dog_app

May 17, 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 [2]: 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

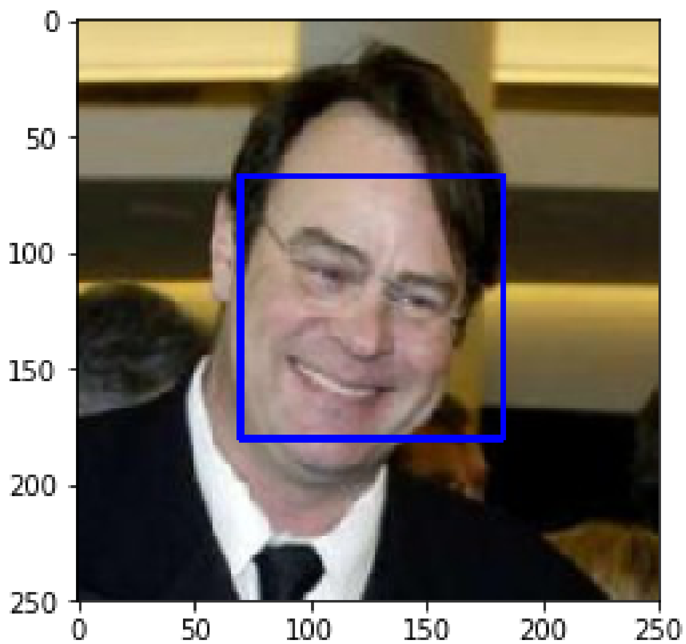
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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:
Percentage of Humans detected as Humans : 98 %
Percentage of Dogs detected as Humans : 17 %

```
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.

        correct = 0
        for img in human_files_short:
            if(face_detector(img)):
                correct += 1


        incorrect = 0
        for img in dog_files_short:
            if(face_detector(img)):
                incorrect += 1

        print("Percentage of Humans detected as Humans : " , correct , "%")
        print("Percentage of Dogs detected as Humans : " , incorrect , "%")
```

```
Percentage of Humans detected as Humans :  98 %
Percentage of Dogs detected as Humans :  17 %
```

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

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

1.1.3 Obtain Pre-trained VGG-16 Model

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

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

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

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

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

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 [60]: 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

             # Preprocessing-
                 # Resize the image
                 # Normalize the image with mean = [0.485, 0.456, 0.406] and std =  [0.229, 0.22
                 # Convert images to a form acceptable by pytorch i.e a Tensor

             '''https://gist.github.com/jkarimi91/d393688c4d4cdb9251e3f939f138876e'''
             '''https://github.com/pytorch/examples/blob/42e5b996718797e45c46a25c55b031e6768f844

             # get the image file
             file = Image.open(img_path)

             # establish the transform pipeline
             transform_pipeline = transforms.Compose([transforms.Resize((224 , 224)) , transform

             # transform the file and add dimension so that image is suitable to pass to VGG16 m
             img = transform_pipeline(file).unsqueeze(0)

             # Use GPU if available
             if use_cuda:
                 img = img.to('cuda')

             # get the prediction
             pred = VGG16(img)

             # get the index
             idx = pred.argmax()

             return idx # 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 [61]: ### returns "True" if a dog is detected in the image stored at img_path
         def dog_detector(img_path):
             ## TODO: Complete the function.
             return ((VGG16_predict(img_path) >= 151) and (VGG16_predict(img_path) <= 268))
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer:
Percentage of Dogs detected as Dogs : 100 %
Percentage of Humans detected as DOGS : 2 %

```
In [62]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.

         correct = 0
         for img in dog_files_short:
             if(dog_detector(img)):
                 correct += 1


         incorrect = 0
         for img in human_files_short:
             if(dog_detector(img)):
                 incorrect += 1

         print("Percentage of Dogs detected as Dogs : " , correct , "%")
         print("Percentage of Humans detected as DOGS : " , incorrect , "%")
```

```
Percentage of Dogs detected as Dogs :  100 %
Percentage of Humans detected as DOGS :  2 %
```

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 [19]: ### (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 [26]: import os
         import torchvision
         import torchvision.transforms as transforms
         from torchvision import datasets

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

         '''https://www.programcreek.com/python/example/105102/torchvision.datasets.ImageFolder'
         '''https://www.programcreek.com/python/example/104834/torchvision.transforms.Resize'''

         data_dir = '/data/dog_images/'

