dog_app

May 5, 2020

1 Convolutional Neural Networks

1.1 Project: Write an Algorithm for a Dog Identification App

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

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

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

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

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

Step 0: Import Datasets

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

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

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

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

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

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

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

        # print number of images in each dataset
        print('There are %d total human images.' % len(human_files))
        print('There are %d total dog images.' % len(dog_files))
```

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

Step 1: Detect Humans

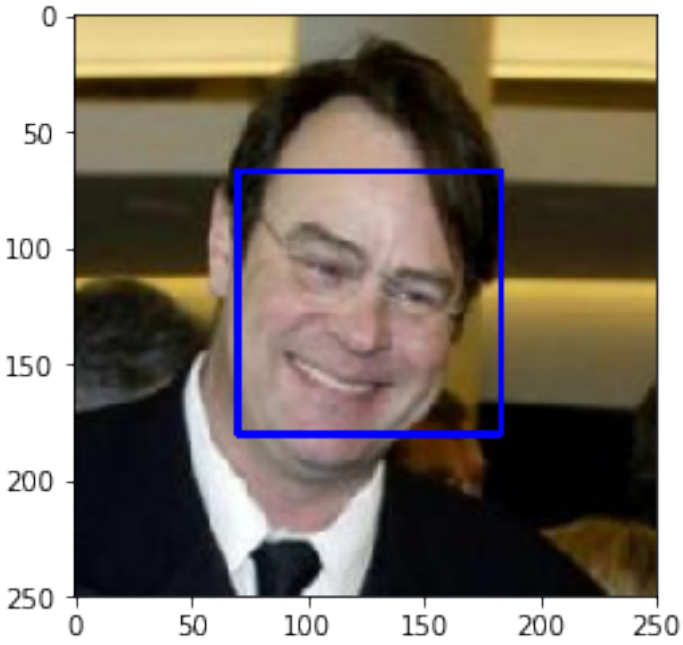
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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

```
In [15]: 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 [10]: # returns "True" if face is detected in image stored at img_path
         def face_detector(img_path):
             img = cv2.imread(img_path)
             gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
             faces = face_cascade.detectMultiScale(gray)
             return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

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

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

        human_images_detected_h = 0
        dog_images_detected_h = 0

        for i in range(len(human_files_short)):
            if face_detector(human_files_short[i]) is True:
                human_images_detected_h += 1
        for k in range(len(dog_files_short)):
            if face_detector(dog_files_short[i]) is True:
                dog_images_detected_h += 1

        print(human_images_detected_h,"%", "of the first 100 images in human_files have a detect
        print(dog_images_detected_h,"%", "of the first 100 images in dog_files have a detected h
```

```
98 % of the first 100 images in human_files have a detected human face.
0 % of the first 100 images in dog_files have a detected human face.
```

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 [11]: 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, 98167701.72it/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 [12]: from PIL import Image
         import torchvision.transforms as transforms

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

             Args:
                 img_path: path to an image

             Returns:
                 Index corresponding to VGG-16 model's prediction
             '''
             ## TODO: Complete the function.
             ## Load and pre-process an image from the given img_path
             ## Return the *index* of the predicted class for that image

             # Loading an image from the given img_path
             image = Image.open(img_path)

             # Define the pre-processing steps
             tensor_transform = transforms.Compose([transforms.Resize(255),
                                                    transforms.CenterCrop(224),
                                                    transforms.ToTensor(),
                                                    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                         std=[0.229, 0.224, 0.225])
                                                   ])

             # Transform the image into an image tensor and add a dimension
             image_tensor = tensor_transform(image)
             image_tensor = image_tensor.unsqueeze(0)

             # Moving to GPU if possible
             if use_cuda:
                 image_tensor = image_tensor.cuda()

             # Turned off gradients and backprob and predict with VGG-16
             with torch.no_grad():
                 output = VGG16(image_tensor)
                 prediction = torch.argmax(output)

             return prediction # predicted class index
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary, you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all

categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained VGG-16 model, we need only check if the pre-trained model predicts an index between 151 and 268 (inclusive).

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

```
In [13]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):

             ## TODO: Complete the function.
             predicted_index = VGG16_predict(img_path)

             return True if 151 <= predicted_index <= 268 else False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:

```
In [9]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        human_images_detected_d = 0
        dog_images_detected_d = 0

        for i in range(len(human_files_short)):
            if dog_detector(human_files_short[i]) is True:
                human_images_detected_d += 1
            if dog_detector(dog_files_short[i]) is True:
                dog_images_detected_d += 1

        print(human_images_detected_d,"%", "of the first 100 images in human_files have a detect
        print(dog_images_detected_d,"%", "of the first 100 images in dog_files have a detected d
```

```
1 % of the first 100 images in human_files have a detected dog face.
100 % of the first 100 images in dog_files have a detected dog face.
```

We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [10]: ### (Optional)
         ### TODO: Report the performance of another pre-trained network.
         ### Feel free to use as many code cells as needed.
```

Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 10%. In Step 4 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have trouble distinguishing between a Brittany and a Welsh Springer Spaniel.

Brittany	Welsh Springer Spaniel

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel

Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador	Chocolate Labrador

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

1.1.7 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate data loaders for the training, validation, and test datasets of dog images (located at `dog_images/train`, `dog_images/valid`, and `dog_images/test`, respectively). You may find this documentation on custom datasets to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of transforms!

```
In [11]: import os
         import torch
         from torchvision import datasets, transforms
```

```
### TODO: Write data loaders for training, validation, and test sets

## Specify appropriate transforms, and batch_sizes
batch_size = 64

# Define transforms for data augmentation and test_transform
train_transforms = transforms.Compose([transforms.Resize(255),
                                       transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomVerticalFlip(),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                            std=[0.229, 0.224, 0.225])
                                      ])

test_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                           std=[0.229, 0.224, 0.225])
                                     ])

train_data = datasets.ImageFolder(root='/data/dog_images/train', transform = train_tran
valid_data = datasets.ImageFolder(root='/data/dog_images/valid', transform = test_trans
test_data = datasets.ImageFolder(root='/data/dog_images/test', transform = test_transfo

trainloader = torch.utils.data.DataLoader(train_data, batch_size = 64, shuffle=True)
validloader = torch.utils.data.DataLoader(valid_data, batch_size = 64, shuffle=True)
testloader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=True)
```

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

Answer:

1. I first resize the input images to 255 Œ 255, then I use "RandomSizeCrop" to make the images become 224 Œ 224, because the authors of the VGG16 network claims that "During training, the input to our ConvNets is a fixed-size 224 Œ 224 RGB image."

2. For preprocessing the training data, I utilized various methods of data augmentation. For instance, I use Random Vertical Flip, Random Horizontal Flip and RandomRotation at 30 degrees to change the orientations of the images.

1.1.8 (IMPLEMENTATION) Model Architecture

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

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


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

# define the CNN architecture
class Net(nn.Module):
    ### TODO: choose an architecture, and complete the class
    def __init__(self):
        super(Net, self).__init__()

        hidden_1 = 500

        ## Define layers of a CNN
        self.conv1 = nn.Conv2d(3, 32, 3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, 3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, 3, padding=1)
        self.conv4 = nn.Conv2d(128, 256, 3, padding=1)
        #self.conv5 = nn.Conv2d(256, 512, 3, padding=1)

        self.pool = nn.MaxPool2d(2,2)

        self.bn1 = nn.BatchNorm2d(32)
        self.bn2 = nn.BatchNorm2d(64)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(256)

        self.fc1 = nn.Linear(14*14*256, hidden_1)
        self.fc2 = nn.Linear(hidden_1, 133)
        #self.fc3 = nn.Linear(500, 133)

        self.dropout = nn.Dropout(p=0.5)

        self.bn1_1 = nn.BatchNorm1d(hidden_1)

    def forward(self, x):

        ## Define forward behavior
        x = self.bn1(self.pool(F.relu(self.conv1(x))))
        x = self.bn2(self.pool(F.relu(self.conv2(x))))
        x = self.bn3(self.pool(F.relu(self.conv3(x))))
        x = self.bn4(self.pool(F.relu(self.conv4(x))))
        #x = self.pool(F.relu(self.conv5(x)))

        x = x.view(-1, 14*14*256)
```

```
                x = self.dropout(x)
                x = self.bn1_1(F.relu(self.fc1(x)))
                x = self.dropout(x)
                x = self.fc2(x)
                #x = F.relu(self.fc2(x))
                #x = self.dropout(x)
                #x = self.fc3(x)

                return x

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

        # instantiate the CNN
        model_scratch = Net()

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

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

```
Net(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv4): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (bn4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc1): Linear(in_features=50176, out_features=500, bias=True)
  (fc2): Linear(in_features=500, out_features=133, bias=True)
  (dropout): Dropout(p=0.5)
  (bn1_1): BatchNorm1d(500, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)
```

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

Answer:

Outline: I use a general approach when constructing my network: using 1 Conv layer, followed by a ReLU activation and then applying a MaxPooling layer. I repeated this for about 4 times, and then I flatten the output of the 4th MaxPooling layer and feeding it into 2 fully connected layers.

Regarding Conv Layers and Pooling Layers: For the convolutional layers, I increased the depth of the feature maps by a factor of 2 at each layer, from 32 to 256 so that my network is able to

learn about the complex features of the images as much as possible. I apply padding to keep the dimensions of the feature maps the same, but I then apply MaxPooling layers to reduce the irrelevant spatial information while preserving the higher order information, or the general shape and the unique characteristics of each image. I also added dropouts for my network as an attempt to reduce overfitting.

Regarding the arithmetic of the fc layers: Since we start with images having dimension of 224Œ224Œ3, after a series of Conv layers and Pooling layers, I ended up with 14Œ14Œ256. Feeding into the fc layers, the in_features wil be 14Œ14Œ256 = 50176. The choice of "500" is arbitary, and I could have tried other numbers too. Lastly, the network ouput 133 classes as indicated previously.

Regarding Batch Normalization I've been training this network for the wholeday today but I didn't get good results, until I encountered batch normalization which essentially speeds up training and makes the network more stable.(correct if I am wrong) I decided to apply 4 layers of batch norm after applying MaxPooling, and I also added a layer of batch norm to my first fully connected layer.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

In [14]:
```
import torch.optim as optim

