dog_app

May 14, 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

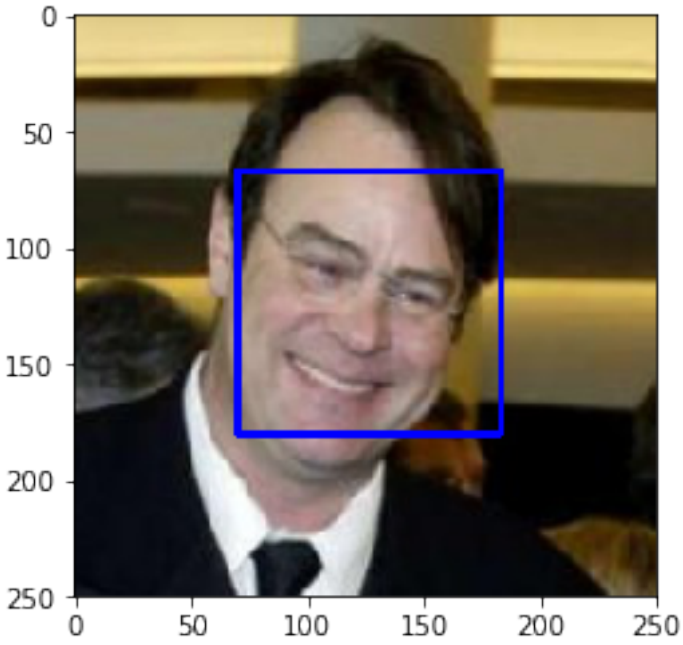
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [3]: 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 [4]: # 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 [5]: 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.
        count1=0
        count2=0
        for image in human_files_short:
            if face_detector(image):
                count1+=1
        for image in dog_files_short:
            if face_detector(image):
                count2+=1
        print(f"{count1}% human in human_files")
        print(f"{count2}% human in dog_files")
```

```
98% human in human_files
17% human in dog_files
```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make

use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```
In [5]: ### (Optional)
        ### TODO: Test performance of anotherface detection algorithm.
        ### Feel free to use as many code cells as needed.
```

Step 2: Detect Dogs

In this section, we use a pre-trained model to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

```
Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg
100%|| 553433881/553433881 [00:08<00:00, 68362548.10it/s]
```

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

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

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

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

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

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

            Args:
                img_path: path to an image

            Returns:
                Index corresponding to VGG-16 model's prediction
            '''
            tran=transforms.Compose([transforms.CenterCrop(244),
                                     transforms.ToTensor(),
                                     transforms.Normalize([0.485, 0.456, 0.406],
                                                          [0.229, 0.224, 0.225])])

            ## 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
            image=Image.open(img_path)
            image=tran(image)
            image=image.unsqueeze(0)
            image=image.cuda()
            output=VGG16(image).data.argmax()
            return output # predicted dex
```

1.1.5 (IMPLEMENTATION) Write a Dog Detector

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

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

```
In [8]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.
            index=VGG16_predict(img_path)
            return 150<index<269
```

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_files_short = human_files[:100]
        dog_files_short = dog_files[:100]
        count1=0
        count2=0
        for image in human_files_short:
            if dog_detector(image):
                count1+=1
        for image in dog_files_short:
            if dog_detector(image):
                count2+=1
        print(f"{count1}% dogs in human_files")
        print(f"{count2}% dogs in dog_files")

0% dogs in human_files
91% dogs in dog_files
```

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 [10]: import os
         from torchvision import datasets
         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

         batch_size=30
         num_worker=0

         transform={
             "train":transforms.Compose([transforms.RandomRotation(45),
                                         transforms.RandomHorizontalFlip(),
                                         transforms.Resize(244),
                                         transforms.CenterCrop(244),
                                         transforms.ToTensor(),
                                         transforms.Normalize((0.485, 0.456, 0.406),
                                                              (0.229, 0.224, 0.225))]),

             "valid":transforms.Compose([transforms.Resize(244),
                                         transforms.CenterCrop(244),
                                         transforms.ToTensor(),
```

```
                                      transforms.Normalize((0.485, 0.456, 0.406),
                                                           (0.229, 0.224, 0.225))]),