         # build the transforms dictionary
         transforms = {
             'train' : transforms.Compose([transforms.CenterCrop(224),
                                           transforms.Resize(224),
                                           transforms.RandomHorizontalFlip(p = 0.5),
                                           transforms.RandomRotation(degrees = (0 , 30)),
                                           transforms.ToTensor(),
                                           transforms.Normalize(mean = [0.485, 0.456, 0.406], std

             'valid' : transforms.Compose([transforms.Resize(224),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0

             'test' : transforms.Compose([transforms.Resize(224),
                                           transforms.CenterCrop(224),
                                           transforms.ToTensor(),
                                           transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0
         }

         # establish the paths to required folders
         train_dir = data_dir + '/train'
         valid_dir = data_dir + '/valid'
         test_dir = data_dir + '/test'

         # Get the images from respective folders
         image_datasets = {
             'train' : datasets.ImageFolder(root = train_dir , transform = transforms['train']),
             'valid' : datasets.ImageFolder(root = valid_dir , transform = transforms['valid']),
             'test' : datasets.ImageFolder(root = test_dir , transform = transforms['test'])
         }

         # Put into a Dataloader using torch library
         loaders_scratch = {
             'train' : torch.utils.data.DataLoader(image_datasets['train'] , batch_size = 50 , s
```

```
        'valid' : torch.utils.data.DataLoader(image_datasets['valid'] , batch_size = 50),
        'test' : torch.utils.data.DataLoader(image_datasets['test'] , batch_size = 50)
    }

    # Data Exploration
    total_len = len(image_datasets['train']) + len(image_datasets['test']) + len(image_data
    print("Total number of classes: " , len(image_datasets['train'].classes))
    print("Total number of train records: " , len(image_datasets['train']))
    print("Total number of test records: " , len(image_datasets['test']))
    print("Total number of valid records: " , len(image_datasets['valid']))
    print("Length of total dataset: " , total_len)
```

```
Total number of classes:  133
Total number of train records:  6680
Total number of test records:  836
Total number of valid records:  835
Length of total dataset:  8351
```

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: I faced a lot of problems while trying to specify data loders , finally I found a wonderful post (https://www.programcreek.com/python/example/105102/torchvision.datasets.ImageFolder) , and was able to do the job.

I performed data augmentation for the train , test and valid set using the following methods:

1. Train set : I first used CenterCrop to Crop the given Image at the center , the I resized the image to 224x224 the performed RandomHorizontalFlip with a probability of 0.5 , the I used RandomRotation with a minimum degree of 0 and a max degree of 30. Finally I converted the augmented image to a tensor and then normalized it.

2. Valid and Test sets : For these sets I did not perform augmentations as the model does not train on these images, I obnly cropped the images in this set at the center and the resized them to 224x224 before converting them to tensor and normalizing them

I selected a batch size of 50 , which I think is neither too high or too low , I found https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network especially helpful as it gives a good insight into how to select a proper batch size.

Finally I used shuffle on only the train loder as the purpose of shuffling is to avoid the model to train on images belonging to some specific classes , hence it has no effect on predictions.

Furthur I have also explored the data a little as printed above

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [27]: 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

                 #Convolutional Layers
                 self.conv1 = nn.Conv2d(in_channels = 3 , out_channels = 32 , kernel_size = 3 ,
                 self.conv2 = nn.Conv2d(in_channels = 32 , out_channels = 64 , kernel_size = 3 ,
                 self.conv3 = nn.Conv2d(in_channels = 64 , out_channels = 128 , kernel_size = 3
                 self.conv4 = nn.Conv2d(in_channels = 128 , out_channels = 256 , kernel_size = 3
                 self.conv5 = nn.Conv2d(in_channels = 256 , out_channels = 512 , kernel_size = 3

                 # dropout layer (p=0.3)
                 self.dropout = nn.Dropout(0.3)

                 # Linear Layers
                 self.fc1 = nn.Linear(in_features = 512*7*7 , out_features = 512)
                 self.fc2 = nn.Linear(in_features = 512 , out_features = 256)
                 self.out = nn.Linear(in_features = 256 , out_features = 133)


             def forward(self, x):
                 ## Define forward behavior
                 #input layer
                 x = x

                  # hidden conv layer 1
                 x = F.relu(self.conv1(x))
                 x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

                  # hidden conv layer 2
                 x = F.relu(self.conv2(x))
                 x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

                  # hidden conv layer 3
                 x = F.relu(self.conv3(x))
                 x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

                  # hidden conv layer 4
                 x = F.relu(self.conv4(x))
                 x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

                 # hidden conv layer 5
```

```
            x = F.relu(self.conv5(x))
            x = F.max_pool2d(x , kernel_size = 2 , stride = 2)

            # hidden linear layer 1
            x = x.reshape(-1 , 512*7*7) #must be flattened for 1st linear layer
            x = F.relu(self.fc1(x))
            x = self.dropout(x)

            # hidden linear layer 2
            x = F.relu(self.fc2(x))
            x = self.dropout(x)

            # outpul layer
            x = self.out(x)

            return x

        def __repr__(self): # Signature function
            return "Abhishek's net"

    #-#-# 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 [28]:
```
!pip install torchsummary