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

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

1.1.10 (IMPLEMENTATION) Train and Validate the Model

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

In [15]:
```
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()

    # Clear out gradients
    optimizer.zero_grad()

    ## find the loss and update the model parameters accordingly
    output = model_scratch(data)
    loss = criterion_scratch(output, target)

    # backward pass
    loss.backward()

    # optimization step
    optimizer.step()

    ## record the average training loss, using something like
    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()

    # turn off gradients
    with torch.no_grad():
        output = model_scratch(data)
        loss = criterion_scratch(output, target)

    ## update the 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(), 'model_scartch.pt')
                    valid_loss_min = valid_loss

            # return trained model
            return model

        # define the term "loaders_scratch"
        loaders_scratch = {'train': trainloader,
                           'valid': validloader,
                           'test': testloader}

        # train the model
        model_scratch = train(25, 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: 4.748022         Validation Loss: 4.441649
Validation loss decreased (inf --> 4.441649).  Saving model ...
Epoch: 2         Training Loss: 4.521432         Validation Loss: 4.338630
Validation loss decreased (4.441649 --> 4.338630).  Saving model ...
Epoch: 3         Training Loss: 4.383112         Validation Loss: 4.201462
Validation loss decreased (4.338630 --> 4.201462).  Saving model ...
Epoch: 4         Training Loss: 4.267701         Validation Loss: 4.013708
Validation loss decreased (4.201462 --> 4.013708).  Saving model ...
Epoch: 5         Training Loss: 4.162623         Validation Loss: 4.164132
Epoch: 6         Training Loss: 4.072495         Validation Loss: 3.929352
Validation loss decreased (4.013708 --> 3.929352).  Saving model ...
Epoch: 7         Training Loss: 3.951066         Validation Loss: 3.703404
Validation loss decreased (3.929352 --> 3.703404).  Saving model ...
Epoch: 8         Training Loss: 3.875613         Validation Loss: 3.722813
Epoch: 9         Training Loss: 3.804441         Validation Loss: 3.797364
Epoch: 10         Training Loss: 3.738830         Validation Loss: 3.599678
Validation loss decreased (3.703404 --> 3.599678).  Saving model ...
Epoch: 11         Training Loss: 3.686341         Validation Loss: 3.475035
Validation loss decreased (3.599678 --> 3.475035).  Saving model ...
Epoch: 12         Training Loss: 3.649364         Validation Loss: 3.590449
```

```
Epoch: 13         Training Loss: 3.558902         Validation Loss: 3.527469
Epoch: 14         Training Loss: 3.507843         Validation Loss: 3.561864
Epoch: 15         Training Loss: 3.502861         Validation Loss: 3.301803
Validation loss decreased (3.475035 --> 3.301803).  Saving model ...
Epoch: 16         Training Loss: 3.423775         Validation Loss: 3.317381
Epoch: 17         Training Loss: 3.369113         Validation Loss: 3.350272
Epoch: 18         Training Loss: 3.361432         Validation Loss: 3.220204
Validation loss decreased (3.301803 --> 3.220204).  Saving model ...
Epoch: 19         Training Loss: 3.294245         Validation Loss: 3.103379
Validation loss decreased (3.220204 --> 3.103379).  Saving model ...
Epoch: 20         Training Loss: 3.290524         Validation Loss: 3.069099
Validation loss decreased (3.103379 --> 3.069099).  Saving model ...
Epoch: 21         Training Loss: 3.228276         Validation Loss: 2.946165
Validation loss decreased (3.069099 --> 2.946165).  Saving model ...
Epoch: 22         Training Loss: 3.218585         Validation Loss: 3.090983
Epoch: 23         Training Loss: 3.191311         Validation Loss: 2.924338
Validation loss decreased (2.946165 --> 2.924338).  Saving model ...
Epoch: 24         Training Loss: 3.145145         Validation Loss: 2.982123
Epoch: 25         Training Loss: 3.075402         Validation Loss: 2.970784
```

```
---------------------------------------------------------------------------

FileNotFoundError                         Traceback (most recent call last)

<ipython-input-15-eb10b909af59> in <module>()
 85
 86 # load the model that got the best validation accuracy
---> 87 model_scratch.load_state_dict(torch.load('model_scratch.pt'))


/opt/conda/lib/python3.6/site-packages/torch/serialization.py in load(f, map_location, p
299             (sys.version_info[0] == 3 and isinstance(f, pathlib.Path)):
300         new_fd = True
--> 301         f = open(f, 'rb')
302     try:
303         return _load(f, map_location, pickle_module)


FileNotFoundError: [Errno 2] No such file or directory: 'model_scratch.pt'
```

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 [16]: def test(loaders, model, criterion, use_cuda):

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

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

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

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

         # call test function
         test(loaders_scratch, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.942720


Test Accuracy: 25% (217/836)
```

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 [2]: import os
        import torch
        from torchvision import datasets, transforms

        import torch
        import torchvision.models as models


        # Define transforms for data augmentation and test_transform
        train_transforms = transforms.Compose([transforms.Resize(255),
                                               transforms.RandomRotation(30),
                                               transforms.RandomResizedCrop(224),
                                               transforms.RandomVerticalFlip(),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.ToTensor(),
                                               transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                    std=[0.229, 0.224, 0.225])
                                              ])

        test_transforms = transforms.Compose([transforms.Resize(255),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                                   std=[0.229, 0.224, 0.225])
                                             ])


        ## TODO: Specify data loaders

        train_data = datasets.ImageFolder(root='/data/dog_images/train', transform = train_trans
        valid_data = datasets.ImageFolder(root='/data/dog_images/valid', transform = test_transf
        test_data = datasets.ImageFolder(root='/data/dog_images/test', transform = test_transfor

        trainloader = torch.utils.data.DataLoader(train_data, batch_size = 128, shuffle = True)
        validloader = torch.utils.data.DataLoader(valid_data, batch_size = 128, shuffle = True)
        testloader = torch.utils.data.DataLoader(test_data, batch_size = 128, shuffle = True)
```

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 [3]: import torchvision.models as models
        import torch.nn as nn
```

```
        ## TODO: Specify model architecture

        # Using a pretrained ResNet-50 model
        model_transfer = models.resnet50(pretrained=True)

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

        # Replace the fc layer with a new fc layer
        n_inputs = model_transfer.fc.in_features
        new_last_layer = nn.Linear(n_inputs, 133)
        model_transfer.fc = new_last_layer

        print(model_transfer.fc)

        # Move to GPU if possible
        use_cuda = torch.cuda.is_available()

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

```
Downloading: "https://download.pytorch.org/models/resnet50-19c8e357.pth" to /root/.torch/models/
100%|| 102502400/102502400 [00:05<00:00, 18324129.62it/s]


Linear(in_features=2048, out_features=133, bias=True)
```

Question 5: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. Describe why you think the architecture is suitable for the current problem.

Answer:

Outline Doing the similar procedures previously, I downloaded the ResNet-50 Network and printed out the its structure to evaluate the components of the network. I discovered that there is only one classifier layer, named "fc" in the end. Since I have a smaller dataset than the ImageNet data but I have similar data, I decided to just sliced off the final fully connected layer and freezed the pretrained weights which are already optimal. I then added in a new linear fully connected layer which has 2048 input features and 133 output features, which are the different classes of the dogs in our image set. I eventually trained the network to update the weights of the new fully connected layer.

Why did I use ResNet-50? I have been pretty interested in using ResNet-50 when I first encountered transfer learning in this course, because we are able to use it to train a much much deeper neural network while not having to face all sorts of challenges, because of its special network structure(the shortcut connections). From my understanding so far, correct me if I am wrong, using a deeper layer allows the model to learn deeper and perform better, but that has a trade off like increasing complexity, vanishing gradients and longer duration of time for training. Also, the

ResNet-50 was trained on the ImageNet dataset which has dogs in it and was used to perform image classification. Given the similar dataset and task, I decided that ResNet-50 was a good idea to implement!