        "test":transforms.Compose([transforms.Resize(244),
                                   transforms.CenterCrop(244),
                                   transforms.ToTensor(),
                                    transforms.Normalize((0.485, 0.456, 0.406),
                                                         (0.229, 0.224, 0.225))])
    }

    train_data=datasets.ImageFolder("/data/dog_images/train",transform=transform["train"])
    valid_data=datasets.ImageFolder("/data/dog_images/valid",transform=transform["valid"])
    test_data=datasets.ImageFolder("/data/dog_images/test",transform=transform["test"])

    data={
        "train":torch.utils.data.DataLoader(train_data,batch_size=batch_size,num_workers=nu
        "valid":torch.utils.data.DataLoader(valid_data,batch_size=batch_size,num_workers=nu
        "test":torch.utils.data.DataLoader(test_data,batch_size=batch_size,num_workers=num_
    }
```

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:

```
<li>I have resized images Resize(244) and CenterCrop(244). Size of input tensor is (3,224,244),
<li>For image augmentation, i have used RandomRotation(45), RandomHorizontalFlip() and Normalize
```

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [54]: 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, 32, 3, padding=1)

                 self.conv2=nn.Conv2d(32, 64, 3, padding=1)
                 self.bn_2=nn.BatchNorm2d(64)

                 self.conv3=nn.Conv2d(64, 128, 3, padding=1)
                 self.bn_3=nn.BatchNorm2d(128)
```

```
            self.conv4=nn.Conv2d(128, 128, 3, padding=1)
            self.bn_4=nn.BatchNorm2d(128)

            self.pool=nn.MaxPool2d(2,2)
            self.fc1=nn.Linear(15*15*128,512)
            self.fc2=nn.Linear(512,256)
            self.fc3=nn.Linear(256,133)
            self.drop=nn.Dropout(0.5)

        def forward(self, x):
            ## Define forward behavior
            x=self.pool(F.relu(self.conv1(x)))
            x=self.pool(F.relu(self.bn_2(self.conv2(x))))
            x=self.pool(F.relu(self.bn_3(self.conv3(x))))
            x=self.pool(F.relu(self.bn_4(self.conv4(x))))
            x=x.view(-1,15*15*128)
            x=self.drop(x)
            x=F.relu(self.fc1(x))
            x=self.drop(x)
            x=F.relu(self.fc2(x))
            x=self.drop(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()
```

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

Answer:

First, I like to mention that Designing Architecture for this kind of complicated task majorly depend in data argumentation.

Early version

```
<h4>CNN Architecture</h4>
 <li>3 -> 8 -> 16 -> 32 -> 64 -> 128</li>
 <li>3 -> 8 -> 16 -> 32 -> 64 -> 128 -> 256</li>
 <li>3 -> 16 -> 32 -> 64 -> 128</li>
 <li>3 -> 32 -> 64 -> 128</li>
 <li>3 -> 32 -> 64 -> 128 ->256</li>
 <li>3 -> 32 -> 64 -> 128 ->256 ->512</li>
```

```
    <li>3 -> 32 -> 64 -> 128 -> 256 ->256</li>
   <h4>Activation function</h4>
    <li>Relu activation function is used in all layer except output layer</li>
   <h4>Regularization</h4>
    <li>DropOut(0.2), DropOut(0.3), DropOut(0.4), DropOut(0.5)</li>
    <li>MaxPool2d(2,2)</li>
    <li>BatchNorm2d(depth)</li>
   <h4>Fully Connected Layer</h4>
    <li>input_dim -> 500 -> 133</li>
    <li>input_dim -> 256 -> 133</li>
    <li>input_dim -> 500 -> 300 -> 133</li>
    <li>input_dim -> 512 -> 133</li>
    <li>input_dim -> 500 -> 256 -> 133</li>
    <h5>Reason:</h5><p>I have try all these combination of Architecture, most of them didn't imp
```

Final version

```
<h4>CNN Architecture</h4>
    <li>3 -> 32 -> 64 -> 128 -> 128</li>
<h4>Activation function</h4>
    <li>Relu activation function is used in all layer except output layer</li>
<h4>Regularization</h4>
    <li>DropOut(0.5)</li>
    <li>MaxPool2d(2,2)</li>
    <li>BatchNorm2d(depth)</li>
<h4>Fully Connected Layer</h4>
<li>input_dim -> 500 -> 256 -> 133</li>
<h5>Reason:</h5><p>After so many hit and trial on pervious Architecture I got maximum accuracy c
```

