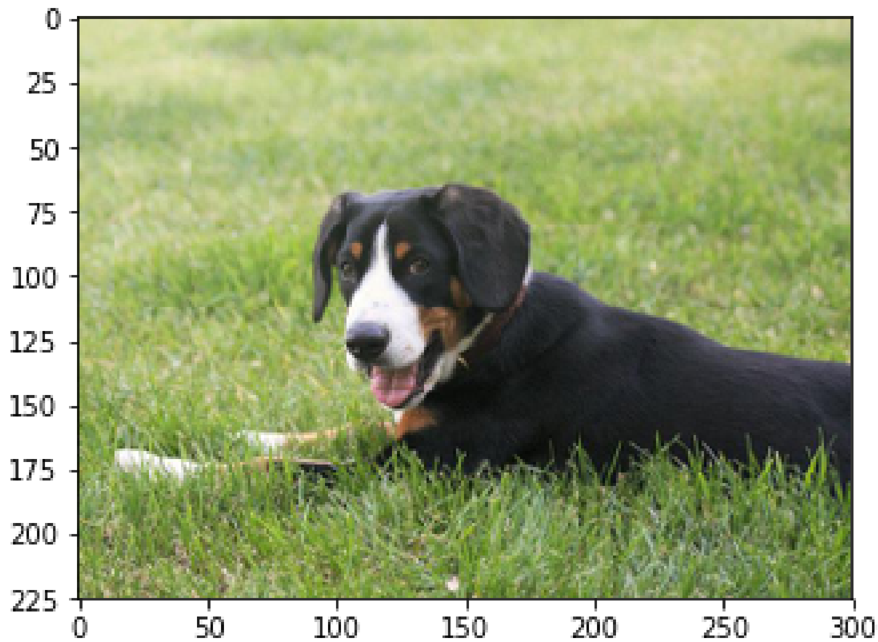

```
Woof-Woof!
This dog looks like a... Entlebucher mountain dog!
```



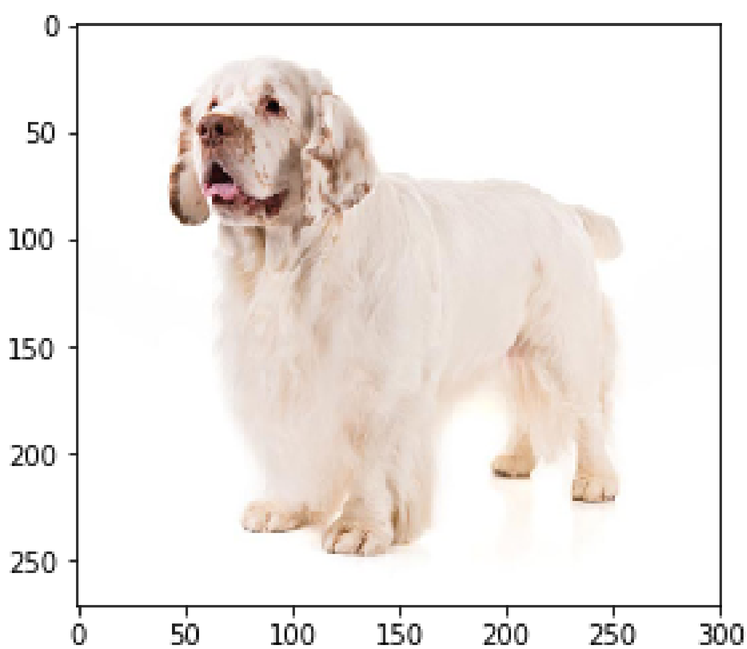

```
*****************************************************************
my_image_13.png
```



```
Woof-Woof!
This dog looks like a... Entlebucher mountain dog!
```




```
*******************************************************************
Clumber Spaniel.png
```