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 [4]: import torch.optim as optim

        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.Adam(model_transfer.fc.parameters(), lr=0.001)
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

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

        # train the model
        def train(n_epochs, loaders, model, optimizer, criterion, use_cuda, save_path):

            # initiate validation loss to infinity to track valid loss
            valid_loss_min = np.Inf

                ##################
                # train the model #
                ##################

            for epoch in range(1, n_epochs+1):
                train_loss = 0.0
                valid_loss = 0.0

                model.train()
                for batch_idx, (data, target) in enumerate(loaders['train']):
                    if use_cuda:
                        data, target = data.cuda(), target.cuda()

                        optimizer.zero_grad()
                        output = model_transfer(data)
                        loss = criterion_transfer(output, target)
                        loss.backward()
                        optimizer.step()
                        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()

            # turn off gradients
            with torch.no_grad():
                output = model_transfer(data)
                loss = criterion_transfer(output, target)

            ## update the 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
            ))

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

    return model

# define the term "loaders_scratch"
loaders_transfer = {'train': trainloader,
                    'valid': validloader,
                    'test': testloader}

# train the model
model_transfer = train(25, loaders_transfer, model_transfer, optimizer_transfer,
                       criterion_transfer, use_cuda, 'model_transfer.pt')

# 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: 3.947006         Validation Loss: 2.131534
Validation loss decreased (inf --> 2.131534).  Saving model ...
Epoch: 2         Training Loss: 2.499634         Validation Loss: 1.218292
Validation loss decreased (2.131534 --> 1.218292).  Saving model ...
Epoch: 3         Training Loss: 2.034947         Validation Loss: 0.913602
Validation loss decreased (1.218292 --> 0.913602).  Saving model ...
Epoch: 4         Training Loss: 1.807682         Validation Loss: 0.766776
Validation loss decreased (0.913602 --> 0.766776).  Saving model ...
Epoch: 5         Training Loss: 1.673991         Validation Loss: 0.762175
Validation loss decreased (0.766776 --> 0.762175).  Saving model ...
Epoch: 6         Training Loss: 1.609483         Validation Loss: 0.673691
Validation loss decreased (0.762175 --> 0.673691).  Saving model ...
Epoch: 7         Training Loss: 1.522450         Validation Loss: 0.644187
Validation loss decreased (0.673691 --> 0.644187).  Saving model ...
Epoch: 8         Training Loss: 1.461931         Validation Loss: 0.637399
Validation loss decreased (0.644187 --> 0.637399).  Saving model ...
Epoch: 9         Training Loss: 1.426138         Validation Loss: 0.603084
Validation loss decreased (0.637399 --> 0.603084).  Saving model ...
Epoch: 10         Training Loss: 1.376650         Validation Loss: 0.593572
Validation loss decreased (0.603084 --> 0.593572).  Saving model ...
Epoch: 11         Training Loss: 1.359857         Validation Loss: 0.547275
Validation loss decreased (0.593572 --> 0.547275).  Saving model ...
Epoch: 12         Training Loss: 1.348329         Validation Loss: 0.576672
Epoch: 13         Training Loss: 1.296765         Validation Loss: 0.528360
Validation loss decreased (0.547275 --> 0.528360).  Saving model ...
Epoch: 14         Training Loss: 1.253486         Validation Loss: 0.510812
Validation loss decreased (0.528360 --> 0.510812).  Saving model ...
Epoch: 15         Training Loss: 1.273037         Validation Loss: 0.529212
Epoch: 16         Training Loss: 1.250449         Validation Loss: 0.543765
Epoch: 17         Training Loss: 1.251667         Validation Loss: 0.497410
Validation loss decreased (0.510812 --> 0.497410).  Saving model ...
Epoch: 18         Training Loss: 1.234266         Validation Loss: 0.513844
Epoch: 19         Training Loss: 1.211701         Validation Loss: 0.513068
Epoch: 20         Training Loss: 1.197471         Validation Loss: 0.512651
Epoch: 21         Training Loss: 1.183837         Validation Loss: 0.489323
Validation loss decreased (0.497410 --> 0.489323).  Saving model ...
Epoch: 22         Training Loss: 1.172897         Validation Loss: 0.533788
Epoch: 23         Training Loss: 1.158804         Validation Loss: 0.522457
Epoch: 24         Training Loss: 1.168018         Validation Loss: 0.489861
Epoch: 25         Training Loss: 1.129422         Validation Loss: 0.513538
```

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 [6]: def test(loaders, model, criterion, use_cuda):

            test_loss = 0.
            correct = 0.
            total = 0.

            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['test']):

                # Use GPU if possible
                if use_cuda:
                    data, target = data.cuda(), target.cuda()

                # forward pass
                output = model_transfer(data)
                loss = criterion_transfer(output, target)
                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))


        test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.458344


Test Accuracy: 85% (715/836)
```

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

         # list of class names by index, i.e. a name can be accessed like class_names[0]
```

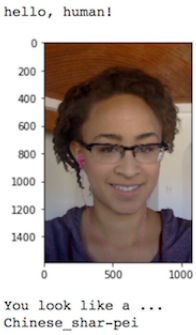
```
class_names = [item[4:].replace("_", " ") for item in train_data.classes]

# loading in my ResNet-50 Transfer Learning Model
model_transfer.load_state_dict(torch.load('model_transfer.pt'))

def predict_breed_transfer(img_path):
    # load the image and return the predicted breed

    # Loading an image from the given img_path
    image = Image.open(img_path)

    # Define the pre-processing steps
    tensor_transform = transforms.Compose([transforms.Resize(255),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize(mean=[0.485, 0.456, 0.4
                                                                      std=[0.229, 0.224, 0.2
                                          ])

    # Transform the image into an image tensor and add a dimension
    image_tensor = tensor_transform(image)
    image_tensor = image_tensor.unsqueeze(0)

    # Moving to GPU if possible
    if use_cuda:
        image_tensor = image_tensor.cuda()

    # Turned off gradients and backprob and predict with my ResNet-50 Transfer Learning
    with torch.no_grad():
        output = model_transfer(image_tensor)
        predicted_index = torch.argmax(output)

    breed = class_names[predicted_index]

    return breed
```

Step 5: Write your Algorithm

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

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

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



Sample Human Output

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [21]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.
         %matplotlib inline

         def run_app(img_path):
             ## handle cases for a human face, dog, and neither

             if face_detector(img_path) is True:
                 print("Hello, human!")
                 plt.imshow(Image.open(img_path))
                 plt.show()
                 print("You look like a...", predict_breed_transfer(img_path))
                 print('\n----------------------------------\n')
             elif dog_detector(img_path) is True:
                 print("Hello, dog!")
                 plt.imshow(Image.open(img_path))
                 plt.show()
                 print('Your predicted breed is...', predict_breed_transfer(img_path))
                 print('\n----------------------------------\n')
             else:
                 print("Sorry, you don't seem to be a human or a dog")
                 print('\n----------------------------------\n')
```

Step 6: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that *you* look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

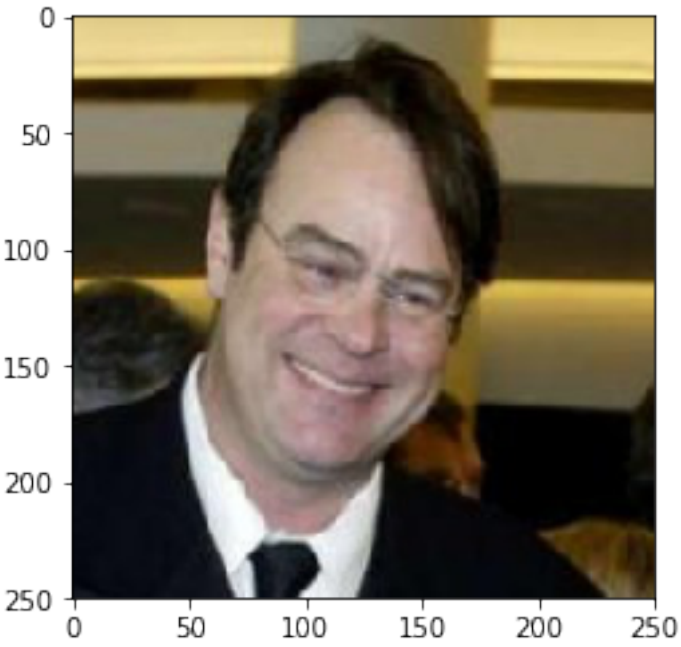
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

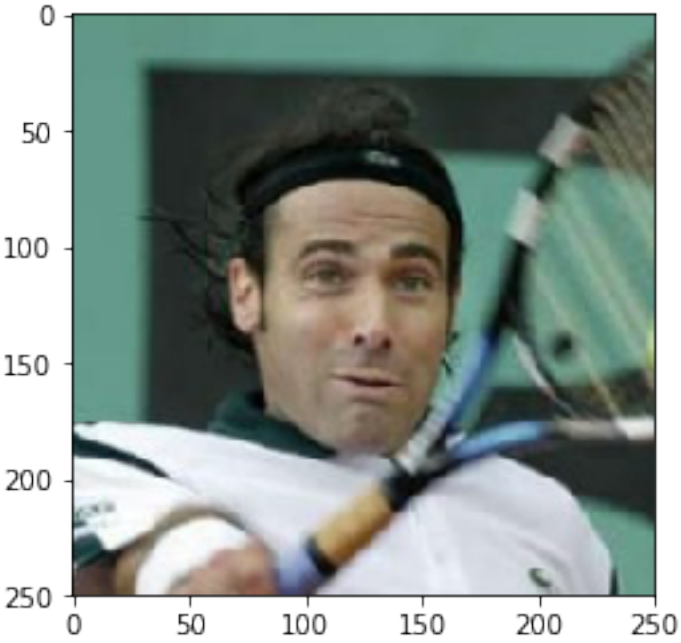
Answer: (Three possible points for improvement)

```
In [24]: ## 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[:11], dog_files[:30])):
             run_app(file)
```