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 [55]: 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 [56]: import time
         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
                 start=time.time()

                 ###################
                 # train the model #
                 ###################
                 model.train()
                 for batch_idx, (data, target) in enumerate(loaders['train']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## find the loss and update the model parameters accordingly
                     ## record the average training loss, using something like
                     ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                     optimizer.zero_grad()
                     out=model(data)
                     loss=criterion(out,target)
                     loss.backward()
                     optimizer.step()
                     train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.item() - train_los
                 ######################
                 # validate the model #
                 ######################
                 model.eval()
                 for batch_idx, (data, target) in enumerate(loaders['valid']):
                     # move to GPU
                     if use_cuda:
                         data, target = data.cuda(), target.cuda()
                     ## update the average validation loss
                     out=model(data)
                     loss=criterion(out,target)
                     valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.item() - valid_los
                 # print training/validation statistics
                 print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tTime Took:
                     epoch,
                     train_loss,
                     valid_loss,
                     time.time()-start
                     ))
```

```
            ## TODO: save the model if validation loss has decreased
            if valid_loss<valid_loss_min:
                torch.save(model.state_dict(), save_path)
                valid_loss_min=valid_loss
                print("Saved!")
        # return trained model
        return model


# train the model
model_scratch = train(100, data, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.941414         Validation Loss: 4.852635         Time Took: 14
Saved!
Epoch: 2         Training Loss: 4.833686         Validation Loss: 4.785933         Time Took: 14
Saved!
Epoch: 3         Training Loss: 4.785044         Validation Loss: 4.787337         Time Took: 14
Epoch: 4         Training Loss: 4.740670         Validation Loss: 4.695531         Time Took: 14
Saved!
Epoch: 5         Training Loss: 4.690743         Validation Loss: 4.648328         Time Took: 14
Saved!
Epoch: 6         Training Loss: 4.676137         Validation Loss: 4.646778         Time Took: 14
Saved!
Epoch: 7         Training Loss: 4.647908         Validation Loss: 4.609945         Time Took: 14
Saved!
Epoch: 8         Training Loss: 4.622295         Validation Loss: 4.597981         Time Took: 14
Saved!
Epoch: 9         Training Loss: 4.607126         Validation Loss: 4.564602         Time Took: 14
Saved!
Epoch: 10         Training Loss: 4.568204         Validation Loss: 4.529598         Time Took: 1
Saved!
Epoch: 11         Training Loss: 4.549955         Validation Loss: 4.593581         Time Took: 1
Epoch: 12         Training Loss: 4.526855         Validation Loss: 4.542934         Time Took: 1
Epoch: 13         Training Loss: 4.504242         Validation Loss: 4.447408         Time Took: 1
Saved!
Epoch: 14         Training Loss: 4.465261         Validation Loss: 4.433761         Time Took: 1
Saved!
Epoch: 15         Training Loss: 4.451074         Validation Loss: 4.421773         Time Took: 1
Saved!
Epoch: 16         Training Loss: 4.411558         Validation Loss: 4.366109         Time Took: 1
Saved!
Epoch: 17         Training Loss: 4.380267         Validation Loss: 4.397091         Time Took: 1
Epoch: 18         Training Loss: 4.347218         Validation Loss: 4.543199         Time Took: 1
Epoch: 19         Training Loss: 4.281910         Validation Loss: 4.241552         Time Took: 1
Saved!
Epoch: 20         Training Loss: 4.259721         Validation Loss: 4.197991         Time Took: 1
```