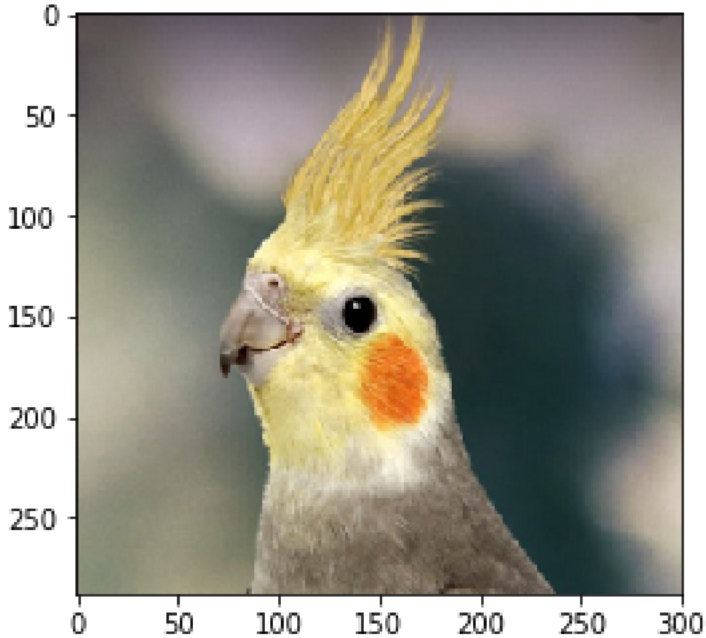
```
Woof-Woof!
This dog looks like a... Clumber spaniel!
```


```
************************************************************************
```

In [ ]:

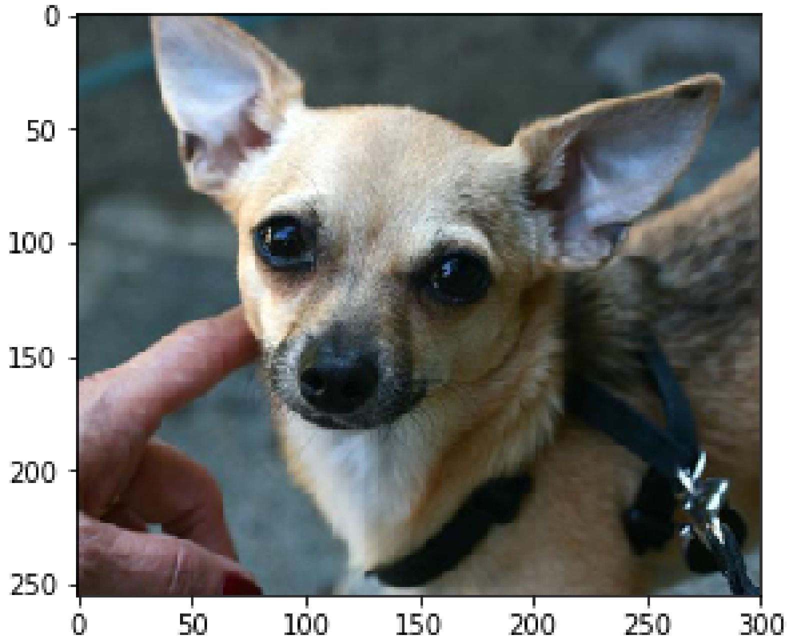
In [36]:
```python
dir_path = './my_images/neither'

for img_file in os.listdir(dir_path):
    if img_file == '.ipynb_checkpoints':
        next
    else:
        print(img_file)
        img_path = os.path.join(dir_path, img_file)
        run_app(img_path)
```

```
my_bird.png
```




```
No dog or human was detected!
Please provide a new image and try again.
*********************************************************************
my_image_4.png
```

```
Hello there!
This human is most similar to a... Chihuahua!
```

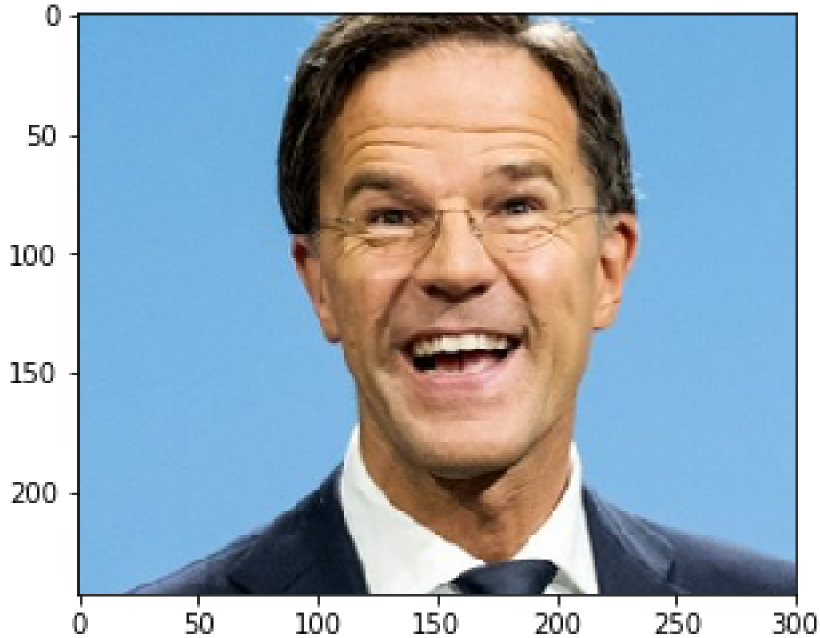


```
**********************************************************************
```