# Model Summary
from torchsummary import summary  # use "pip install torchsummary" in terminal if torch
model_summary = model_scratch.cuda() if use_cuda else model_scratch
summary(model_summary , (3 , 224 , 224))
```

```
Requirement already satisfied: torchsummary in /opt/conda/lib/python3.6/site-packages (1.5.1)
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 32, 224, 224]             896
            Conv2d-2         [-1, 64, 112, 112]          18,496
            Conv2d-3          [-1, 128, 56, 56]          73,856
            Conv2d-4          [-1, 256, 28, 28]         295,168
            Conv2d-5          [-1, 512, 14, 14]       1,180,160
            Linear-6                  [-1, 512]      12,845,568
           Dropout-7                  [-1, 512]               0
            Linear-8                  [-1, 256]         131,328
```

```
          Dropout-9                    [-1, 256]               0
          Linear-10                    [-1, 133]          34,181
================================================================
Total params: 14,579,653
Trainable params: 14,579,653
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 23.75
Params size (MB): 55.62
Estimated Total Size (MB): 79.94
----------------------------------------------------------------
```

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

Answer: My CNN architecture consists of 5 Convolutional layers and 3 Fully Connected Linear Layers. Each convolutional layer is followd by a max_pool2d function which ha a kernel size = 2 and stride = 2 , similarly each Linear layer except the output layer is followd by a dropout layer with a dropout probability of 0.3.

For the first conv layers there are 3 in-channels and 32 out channels , after that I just kept increasing the channels by a factor of 2. Each conv layer has a kernel size = 3 , stride = 1 and padding = 1.

After the tensor passes through the conv layers it reaches the 1st linear layer, there I flatten the tensor(the logic of flattening is described below) and pass it through the 1st linear layer, after that the tensor passes through a dropout layer and then again to a linear layer and finally to the output layer. Since the number of output classes are 133 (i.e the number of dog breeds in dataset) Hence the output layer has 133 out-features. Along with this I have included a signature function which prints "Abhishek's net" when print(model_scratch) is used , and in order to provide a summary of my model , I have used torchsummary module which is very similar to Keras's model.summary()

1.2 Logic for tensor flattening:

CNN Output Size Formula (Square)

```
. Suppose we have N x N input.
. Suppose we have f x f filter.
. Suppose we have a padding of p and a stride of s.
```

The output size O is given by this formula: ((N - f + 2*p) / s) + 1

This value will be the height and width of the output. However, if the input or the filter isn't a square, this formula needs to be applied twice, once for the width and once for the height.

Now we started with a tensor of (224 x 224) hence after the 1st conv. layer it's size is:

```
 O1 = ((224 - 3 + 2*1) / 1) + 1 = 224
```

But when this O1 passes thr. the mak_pool2d layer the output becomes:

```
O2 = ((O1 - 2  + 0) / 2) + 1 = ((224 - 2) / 2) + 1 = 112 , this is also evident from the mod
```

Hence like this the tensor reaches the 5th conv. layer as (14 x 14) and after that it passes through the final max_pool2d function it becomes (7 x 7), hence to flatten the tensor i use (512 (output_channels of last conv layer) x 7 x 7)

1.2.1 (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 [29]: import torch.optim as optim

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

         ### TODO: select optimizer

         # Selected the Adam optimizer and learning rate = 0.001 , Go to the link for more :
         '''https://medium.com/octavian-ai/which-optimizer-and-learning-rate-should-i-use-for-de

         optimizer_scratch = torch.optim.Adam(model_scratch.parameters() , lr = 0.001)
```

1.2.2 (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 [30]: 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()
        ## find the loss and update the model parameters accordingly
        ## record the average training loss, using something like
        ## train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_lo

        # clear the gradients , we dont want the previous direction to affect what
        optimizer.zero_grad()

        # get the predictions
        preds = model(data)

        # calculate the loss
        loss = criterion(preds , target)

        # back propogate to calculate loss gradient
        loss.backward()

        # optimize
        optimizer.step()

        # update the train_loss
        train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_loss)

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

        #no back prop. required in eval mode:
        with torch.no_grad():
            preds = model(data) # predictions

        # calculate the validation loss
        loss = criterion(preds , target)

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

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

```
                ))

            ## TODO: save the model if validation loss has decreased

            if valid_loss_min > valid_loss:
                torch.save(model.state_dict(), save_path)
                print('Model Saved : Previous minimum Validation Loss : {:.6f} \t:::\t New
                valid_loss_min,
                valid_loss))
                valid_loss_min = valid_loss

        # return trained model
        return model


    # train the model
    model_scratch = train(40, 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: 4.846717         Validation Loss: 4.778686
Model Saved : Previous minimum Validation Loss : inf         :::         New minimum Validation
Epoch: 2         Training Loss: 4.669957         Validation Loss: 4.652830
Model Saved : Previous minimum Validation Loss : 4.778686         :::         New minimum Valida
Epoch: 3         Training Loss: 4.583079         Validation Loss: 4.670770
Epoch: 4         Training Loss: 4.443510         Validation Loss: 4.525459
Model Saved : Previous minimum Validation Loss : 4.652830         :::         New minimum Valida
Epoch: 5         Training Loss: 4.346594         Validation Loss: 4.401968
Model Saved : Previous minimum Validation Loss : 4.525459         :::         New minimum Valida
Epoch: 6         Training Loss: 4.238697         Validation Loss: 4.387325
Model Saved : Previous minimum Validation Loss : 4.401968         :::         New minimum Valida
Epoch: 7         Training Loss: 4.131093         Validation Loss: 4.229009
Model Saved : Previous minimum Validation Loss : 4.387325         :::         New minimum Valida
Epoch: 8         Training Loss: 4.036647         Validation Loss: 4.223511
Model Saved : Previous minimum Validation Loss : 4.229009         :::         New minimum Valida
Epoch: 9         Training Loss: 3.940846         Validation Loss: 4.204190
Model Saved : Previous minimum Validation Loss : 4.223511         :::         New minimum Valida
Epoch: 10         Training Loss: 3.855445         Validation Loss: 4.317528
Epoch: 11         Training Loss: 3.770641         Validation Loss: 4.177781
Model Saved : Previous minimum Validation Loss : 4.204190         :::         New minimum Valida
Epoch: 12         Training Loss: 3.691767         Validation Loss: 4.129741
Model Saved : Previous minimum Validation Loss : 4.177781         :::         New minimum Valida
Epoch: 13         Training Loss: 3.614289         Validation Loss: 4.075459
Model Saved : Previous minimum Validation Loss : 4.129741         :::         New minimum Valida
Epoch: 14         Training Loss: 3.510494         Validation Loss: 4.088908
Epoch: 15         Training Loss: 3.454075         Validation Loss: 4.061162
```

```
Model Saved : Previous minimum Validation Loss : 4.075459         :::         New minimum Valida
Epoch: 16         Training Loss: 3.345827         Validation Loss: 4.073728
Epoch: 17         Training Loss: 3.279313         Validation Loss: 4.143792
Epoch: 18         Training Loss: 3.169544         Validation Loss: 4.374947
Epoch: 19         Training Loss: 3.105833         Validation Loss: 4.418487
Epoch: 20         Training Loss: 3.029780         Validation Loss: 4.108554
Epoch: 21         Training Loss: 2.947317         Validation Loss: 4.264196
Epoch: 22         Training Loss: 2.837724         Validation Loss: 4.317760
Epoch: 23         Training Loss: 2.775526         Validation Loss: 4.276844
Epoch: 24         Training Loss: 2.686486         Validation Loss: 4.480589
Epoch: 25         Training Loss: 2.601747         Validation Loss: 4.488660
Epoch: 26         Training Loss: 2.547263         Validation Loss: 4.397878
Epoch: 27         Training Loss: 2.469001         Validation Loss: 4.626958
Epoch: 28         Training Loss: 2.397267         Validation Loss: 4.982364
Epoch: 29         Training Loss: 2.298041         Validation Loss: 4.594360
Epoch: 30         Training Loss: 2.232332         Validation Loss: 4.853425
Epoch: 31         Training Loss: 2.190562         Validation Loss: 4.869522
Epoch: 32         Training Loss: 2.110886         Validation Loss: 4.928406
Epoch: 33         Training Loss: 2.068870         Validation Loss: 4.858846
Epoch: 34         Training Loss: 1.983660         Validation Loss: 4.914964
Epoch: 35         Training Loss: 1.937053         Validation Loss: 5.072016
Epoch: 36         Training Loss: 1.898889         Validation Loss: 4.963741
Epoch: 37         Training Loss: 1.825219         Validation Loss: 5.264280
Epoch: 38         Training Loss: 1.786697         Validation Loss: 5.057136
Epoch: 39         Training Loss: 1.763249         Validation Loss: 5.338131
Epoch: 40         Training Loss: 1.701443         Validation Loss: 5.240594
```