```
Saved!
Epoch: 21         Training Loss: 4.200379         Validation Loss: 4.205253         Time Took: 1
Epoch: 22         Training Loss: 4.172917         Validation Loss: 4.158079         Time Took: 1
Saved!
Epoch: 23         Training Loss: 4.120597         Validation Loss: 4.044804         Time Took: 1
Saved!
Epoch: 24         Training Loss: 4.080536         Validation Loss: 4.013608         Time Took: 1
Saved!
Epoch: 25         Training Loss: 4.049131         Validation Loss: 4.005262         Time Took: 1
Saved!
Epoch: 26         Training Loss: 4.016993         Validation Loss: 3.932519         Time Took: 1
Saved!
Epoch: 27         Training Loss: 3.989282         Validation Loss: 3.882300         Time Took: 1
Saved!
Epoch: 28         Training Loss: 3.940849         Validation Loss: 3.801624         Time Took: 1
Saved!
Epoch: 29         Training Loss: 3.906963         Validation Loss: 3.880912         Time Took: 1
Epoch: 30         Training Loss: 3.896759         Validation Loss: 3.814850         Time Took: 1
Epoch: 31         Training Loss: 3.850633         Validation Loss: 3.762826         Time Took: 1
Saved!
Epoch: 32         Training Loss: 3.840272         Validation Loss: 3.709170         Time Took: 1
Saved!
Epoch: 33         Training Loss: 3.776281         Validation Loss: 3.724579         Time Took: 1
Epoch: 34         Training Loss: 3.775828         Validation Loss: 3.678523         Time Took: 1
Saved!
Epoch: 35         Training Loss: 3.728153         Validation Loss: 3.662915         Time Took: 1
Saved!
Epoch: 36         Training Loss: 3.700731         Validation Loss: 3.627305         Time Took: 1
Saved!
Epoch: 37         Training Loss: 3.658333         Validation Loss: 3.606213         Time Took: 1
Saved!
Epoch: 38         Training Loss: 3.651324         Validation Loss: 3.588685         Time Took: 1
Saved!
Epoch: 39         Training Loss: 3.644745         Validation Loss: 3.641346         Time Took: 1
Epoch: 40         Training Loss: 3.613393         Validation Loss: 3.571905         Time Took: 1
Saved!
Epoch: 41         Training Loss: 3.577824         Validation Loss: 3.534857         Time Took: 1
Saved!
Epoch: 42         Training Loss: 3.551511         Validation Loss: 3.576112         Time Took: 1
Epoch: 43         Training Loss: 3.531660         Validation Loss: 3.435370         Time Took: 1
Saved!
Epoch: 44         Training Loss: 3.519716         Validation Loss: 3.445125         Time Took: 1
Epoch: 45         Training Loss: 3.479277         Validation Loss: 3.468793         Time Took: 1
Epoch: 46         Training Loss: 3.479031         Validation Loss: 3.410513         Time Took: 1
Saved!
Epoch: 47         Training Loss: 3.454519         Validation Loss: 3.381493         Time Took: 1
Saved!
Epoch: 48         Training Loss: 3.433129         Validation Loss: 3.399312         Time Took: 1
```

```
Epoch: 49         Training Loss: 3.394466         Validation Loss: 3.328353         Time Took: 1
Saved!
Epoch: 50         Training Loss: 3.381346         Validation Loss: 3.344064         Time Took: 1
Epoch: 51         Training Loss: 3.360281         Validation Loss: 3.300232         Time Took: 1
Saved!
Epoch: 52         Training Loss: 3.329239         Validation Loss: 3.245192         Time Took: 1
Saved!
Epoch: 53         Training Loss: 3.300626         Validation Loss: 3.325883         Time Took: 1
Epoch: 54         Training Loss: 3.297437         Validation Loss: 3.252809         Time Took: 1
Epoch: 55         Training Loss: 3.281258         Validation Loss: 3.247808         Time Took: 1
Epoch: 56         Training Loss: 3.240496         Validation Loss: 3.213053         Time Took: 1
Saved!
Epoch: 57         Training Loss: 3.215573         Validation Loss: 3.331187         Time Took: 1
Epoch: 58         Training Loss: 3.209703         Validation Loss: 3.189463         Time Took: 1
Saved!
Epoch: 59         Training Loss: 3.221619         Validation Loss: 3.168249         Time Took: 1
Saved!
Epoch: 60         Training Loss: 3.203685         Validation Loss: 3.211174         Time Took: 1
Epoch: 61         Training Loss: 3.181180         Validation Loss: 3.168700         Time Took: 1
Epoch: 62         Training Loss: 3.158722         Validation Loss: 3.096801         Time Took: 1
Saved!
Epoch: 63         Training Loss: 3.135061         Validation Loss: 3.153738         Time Took: 1
Epoch: 64         Training Loss: 3.105800         Validation Loss: 3.128957         Time Took: 1
Epoch: 65         Training Loss: 3.110203         Validation Loss: 3.058874         Time Took: 1
Saved!
Epoch: 66         Training Loss: 3.098205         Validation Loss: 3.106298         Time Took: 1
Epoch: 67         Training Loss: 3.055284         Validation Loss: 3.047528         Time Took: 1
Saved!
Epoch: 68         Training Loss: 3.044253         Validation Loss: 3.055920         Time Took: 1
Epoch: 69         Training Loss: 3.040889         Validation Loss: 3.012601         Time Took: 1
Saved!
Epoch: 70         Training Loss: 3.026214         Validation Loss: 3.043821         Time Took: 1
Epoch: 71         Training Loss: 3.030630         Validation Loss: 2.979015         Time Took: 1
Saved!
Epoch: 72         Training Loss: 3.015910         Validation Loss: 3.039057         Time Took: 1
Epoch: 73         Training Loss: 2.948396         Validation Loss: 3.057590         Time Took: 1
Epoch: 74         Training Loss: 2.953296         Validation Loss: 2.991109         Time Took: 1
Epoch: 75         Training Loss: 2.933637         Validation Loss: 3.056332         Time Took: 1
Epoch: 76         Training Loss: 2.948279         Validation Loss: 3.034649         Time Took: 1
Epoch: 77         Training Loss: 2.910864         Validation Loss: 2.979034         Time Took: 1
Epoch: 78         Training Loss: 2.925085         Validation Loss: 2.940932         Time Took: 1
Saved!
Epoch: 79         Training Loss: 2.904493         Validation Loss: 2.930520         Time Took: 1
Saved!
Epoch: 80         Training Loss: 2.883464         Validation Loss: 2.893702         Time Took: 1
Saved!
Epoch: 81         Training Loss: 2.853759         Validation Loss: 2.984715         Time Took: 1
Epoch: 82         Training Loss: 2.871481         Validation Loss: 2.928260         Time Took: 1
```