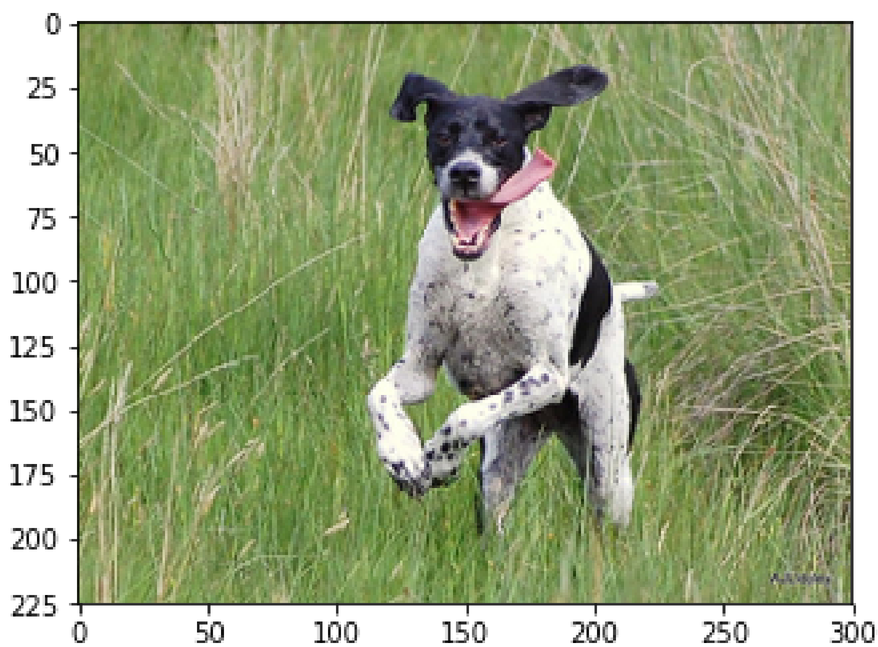
In [ ]:

In [37]:
```python
dir_path = './my_images/humans'

for img_file in os.listdir(dir_path):
    if img_file == '.ipynb_checkpoints':
        next
    else:
        print(img_file)
        img_path = os.path.join(dir_path, img_file)
        run_app(img_path)
```