1.2.3 (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 [31]: 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: 4.083192


Test Accuracy: 11% (95/836)
```

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.

1.2.4 (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 [32]: ## TODO: Specify data loaders

         # Using the already created loaders
         loaders_transfer = loaders_scratch
```

1.2.5 (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 [40]: import torchvision.models as models
         import torch.nn as nn
```

```
## TODO: Specify model architecture
model_transfer = models.resnet50(pretrained = True)

'''https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html'''

# freeze the layers of already trained resnet50 model to avoid furthur training, parame
for param in model_transfer.parameters():
    param.requires_grad = False

# remove the last fc layer and add a new one insted which has 133 out_features correspo
num_ftrs = model_transfer.fc.in_features
model_transfer.fc = nn.Linear(num_ftrs , 133)

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: I am using a pretrained ResNet50 model to make predictions using Transfer Learning , I am using the already defined dataloders in loders_scratch.

ResNet50 is a deep residual network which is mostly used for image classification problems.It's a subclass of Convolutional Neural Network. The main innovation of resnet is the skip connection. I preferred it over VGG-16 and AlexNet as ResNet is way deeper than either of those 2 and has better chances of predicting correct labels.

Firstly, I downloaded the pretrained ResNet model then I froze all the layers of this model and finally replaced the last fc layer with a custom Linear Layer which ha 133 out_features corresponding to 133 dog breeds in our dataset.

1.2.6 (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 [42]: criterion_transfer = nn.CrossEntropyLoss()

         #optimize the parameters of only the final layer
         optimizer_transfer = optim.Adam(model_transfer.fc.parameters() , lr = 0.001)
```

1.2.7 (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 [44]: # train the model
         model_transfer = train(25, loaders_transfer, model_transfer, optimizer_transfer, criter

         # 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: 3.246610        Validation Loss: 1.490584
Model Saved : Previous minimum Validation Loss : inf         :::         New minimum Validation
Epoch: 2        Training Loss: 1.699598        Validation Loss: 0.998128
Model Saved : Previous minimum Validation Loss : 1.490584         :::         New minimum Valida
Epoch: 3        Training Loss: 1.361734        Validation Loss: 0.855585
Model Saved : Previous minimum Validation Loss : 0.998128         :::         New minimum Valida
Epoch: 4        Training Loss: 1.181321        Validation Loss: 0.793758
Model Saved : Previous minimum Validation Loss : 0.855585         :::         New minimum Valida
Epoch: 5        Training Loss: 1.089179        Validation Loss: 0.801911
Epoch: 6        Training Loss: 0.997274        Validation Loss: 0.733124
Model Saved : Previous minimum Validation Loss : 0.793758         :::         New minimum Valida
Epoch: 7        Training Loss: 0.930874        Validation Loss: 0.735071
Epoch: 8        Training Loss: 0.858002        Validation Loss: 0.778544
Epoch: 9        Training Loss: 0.822964        Validation Loss: 0.703604
Model Saved : Previous minimum Validation Loss : 0.733124         :::         New minimum Valida
Epoch: 10        Training Loss: 0.817733        Validation Loss: 0.701435
Model Saved : Previous minimum Validation Loss : 0.703604         :::         New minimum Valida
Epoch: 11        Training Loss: 0.761179        Validation Loss: 0.724134
Epoch: 12        Training Loss: 0.743986        Validation Loss: 0.745640
Epoch: 13        Training Loss: 0.722353        Validation Loss: 0.723333
Epoch: 14        Training Loss: 0.675358        Validation Loss: 0.702843
Epoch: 15        Training Loss: 0.648906        Validation Loss: 0.742256
Epoch: 16        Training Loss: 0.653876        Validation Loss: 0.728863
Epoch: 17        Training Loss: 0.591390        Validation Loss: 0.728065
Epoch: 18        Training Loss: 0.589451        Validation Loss: 0.727031
Epoch: 19        Training Loss: 0.576106        Validation Loss: 0.691191
Model Saved : Previous minimum Validation Loss : 0.701435         :::         New minimum Valida
Epoch: 20        Training Loss: 0.578342        Validation Loss: 0.751341
Epoch: 21        Training Loss: 0.557252        Validation Loss: 0.735404
Epoch: 22        Training Loss: 0.548881        Validation Loss: 0.743235
Epoch: 23        Training Loss: 0.508585        Validation Loss: 0.809770
Epoch: 24        Training Loss: 0.527154        Validation Loss: 0.770055
Epoch: 25        Training Loss: 0.499466        Validation Loss: 0.743271
```