```
Epoch: 83         Training Loss: 2.867667         Validation Loss: 2.936177         Time Took: 1
Epoch: 84         Training Loss: 2.875197         Validation Loss: 2.908007         Time Took: 1
Epoch: 85         Training Loss: 2.848344         Validation Loss: 2.867977         Time Took: 1
Saved!
Epoch: 86         Training Loss: 2.841463         Validation Loss: 2.865034         Time Took: 1
Saved!
Epoch: 87         Training Loss: 2.807728         Validation Loss: 2.882562         Time Took: 1
Epoch: 88         Training Loss: 2.784172         Validation Loss: 2.947966         Time Took: 1
Epoch: 89         Training Loss: 2.764668         Validation Loss: 2.920759         Time Took: 1
Epoch: 90         Training Loss: 2.754492         Validation Loss: 2.836397         Time Took: 1
Saved!
Epoch: 91         Training Loss: 2.757014         Validation Loss: 2.913701         Time Took: 1
Epoch: 92         Training Loss: 2.780694         Validation Loss: 2.812080         Time Took: 1
Saved!
Epoch: 93         Training Loss: 2.736387         Validation Loss: 2.825394         Time Took: 1
Epoch: 94         Training Loss: 2.704051         Validation Loss: 2.803678         Time Took: 1
Saved!
Epoch: 95         Training Loss: 2.707713         Validation Loss: 2.719924         Time Took: 1
Saved!
Epoch: 96         Training Loss: 2.711509         Validation Loss: 2.844351         Time Took: 1
Epoch: 97         Training Loss: 2.703934         Validation Loss: 2.854764         Time Took: 1
Epoch: 98         Training Loss: 2.691585         Validation Loss: 2.777913         Time Took: 1
Epoch: 99         Training Loss: 2.673250         Validation Loss: 2.766931         Time Took: 1
Epoch: 100         Training Loss: 2.682129         Validation Loss: 2.883792         Time Took:
```

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 [58]: # load the model that got the best validation accuracy
         model_scratch.load_state_dict(torch.load('model_scratch.pt'))
         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(data, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 2.759789


Test Accuracy: 28% (238/836)
```

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 [11]: loaders_transfer= data
```

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

         ## TODO: Specify model architecture
```

```
model_transfer=models.vgg19(pretrained=True)

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

n_inputs = model_transfer.classifier[6].in_features
model_transfer.classifier[6]=nn.Linear(n_inputs,133)

for param in model_transfer.classifier[6].parameters():
    param.requires_grad = True

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

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

Answer:

```
<li>I have used the vgg19 model because I noticed in style transfer exercise that vgg19 models a
<li>Model architecture is the same as vgg19 expect sixth Linear layer out is changed to 133.</li
```

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 [23]:
```
from torch import optim

criterion_transfer = nn.CrossEntropyLoss()
optimizer_transfer = optim.Adam(model_transfer.classifier[6].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 [25]:
```
import time
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
        start=time.time()

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

```
            # train the model #
            ###################
            model.train()
            for batch_idx, (data, target) in enumerate(loaders['train']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## find the loss and update the model parameters accordingly
                ## record the average training loss, using something like
                ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo
                optimizer.zero_grad()
                out=model(data)
                loss=criterion(out,target)
                loss.backward()
                optimizer.step()
                train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.item() - train_los
            ######################
            # validate the model #
            ######################
            model.eval()
            for batch_idx, (data, target) in enumerate(loaders['valid']):
                # move to GPU
                if use_cuda:
                    data, target = data.cuda(), target.cuda()
                ## update the average validation loss
                out=model(data)
                loss=criterion(out,target)
                valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.item() - valid_los
            # print training/validation statistics
            print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tTime Took:
                epoch,
                train_loss,
                valid_loss,
                time.time()-start
                ))

            ## TODO: save the model if validation loss has decreased
            if valid_loss<valid_loss_min:
                torch.save(model.state_dict(), save_path)
                valid_loss_min=valid_loss
                print("Saved!")
        # return trained model
        return model
```

In [ ]:
```
# train the model
n_epochs=10
model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, c
```

```
Epoch: 1         Training Loss: 1.256571         Validation Loss: 0.541658         Time Took: 33
```

```
Saved!
Epoch: 2         Training Loss: 0.570307         Validation Loss: 0.424150         Time Took: 31
Saved!
Epoch: 3         Training Loss: 0.513456         Validation Loss: 0.428617         Time Took: 31
```

```
In [27]: # load the model that got the best validation accuracy (uncomment the line below)
         model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

1.1.16 (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 [29]: 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))

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

```
Test Loss: 0.420989
```

```
Test Accuracy: 87% (730/836)
```

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 [41]: ### 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]

         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             tran=transforms.Compose([transforms.CenterCrop(244),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],
                                                           [0.229, 0.224, 0.225])])
             image=Image.open(img_path)
             image=tran(image)
             image=image.unsqueeze(0)
             image=image.cuda()
             output=model_transfer(image).data.argmax()
             return class_names[output]
```

Step 5: Write your Algorithm

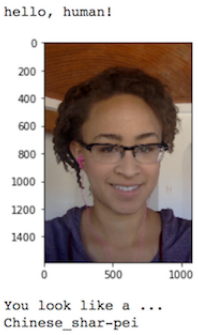
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then, - if a **dog** is detected in the image, return the predicted breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is 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 `human_detector` functions developed above. You are **required** to use your CNN from Step 4 to predict dog breed.

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [56]: ### TODO: Write your algorithm.
         ### Feel free to use as many code cells as needed.

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



Sample Human Output

```
if(dog==True):
    print("hello, dog")
    breed = predict_breed_transfer(img_path)
    print(f"You look like a...{breed}")
    image=Image.open(img_path)
    plt.imshow(image)
    plt.show()
elif(human==True):
    print("hello, human")
    image=Image.open(img_path)
    plt.imshow(image)
    plt.show()
else:
    print("You are neither human nor dog")
    image=Image.open(img_path)
    plt.imshow(image)
    plt.show()
```

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.

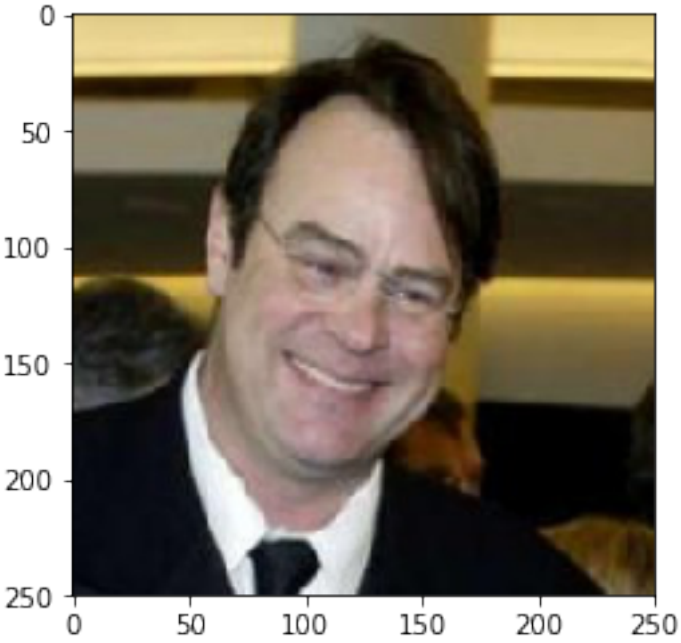
Answer: Output is better than i have expected :)