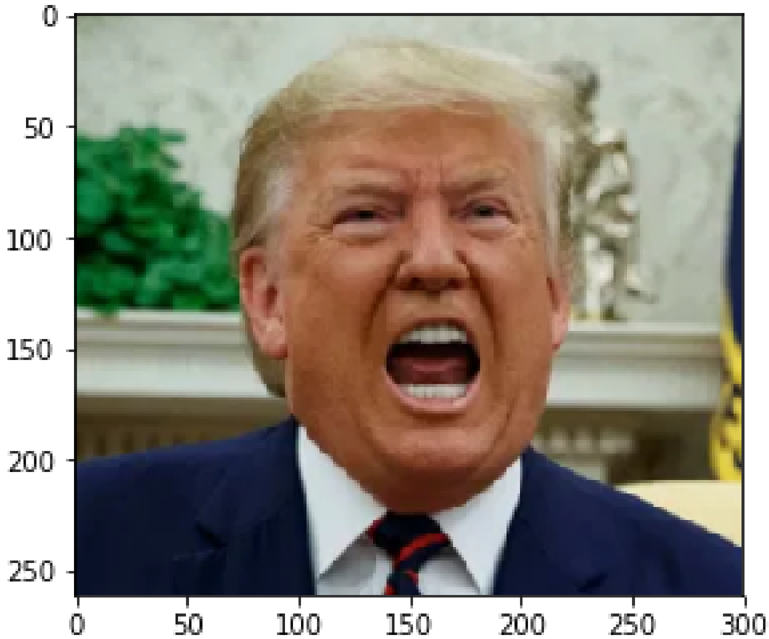
```
my_human_6.png
```



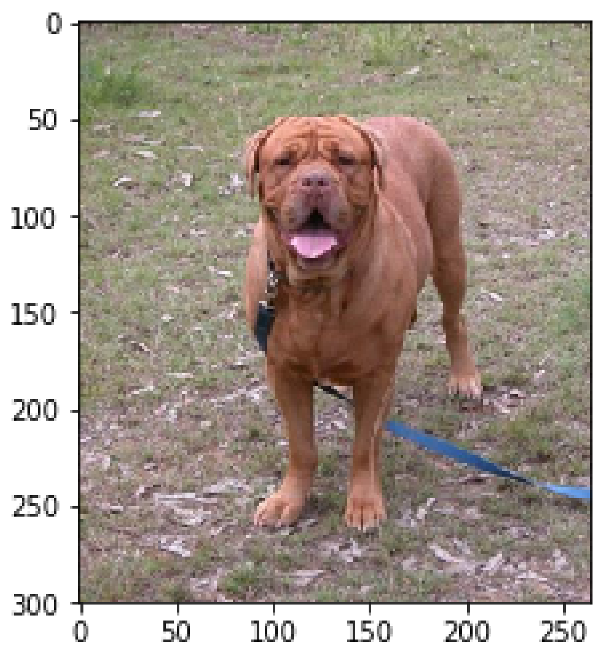

```
Hello there!
This human is most similar to a... Pointer!
```



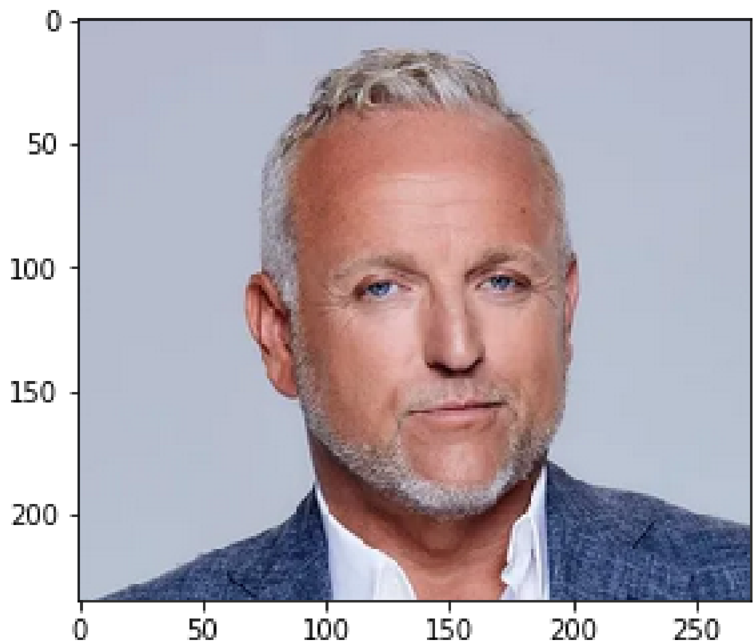
```
*********************************************************************
my_human_5.png
```



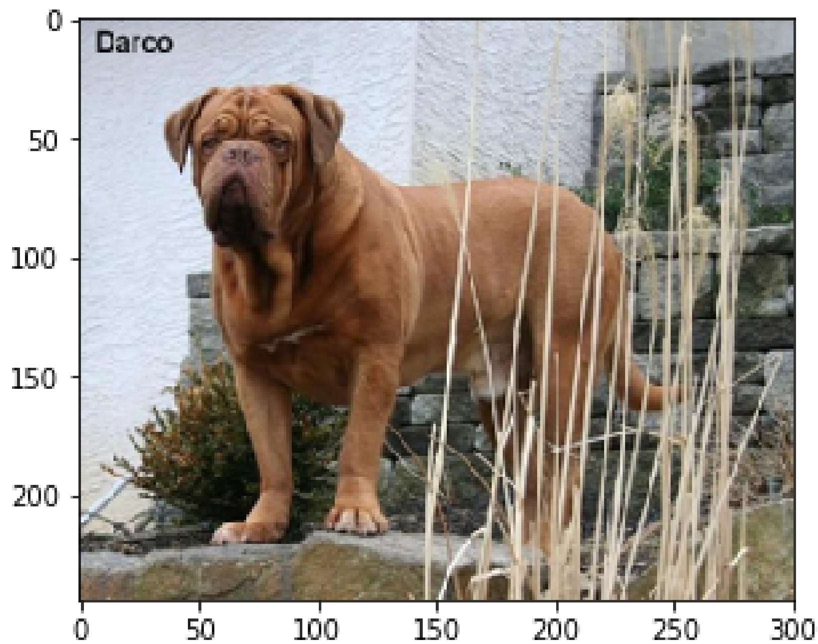

```
Hello there!
This human is most similar to a... Dogue de bordeaux!
```