1.2.8 (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 [45]: test(loaders_transfer, model_transfer, criterion_transfer, use_cuda)
```

```
Test Loss: 0.716861


Test Accuracy: 78% (659/836)
```

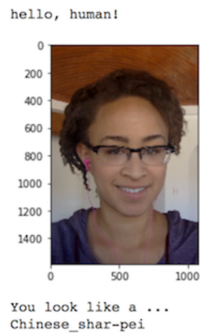
1.2.9 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [89]: ### 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 image_datasets['train'].classes]

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

             # get the image file
             file = Image.open(img_path)

             # bulild the transform pipeline
             in_transform_pipeline = transforms.Compose([transforms.Resize((224 , 224)),
                                                         transforms.CenterCrop((224,224)),
                                                         transforms.ToTensor(),
                                                         transforms.Normalize((0.485, 0.456, 0.40

             # transform the file and add dimension so that image is suitable
             img = in_transform_pipeline(file).unsqueeze(0)

              # use GPU if available
             if use_cuda:
                 img = img.cuda()

             # switching to evaluate mode for predictions
             model_transfer.eval()

             # no_grad used in sync with model.eval()
             with torch.no_grad():
                 res = model_transfer(img)
                 preds = torch.argmax(res)

             # preds.item() actual integer value
             return class_names[preds.item()]
```

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.



Sample Human Output

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!

1.2.10 (IMPLEMENTATION) Write your Algorithm

In [90]:
```
### 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
    breed = "it's all fuzzy in here, I am not sure about this image"
    if dog_detector(img_path):
        print("Hi there, dogo!")
        breed = "You look like a {} ".format(predict_breed_transfer(img_path))
    elif face_detector(img_path):
        print("Hi there, human!")
        breed = "Time for a fun fact, you look like a {} ".format(predict_breed_transfe

    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()
    print(breed)
    print("\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?

1.2.11 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

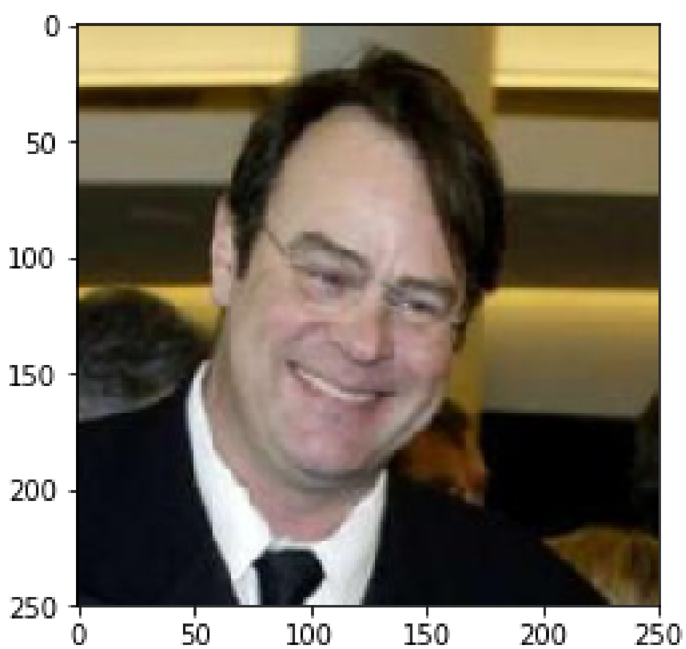
Answer: (Three possible points for improvement) The output is around the level I expected. With the CNN i made from scratch I got a mere 11% accuracy , but with fine tuning the ResNet50 model I got a 78% accuracy.

Still there is space for some improvement:

```
1) Hyperparameter optimization can increase accuracy.

2) A bigger training dataset and better data agumentation might result in better training and he

3) Using more epochs for training might result in better trained model.

4) Other evaluation metric can be tried

5) Model can be designed in such a way that it show the next most closest breed of the dog.
```

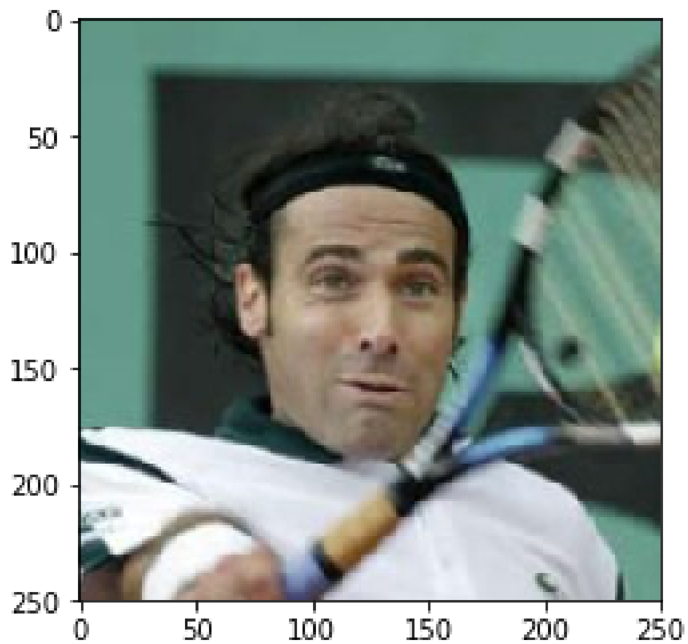
```
In [91]: ## TODO: Execute your algorithm from Step 6 on
         ## at least 6 images on your computer.
         ## Feel free to use as many code cells as needed.

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

```
Hi there, human!
```



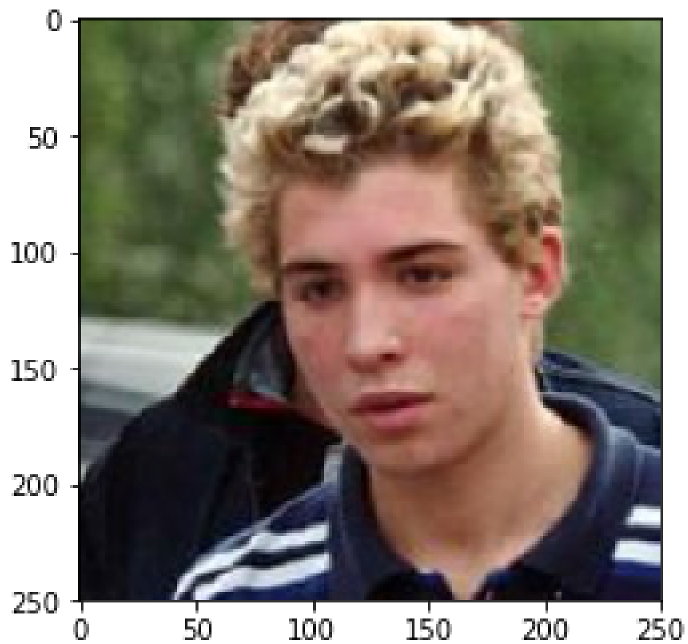
```
Time for a fun fact, you look like a Chihuahua
```

```
Hi there, human!
```