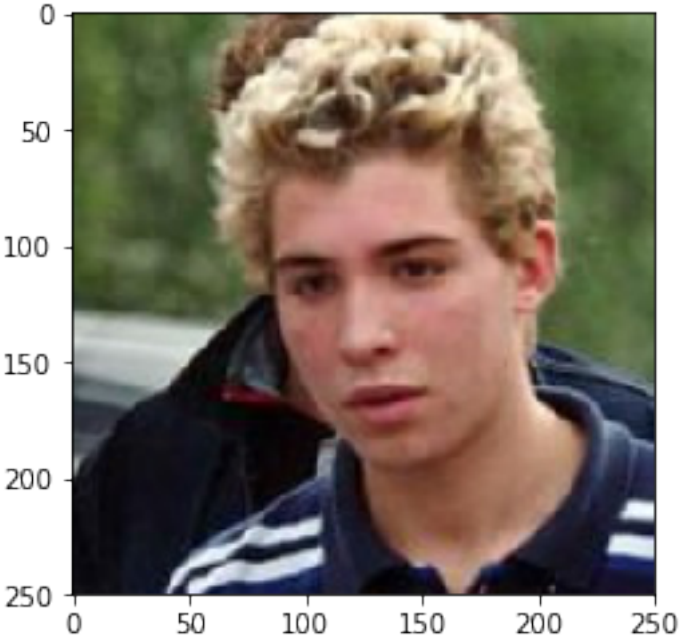
```
Hello, human!
```



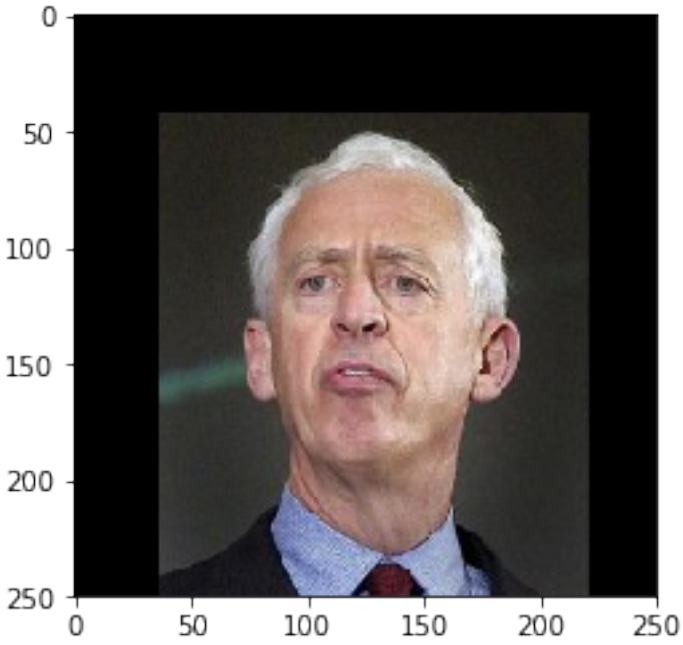

```
You look like a... French bulldog

-----------------------------------

Hello, human!
```



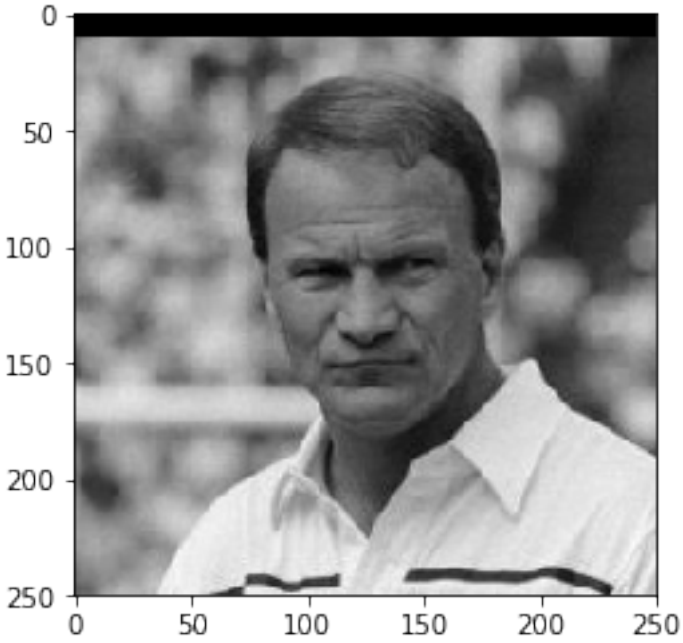

```
You look like a... American foxhound

-----------------------------------

Hello, human!
```



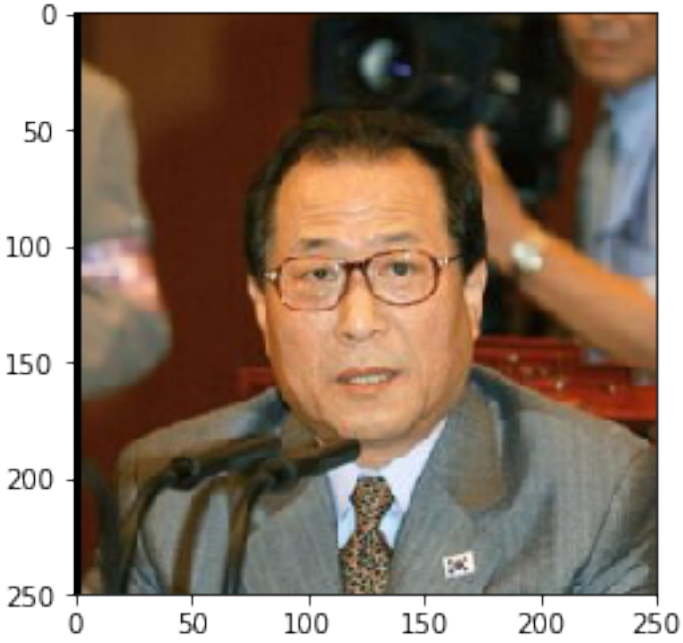
```
You look like a... American water spaniel

-----------------------------------

Hello, human!
```



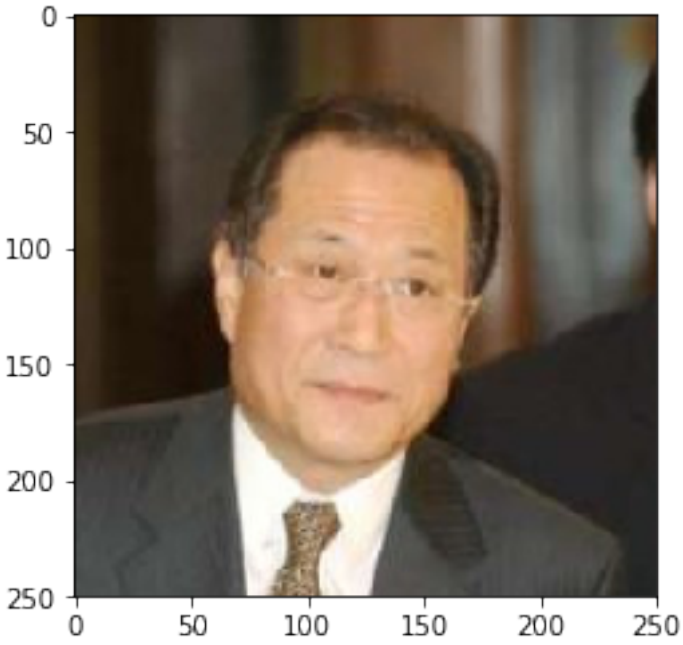

```
You look like a... Norfolk terrier

-----------------------------------

Hello, human!
```



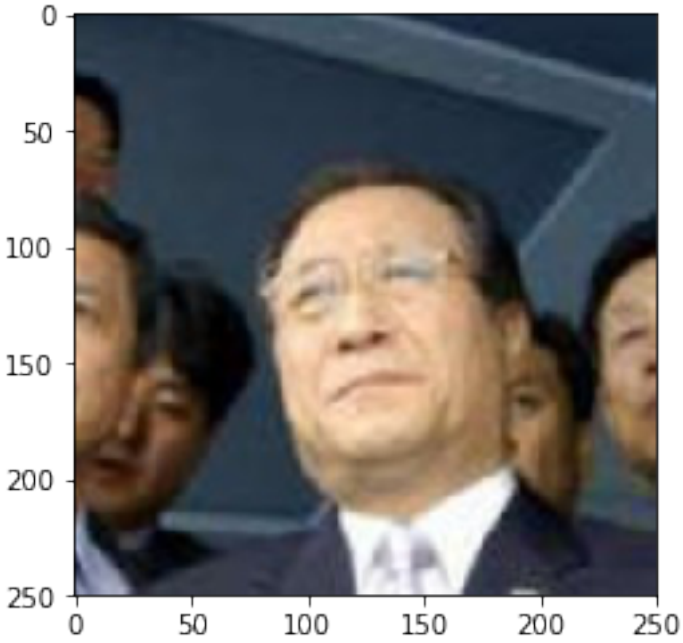

```
You look like a... German pinscher

-----------------------------------

Hello, human!
```



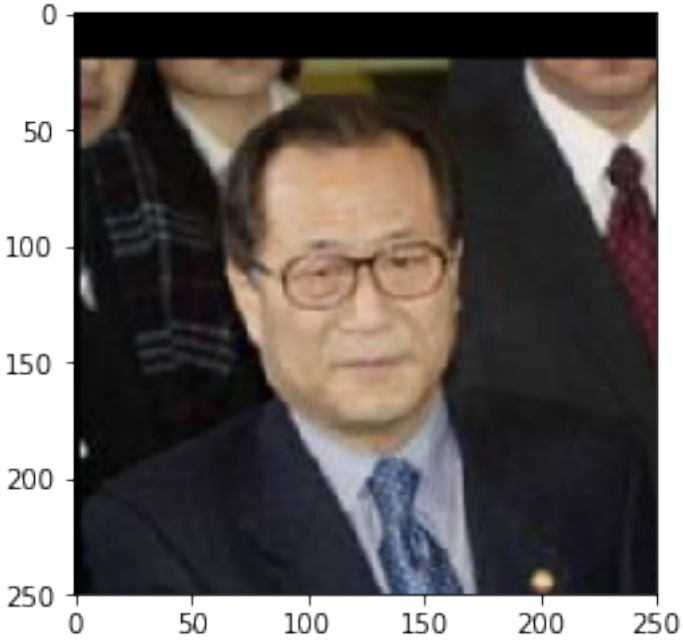
```
You look like a... Poodle

-----------------------------------

Hello, human!
```



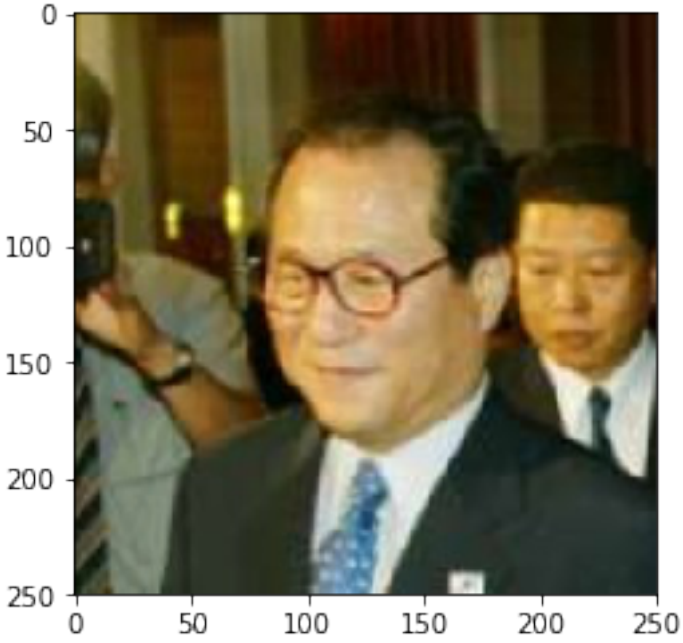

```
You look like a... Norfolk terrier

-----------------------------------

Hello, human!
```



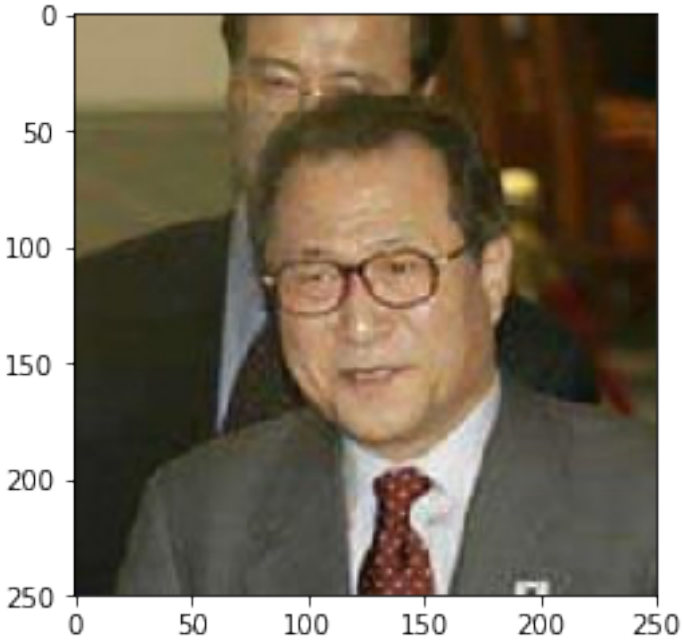

```
You look like a... Bloodhound

-----------------------------------

Hello, human!
```



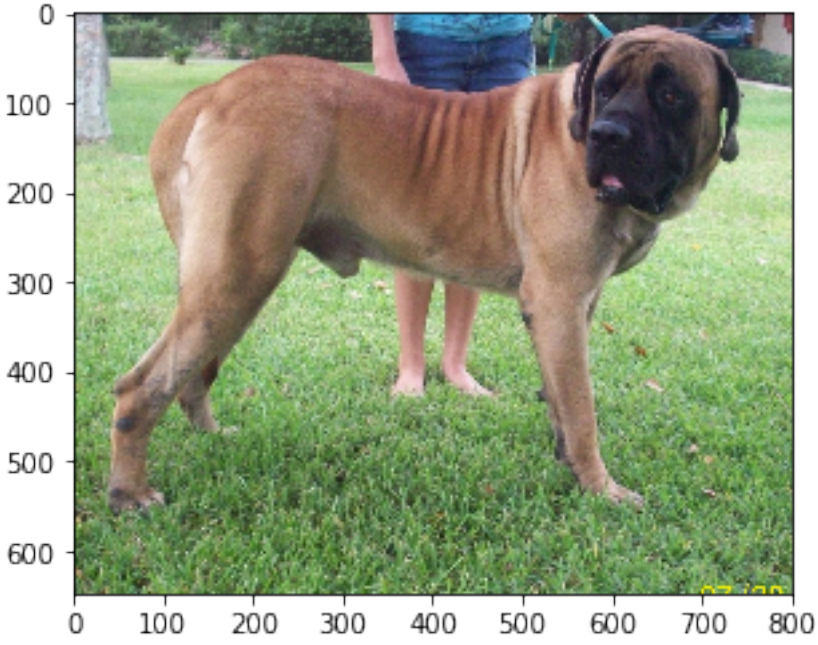
```
You look like a... Irish wolfhound

-----------------------------------

Hello, human!
```



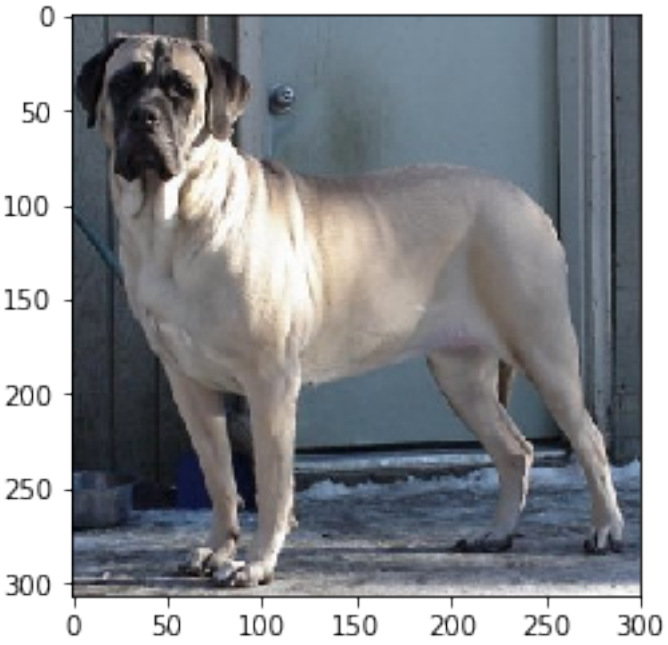

```
You look like a... Basenji

-----------------------------------

Hello, human!
```



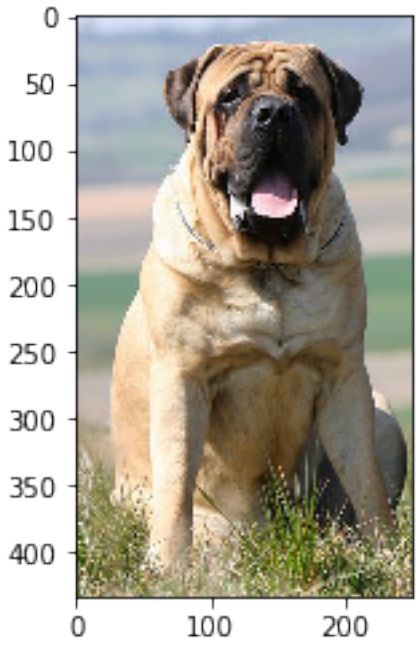

```
You look like a... Norfolk terrier

-----------------------------------

Hello, dog!
```



```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```

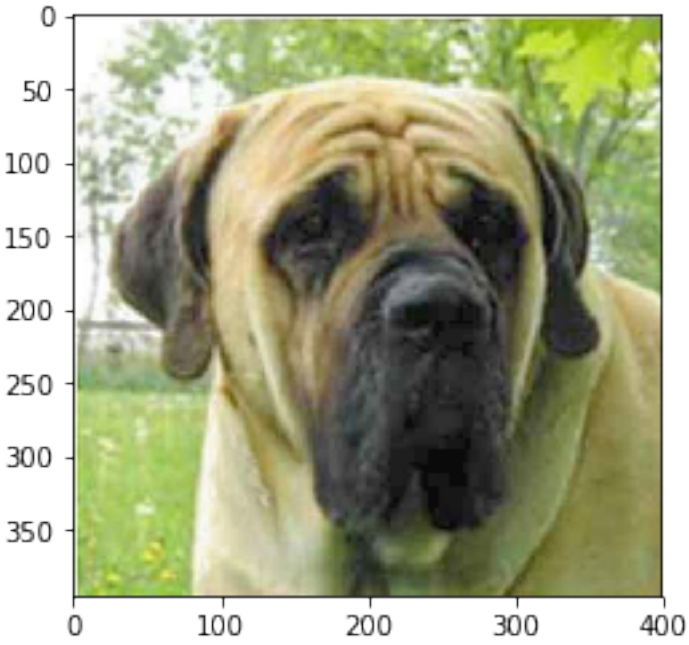