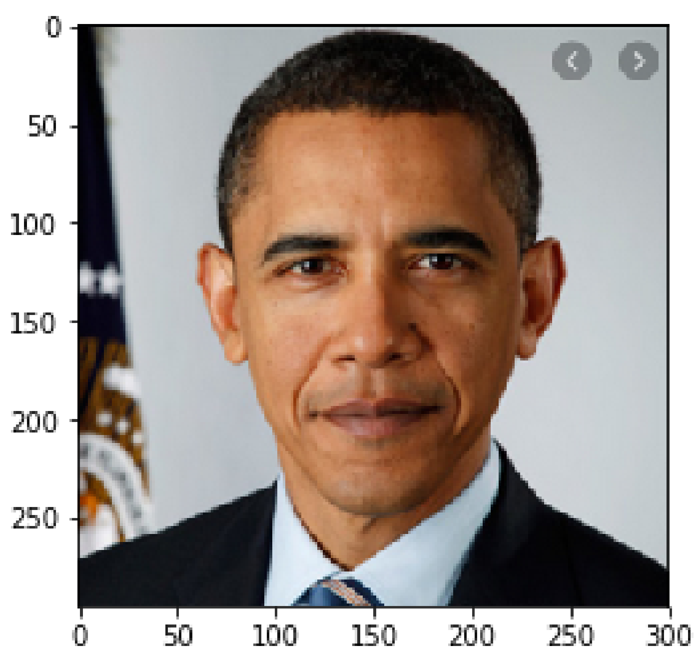
****
my_human_1.png



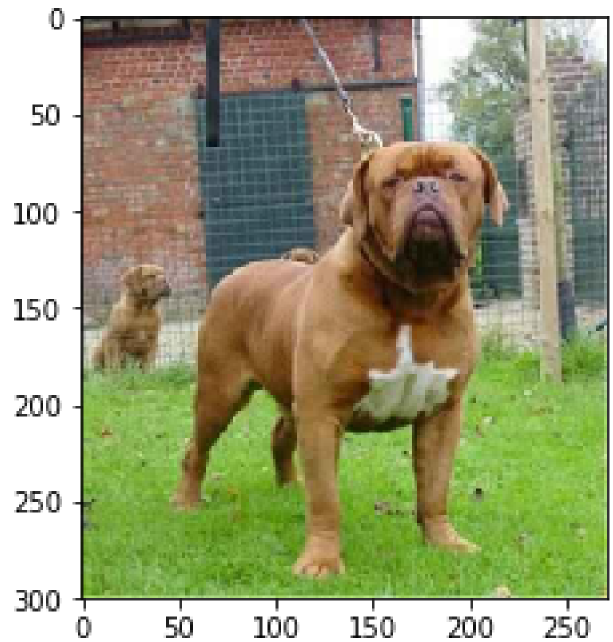
Hello there!
This human is most similar to a... Dogue de bordeaux!