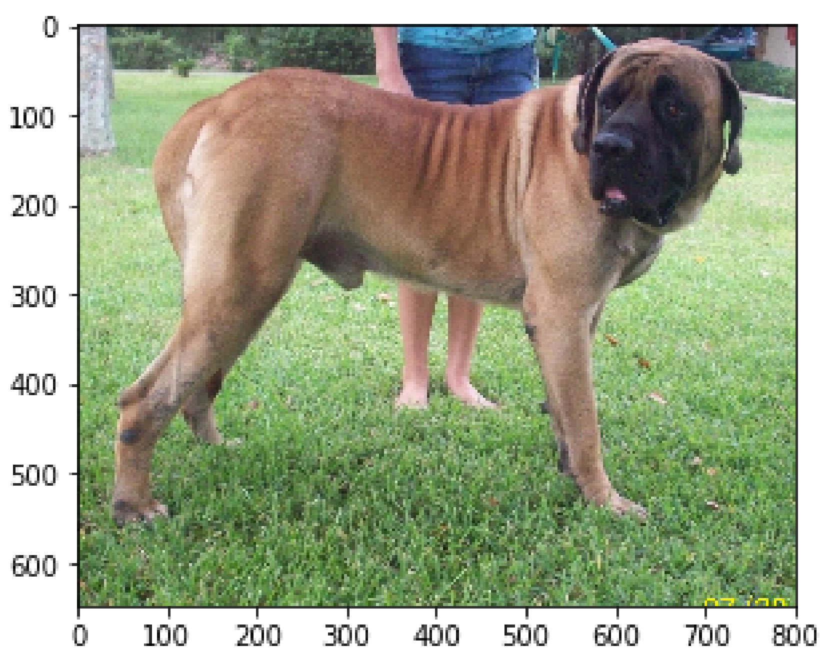
```
Time for a fun fact, you look like a English springer spaniel
```

```
Hi there, human!
```



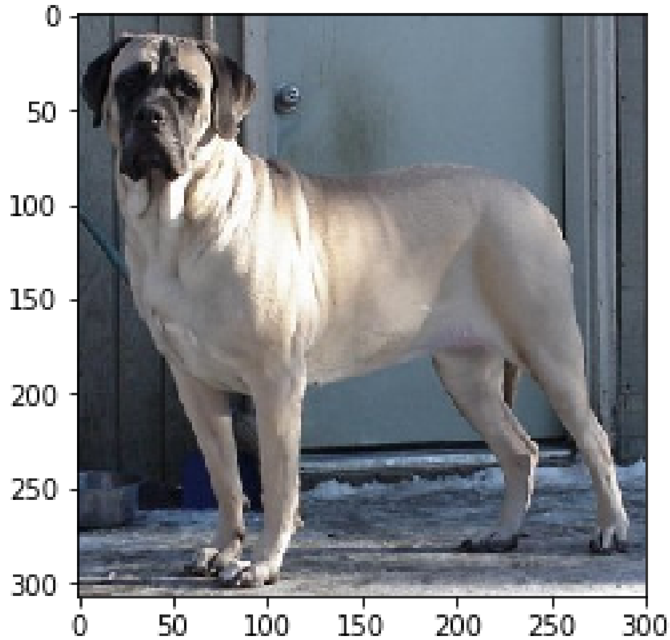
Time for a fun fact, you look like a Portuguese water dog

Hi there, dogo!



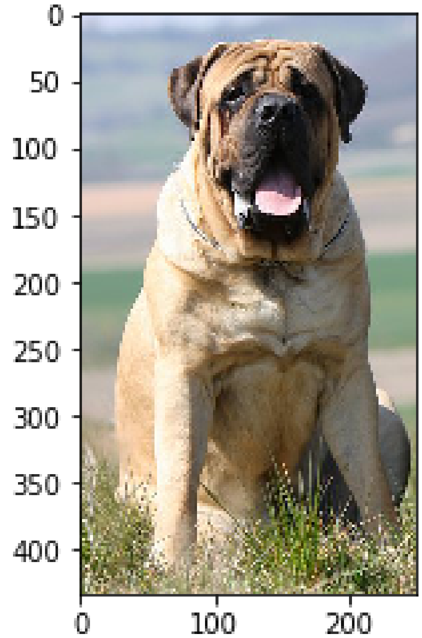
```
You look like a Bullmastiff
```

```
Hi there, dogo!
```




```
You look like a Bullmastiff
```

```
Hi there, dogo!
```




```
You look like a Mastiff
```

```
In [ ]:
```