```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



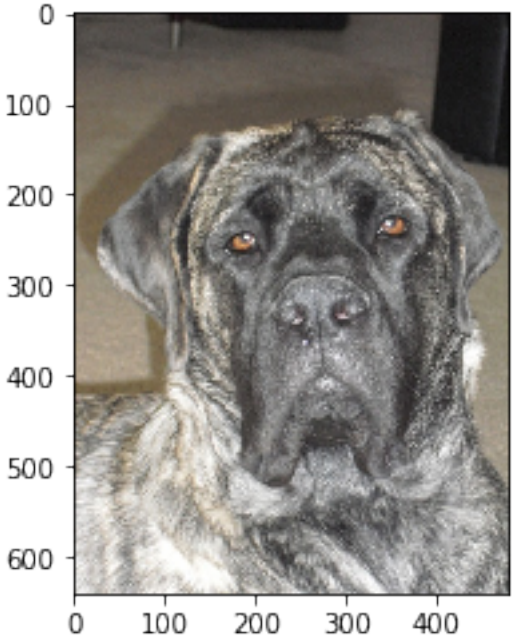

```
Your predicted breed is... Bullmastiff

-----------------------------------

Hello, dog!
```

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



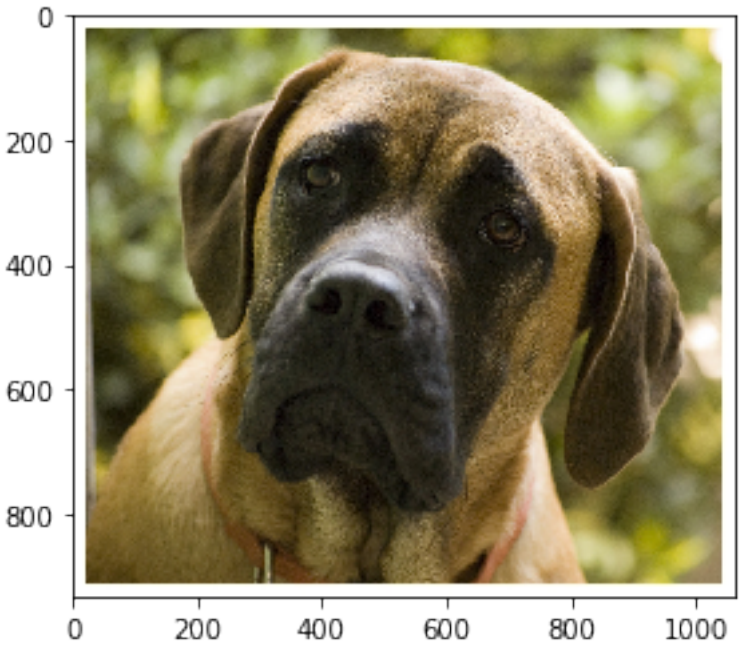

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```


```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```

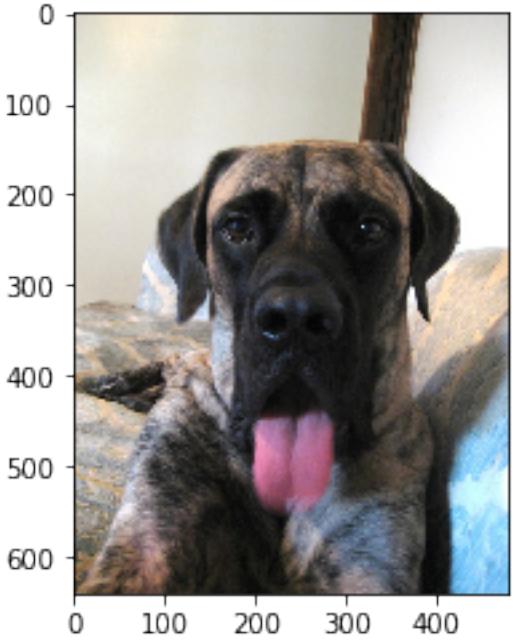


```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```

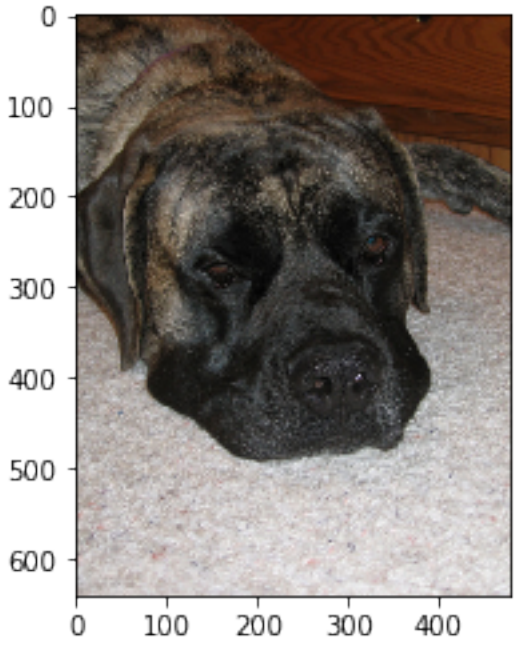

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



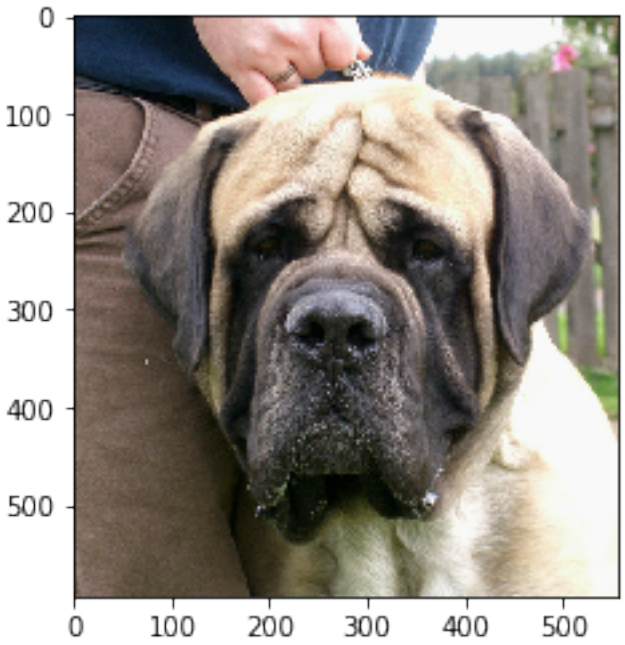

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



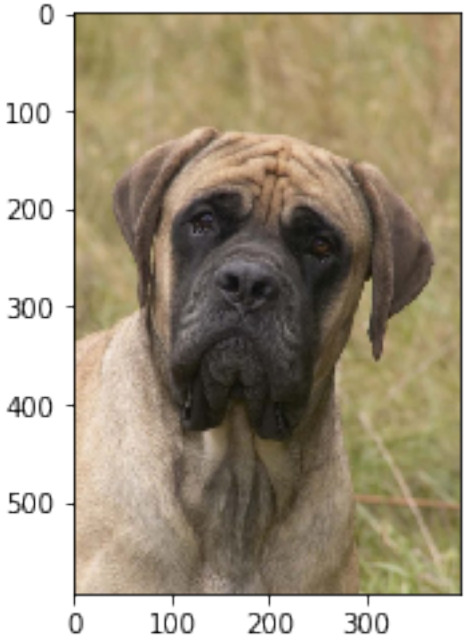
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



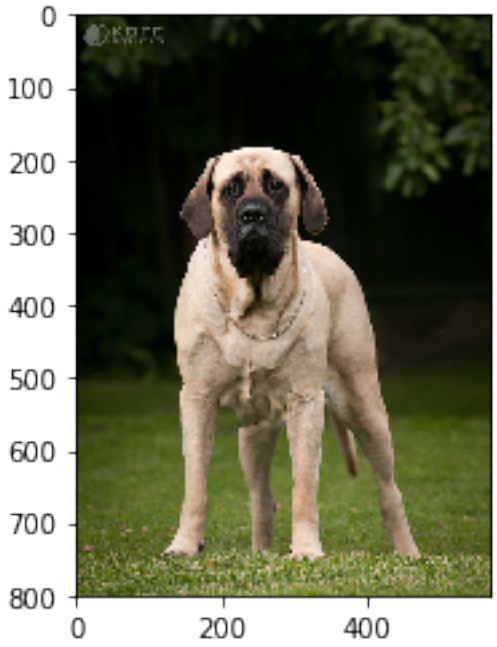

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



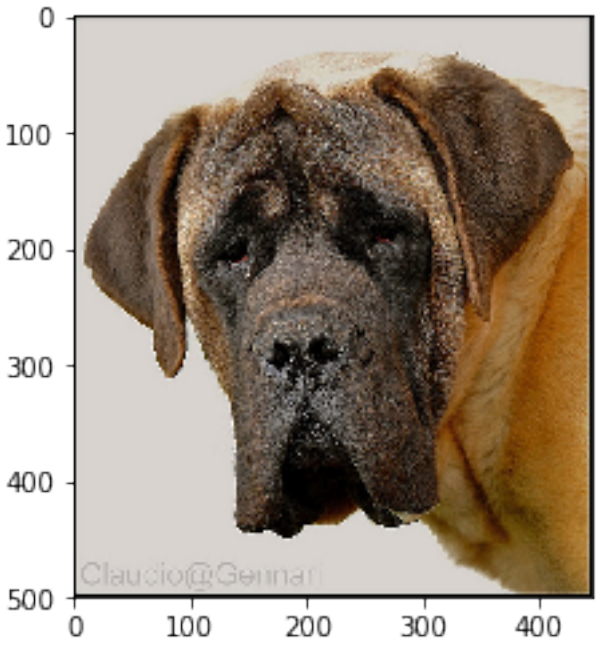

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



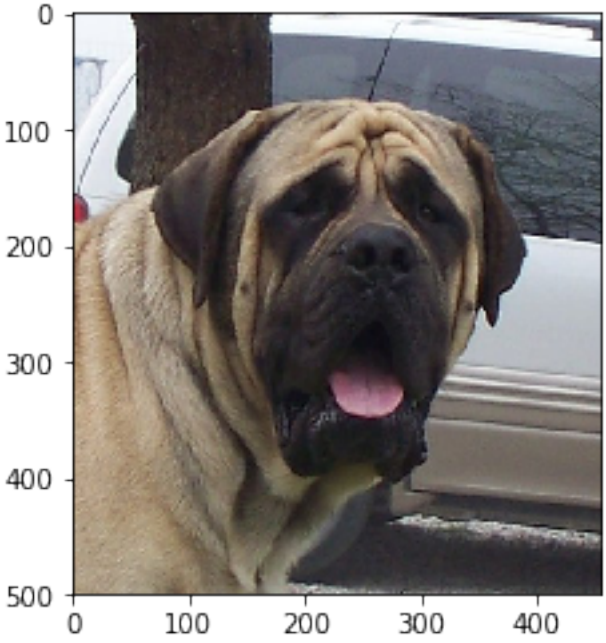
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



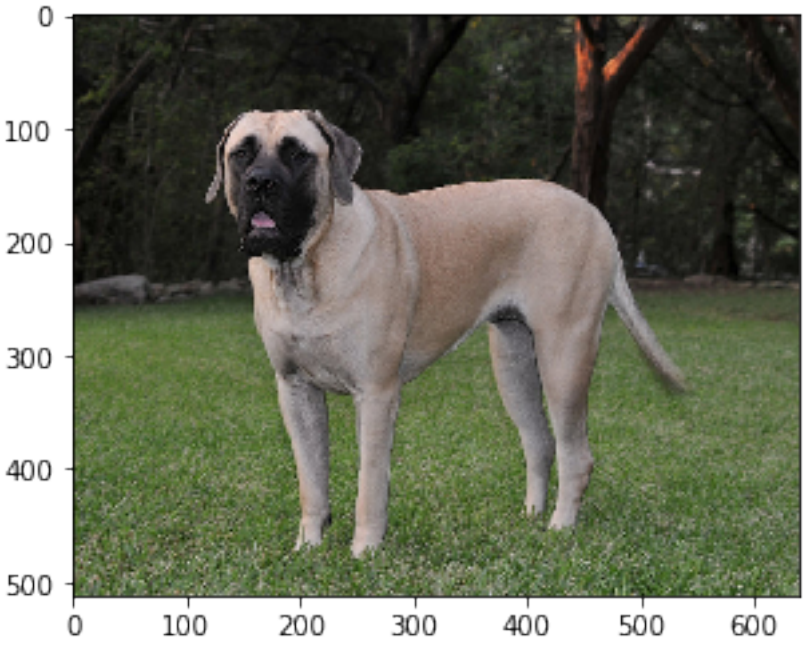

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



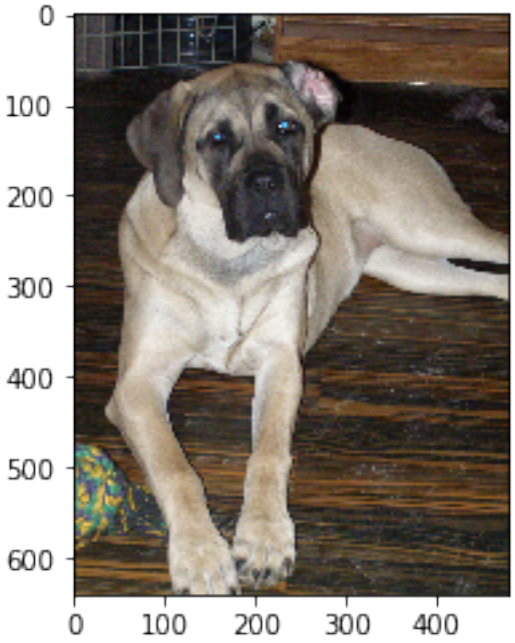

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



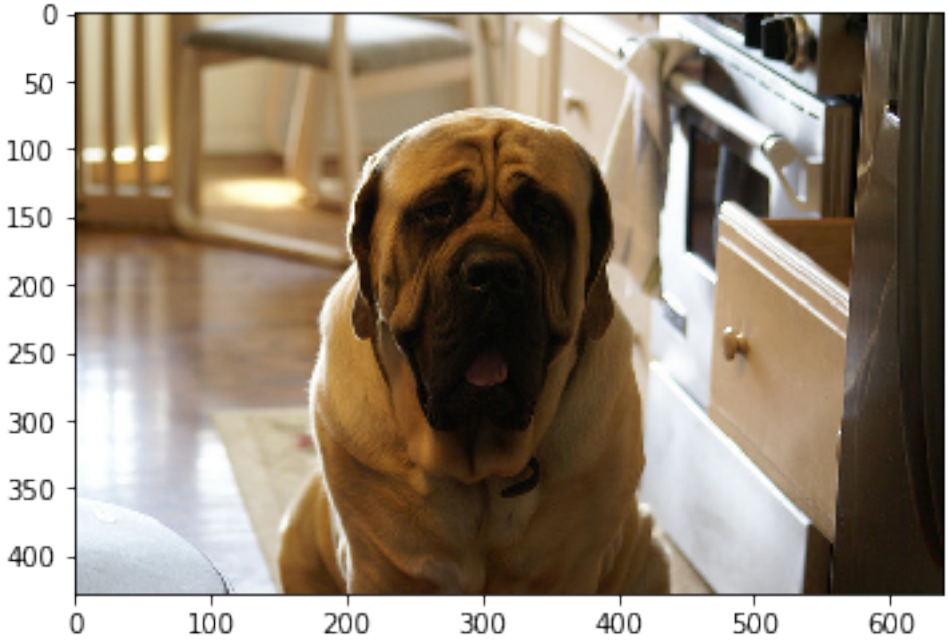
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



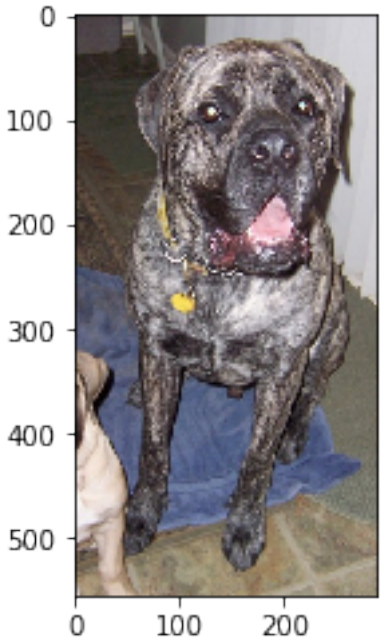

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



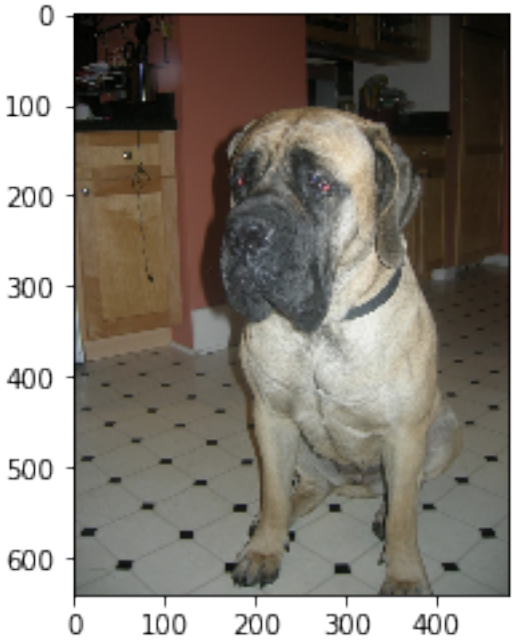

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



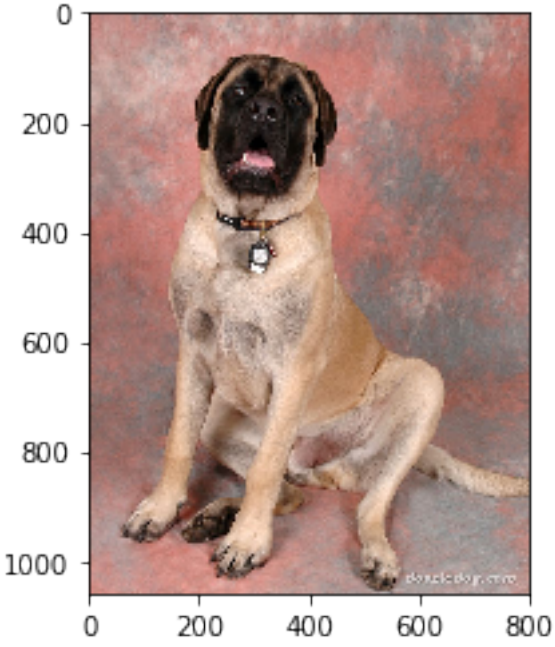
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