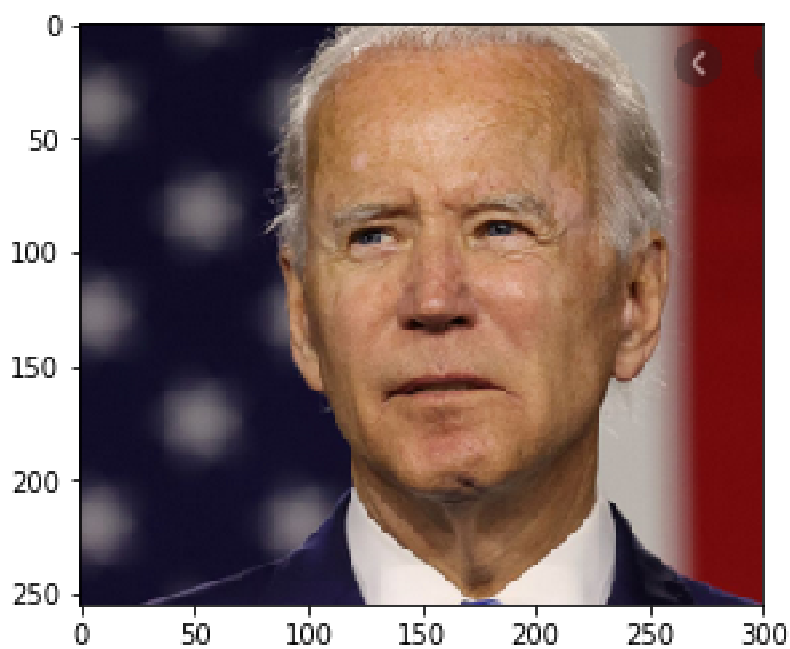

```
*********************************************************************
my_human_2.png
```



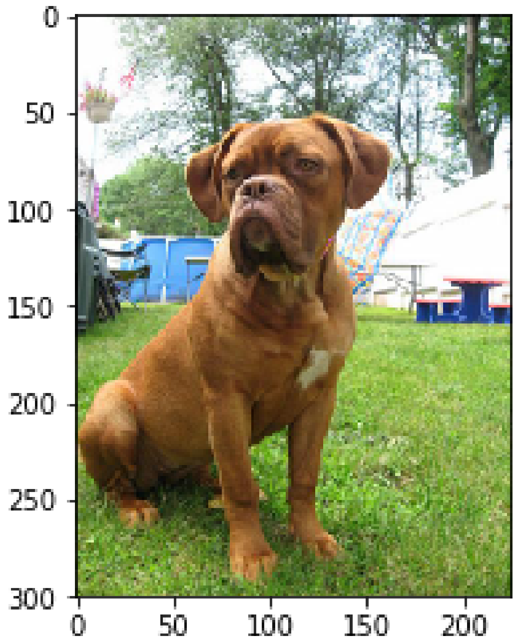
```
Hello there!
This human is most similar to a... Dogue de bordeaux!
```




```
*********************************************************************
my_human_4.png
```

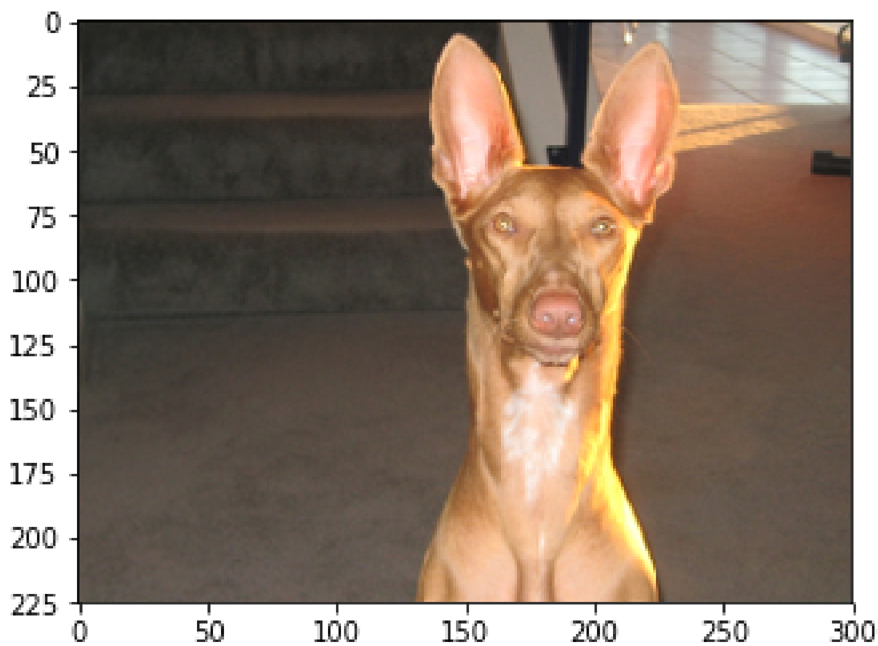


```
Hello there!
This human is most similar to a... Dogue de bordeaux!
```




```
******************************************************************
my_human_3.png
```


```
Hello there!
This human is most similar to a... Pharaoh hound!
```



```
**********************************************************************
```

```
In [ ]:
```