```
<h4>Possible points of improvement</h4>
<li>Increasing number of epoch</li>
<li>Increase number of CNN layer</li>
```

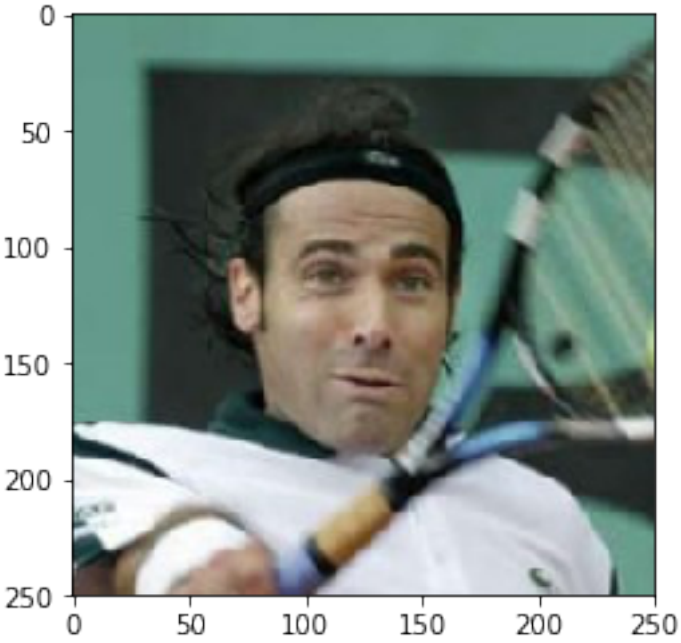
Use Image augmentation to increase number of input images
DropOut2d in convolutional network can be used for regularization

In [57]:
```
## 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)
```

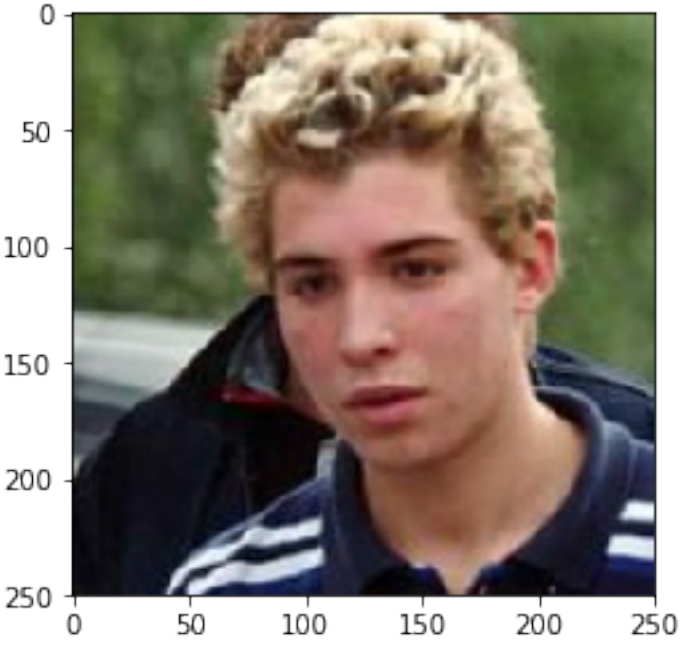
```
hello, human
```



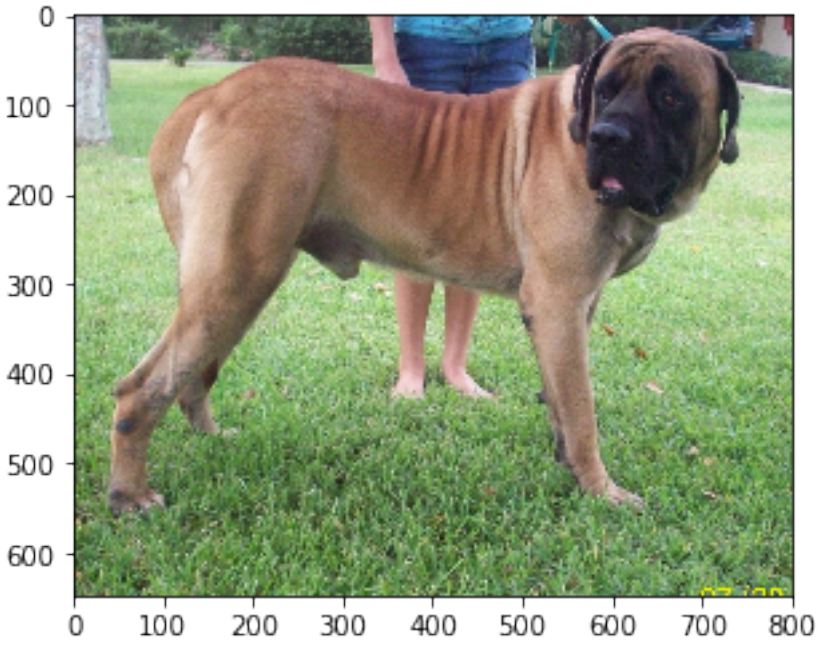