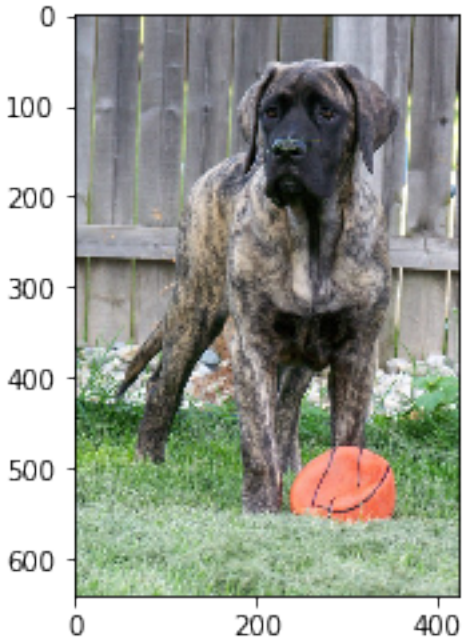

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



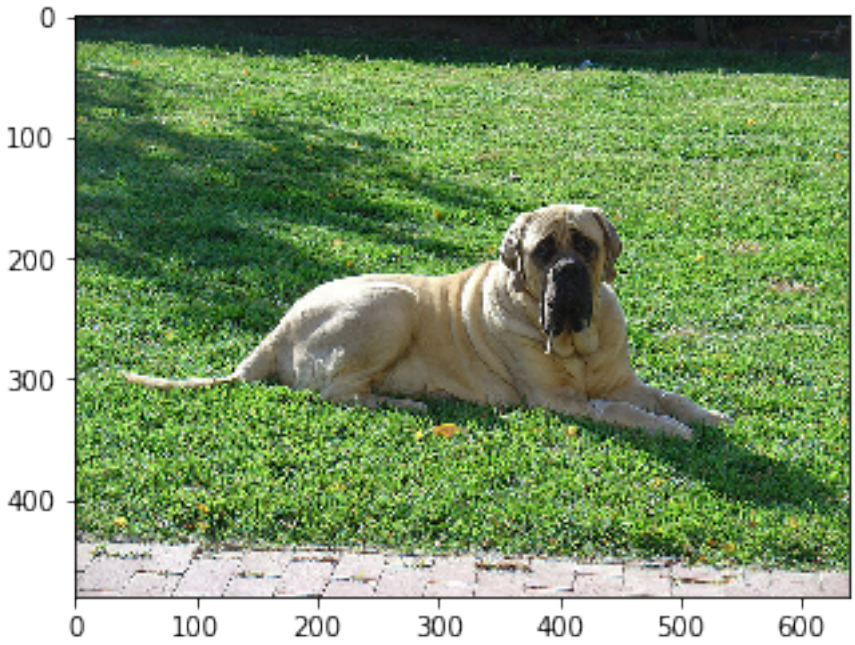

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



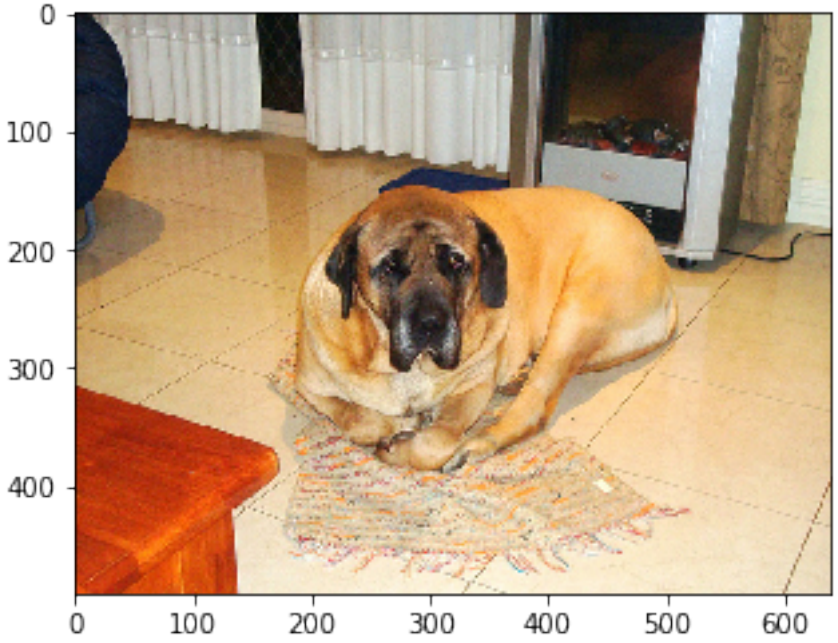
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



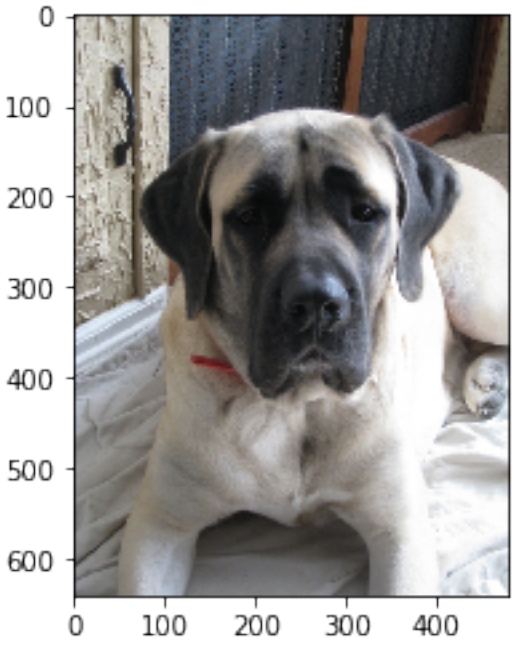

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



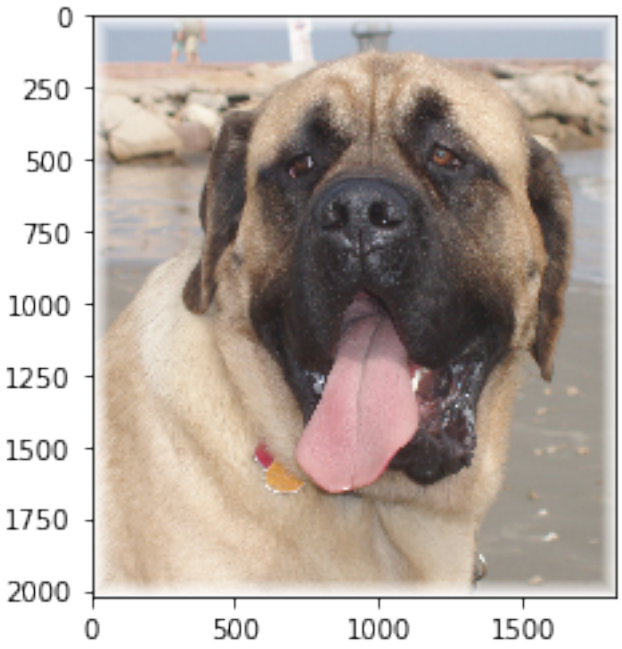

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



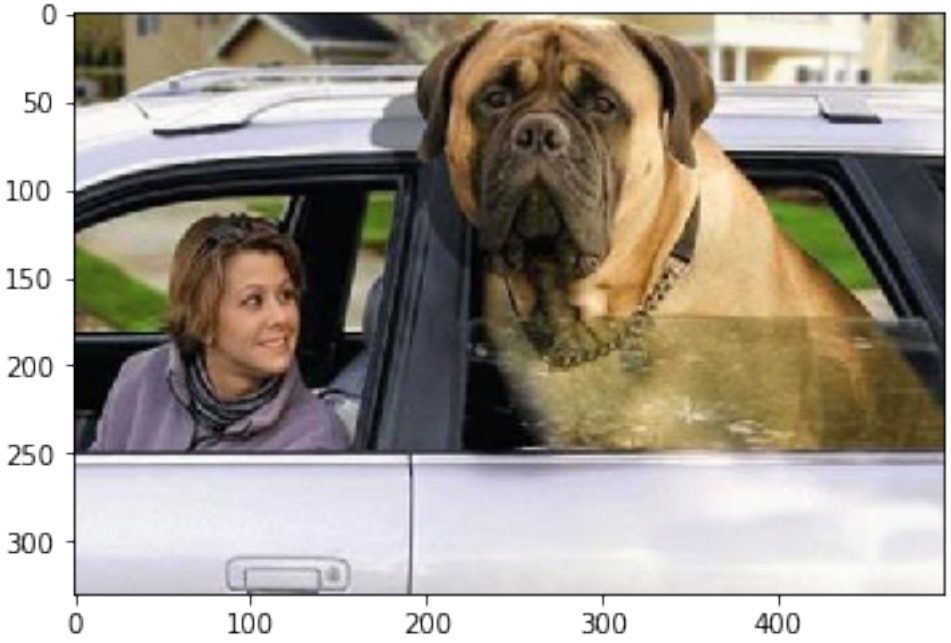
```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```



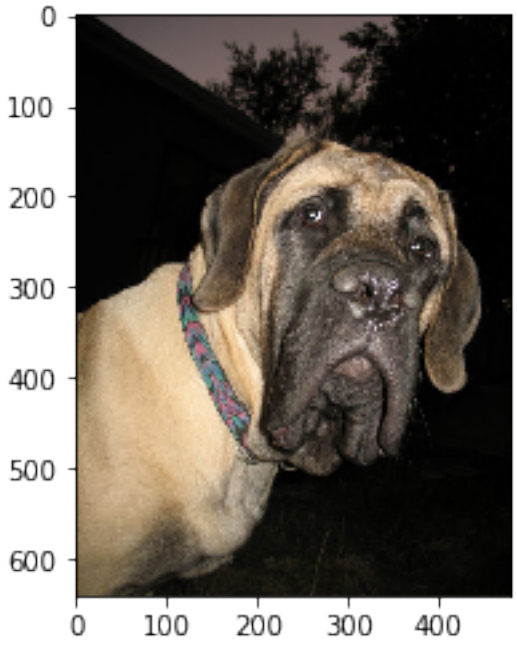

```
Your predicted breed is... Mastiff

-----------------------------------

Hello, human!
```




```
You look like a... Mastiff

-----------------------------------

Hello, human!
```



```
You look like a... Mastiff

-----------------------------------

Hello, dog!
```




```
Your predicted breed is... Mastiff

-----------------------------------

Hello, dog!
```


```
Your predicted breed is... Mastiff

-----------------------------------
```

```
In [ ]:
```