```
hello, human
```


0
50
100
150
200
250
0
50
100
150
200
250

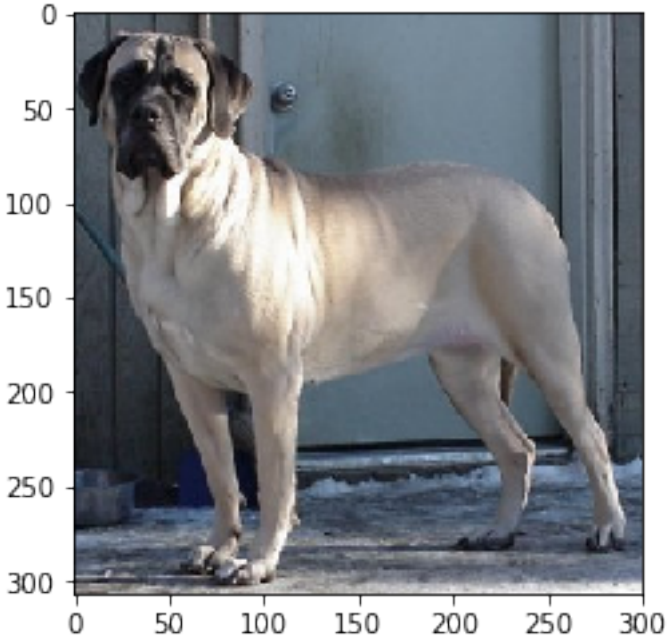
```
hello, human
```


0
50
100
150
200
250
0
50
100
150
200
250

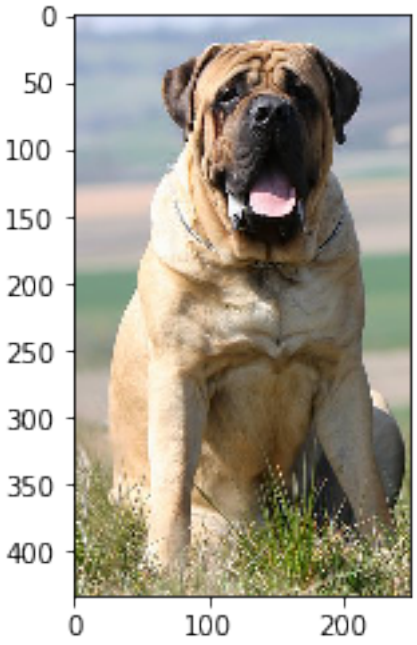
```
hello, dog
You look like a...German shepherd dog
```




```
hello, dog
You look like a...Mastiff
```



```
hello, dog
You look like a...Mastiff
```



In [ ]: