dog_app

June 24, 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

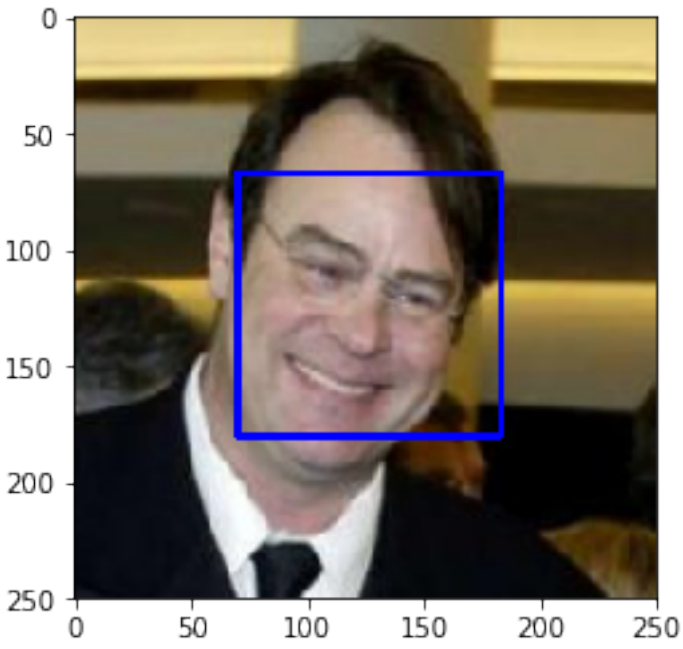
In this section, we use OpenCV's implementation of Haar feature-based cascade classifiers to detect human faces in images.

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 [2]: import cv2
        import matplotlib.pyplot as plt
        %matplotlib inline

        # extract pre-trained face detector
        face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

        # load color (BGR) image
        img = cv2.imread(human_files[0])
        # convert BGR image to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

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

        # 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()
```

```
[[ 70  67 113 113]]
Number of faces detected: 1
```



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

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

1.1.1 Write a Human Face Detector

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

```
In [3]: # returns "True" if face is detected in image stored at img_path
        def face_detector(img_path):
            img = cv2.imread(img_path)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            faces = face_cascade.detectMultiScale(gray)
            return len(faces) > 0
```

1.1.2 (IMPLEMENTATION) Assess the Human Face Detector

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

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

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

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

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

In [5]: #-#-# Do NOT modify the code above this line. #-#-#

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

        human_detected, dog_detected = 0,0
        for hum_path,dog_path in zip(human_files_short,dog_files_short):
            hum_detect = face_detector(hum_path)
            dog_detect = face_detector(dog_path)

            if (hum_detect):
                human_detected += 1
            if (dog_detect):
                dog_detected += 1

        print(f'Humans detected: {human_detected}% \t Dog detected" {dog_detected}%')

Humans detected: 98%          Dog detected" 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 [2]:
```
from tqdm import tqdm

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

In [3]:
```
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.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 [4]:
```
from PIL import Image
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
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

    #Open the image using Image
    img = Image.open(img_path)

    #Define the transform for the image to be turned into a tensor
    trans = transforms.Compose([transforms.Resize((224,224)),transforms.ToTensor()])

    #.unsqueeze(0) as tensors want a tuple of 4 dimensions for image input (#no.of image
    ten_img = trans(img).unsqueeze(0)

    #Move to GPU
    if use_cuda:
        ten_img = ten_img.cuda()

    #runs the image through vgg16 model. Gets the output as a [1,100] matrix.
    # Then converts the tensor into a numpy array and finds the most possible probabilit
    if use_cuda:
        answer = vgg16(ten_img).cpu().data.numpy().argmax()
    else:
        answer = vgg16(ten_img).data.numpy().argmax()

    return answer # return predicted class index

#Test Case to check vgg16_predict function
t_case = VGG16_predict(human_files_short[-1])
print(t_case)
```

```
678
```

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 [5]: ### returns "True" if a dog is detected in the image stored at img_path
        def dog_detector(img_path):
            ## TODO: Complete the function.

            #gets the image's possible class index
            answer = VGG16_predict(img_path)

            #checks if the index's class is a dog
            if (answer > 151) & (answer <= 268):
                return True
            return False # true/false
```

1.1.6 (IMPLEMENTATION) Assess the Dog Detector

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

Answer: 0% humans detected as dogs. 93% dogs detected as dogs.

```
In [6]: ### TODO: Test the performance of the dog_detector function
        ### on the images in human_files_short and dog_files_short.

        #Counter for detected human,dog as dog
        hum_detected,dog_detected = 0,0

        #Takes human,dog images and outputs %humans %dogs detected as dog
        for hum_path,dog_path in zip(human_files_short,dog_files_short):
            hum_detect = dog_detector(hum_path)
            dog_detect = dog_detector(dog_path)
            if (hum_detect):
                human_detected += 1
            if (dog_detect):
                dog_detected += 1
        print(f'Humans detected as dogs: {hum_detected}%\t Dogs detected as dogs: {dog_detected}
```

```
Humans detected as dogs: 0%         Dogs detected as dogs: 93%
```

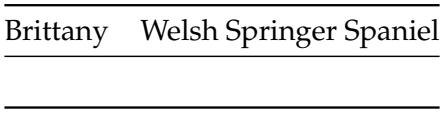
We suggest VGG-16 as a potential network to detect dog images in your algorithm, but you are free to explore other pre-trained networks (such as Inception-v3, ResNet-50, etc). Please use the code cell below to test other pre-trained PyTorch models. If you decide to pursue this *optional* task, report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

```
In [20]: ### (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)

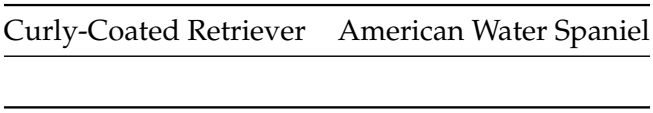
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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



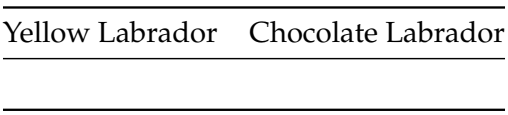
Brittany Welsh Springer Spaniel

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 [1]:
```
import os
import torch
from torchvision import datasets,transforms
from PIL import ImageFile #Help load truncated files better
ImageFile.LOAD_TRUNCATED_IMAGES = True

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


# Directories of train,valid and test datasets of Dogs
train_dir = '/data/dog_images/train'
valid_dir = '/data/dog_images/valid'
test_dir = '/data/dog_images/test'


#Apply different transformations for train and valid data to train and check the model b
train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.RandomResizedCrop(224),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],[0.229

valid_transforms = transforms.Compose([transforms.Resize(224),
                                       transforms.CenterCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.485, 0.456, 0.406],[0.229

#No need to drastically transform test data. Only resizes it to [224,224] and converts i
test_transforms = transforms.Compose([transforms.Resize(224),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406],[0.229,


#Loading Datasets from their directories.
train_datasets = datasets.ImageFolder(train_dir, transform = train_transforms)
valid_datasets = datasets.ImageFolder(valid_dir, transform=valid_transforms)
test_datasets = datasets.ImageFolder(test_dir, transform=test_transforms)
```

```
#Making a sub-batch based dataloaders for efficient training and testing
loaders = {'train':torch.utils.data.DataLoader(train_datasets, batch_size=24, shuffle=Tr
           'valid':torch.utils.data.DataLoader(valid_datasets, batch_size=24, shuffle=Tr
           'test':torch.utils.data.DataLoader(test_datasets, batch_size =24, shuffle=Tru
```

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

Answer: 1. My code resizes my code with a bunch of transformations. Firstly, it randomly rotates my input image by 30 degrees either direction. Then it randomly resizes it into a 244x244 image. Afterwards, It randomly flips the image horizontaly and converts it into a tensor. Finally, it normalizes the tensor to help run the data through the model faster. I picked a 224x224 tensor for my input tensor because thats the input data requirement for the vgg16 model.

2. I did decide to augment the dataset. I detailed the transformations in my previous answer. I decided to do it because real world images are going to be taken in many directions and will focus the content in different parts of the image. Hence, doing random transforms on our train-data makes it emulate real-data and help make our model more efficient and work better on real-world data.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```
In [2]: import torch.nn as nn
        import torch
        import torch.nn.functional as F
        import numpy as np
        # define the CNN architectureQ
        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, 64, 3, padding=1)
                self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
                self.conv3 = nn.Conv2d(128, 256, 3, padding=1)

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

                self.fc1 = nn.Linear(28*28*256,512)
                self.fc2 = nn.Linear(512, 256)
                self.fc3 = nn.Linear(256, 133)

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

            def forward(self, x):
```

```
        ## Define forward behavior
        #Flattens the images into vectors
        x = F.relu(self.conv1(x))
        x = self.pool(x)

        x = F.relu(self.conv2(x))
        x = self.pool(x)

        x = F.relu(self.conv3(x))
        x = self.pool(x)

        x = x.view(x.shape[0], -1)
        #print(x.shape)
        x = self.dropout(F.relu(self.fc1(x)))
        x = self.dropout(F.relu(self.fc2(x)))
        x = self.fc3(x)
        return x

#Initial weights are assigned from a normal distribution to both CNN and Linear Layers
def weight_initialization_normal(m):
    classname = m.__class__.__name__
    if classname.find('conv1') != -1 | classname.find('Linear') != -1:
        n = m.in_features
        y = 1/np.sqrt(n)
        m.weight.data.normal_(0.0, y)
        m.bias.fill_(0)

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

# instantiate the CNN
model_scratch = Net()

#Gives better initial weights
model_scratch.apply(weight_initialization_normal)

# move tensors to GPU if CUDA is available
use_cuda = torch.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: I choose a slightly complex CNN architecture with about 3 CNN layers with Max-pooling for detecting features and 3 fully connected layers for classifier. Based on the example images, I thought that in order to differentiate similar breeds, you would need to gather and preserve higher details among the image as it makes it way from feature detectors to the classifier. Hence, for the feature detector, I used CNN Layers with 64,128,256 filters with 3x3 kernel size, padding

of 1, stride of 0 and maxpooling of size 2 and stride 2. The maxpooling is there to reduce the input feature and gather high-quality details out. So these feature detectors allowed me to gather high quality features towards the classifier. Then for the classifier I choose 3 fully connected hidden linear layers of nodes 516, 256, 133. This level of complexitiy I hoped to help the model classify similar looking images better. I also choose to use dropout of about 0.25 on my classifier's layers to help it generalize more. Also I choose to apply a weight initializaiton for both CNN and Linear layers by choosing weights from a normal distribution of y = 1/np.sqrt(m.in_features) to help my model have a better start.

1.1.9 (IMPLEMENTATION) Specify Loss Function and Optimizer

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

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

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

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

In [4]:
```
#Test sequence
        import numpy as np
        x,y = next(iter(loaders['train']))
        print(x.shape)
        output = model_scratch(x.cuda())
        output.shape
```

```
torch.Size([24, 3, 224, 224])
```

Out[4]:
```
torch.Size([24, 133])
```

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 [5]:
```
import numpy as np

        valid_losses =[]
        valid_losses.append(0)
        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

    optimizer.zero_grad() #reset all gradient steps to 0

    output = model.forward(data) #Run the image through the model

    loss= criterion(output, target) #Calculate loss

    loss.backward() #Performs back propagation

    optimizer.step()

    #Calculates average loss for finished batches
    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()

    #Calculate Output and Loss
    output = model(data)
    loss = criterion(output, target)

    ## update the average validation 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 < valid_loss_min:
                    torch.save(model.state_dict(), save_path)
                    valid_loss_min = valid_loss
                    print('Low Valid Loss detected... Saving model')

            # return trained model
            state_dict = torch.load(save_path)
            return model.load_state_dict(state_dict)
```

In [18]:
```
# train the model
model_scratch = train(50, loaders, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')
```

```
Epoch: 1         Training Loss: 4.884542         Validation Loss: 4.868464
Low Valid Loss detected... Saving model
Epoch: 2         Training Loss: 4.845132         Validation Loss: 4.752935
Low Valid Loss detected... Saving model
Epoch: 3         Training Loss: 4.735743         Validation Loss: 4.539616
Low Valid Loss detected... Saving model
Epoch: 4         Training Loss: 4.672903         Validation Loss: 4.531349
Low Valid Loss detected... Saving model
Epoch: 5         Training Loss: 4.633700         Validation Loss: 4.505930
Low Valid Loss detected... Saving model
Epoch: 6         Training Loss: 4.592331         Validation Loss: 4.460260
Low Valid Loss detected... Saving model
Epoch: 7         Training Loss: 4.558197         Validation Loss: 4.423133
Low Valid Loss detected... Saving model
Epoch: 8         Training Loss: 4.537290         Validation Loss: 4.620822
Epoch: 9         Training Loss: 4.522281         Validation Loss: 4.403786
Low Valid Loss detected... Saving model
Epoch: 10         Training Loss: 4.488481         Validation Loss: 4.450564
Epoch: 11         Training Loss: 4.467666         Validation Loss: 4.380539
Low Valid Loss detected... Saving model
Epoch: 12         Training Loss: 4.434833         Validation Loss: 4.240440
Low Valid Loss detected... Saving model
Epoch: 13         Training Loss: 4.393126         Validation Loss: 4.177338
Low Valid Loss detected... Saving model
Epoch: 14         Training Loss: 4.382843         Validation Loss: 4.337306
Epoch: 15         Training Loss: 4.354851         Validation Loss: 4.141783
Low Valid Loss detected... Saving model
Epoch: 16         Training Loss: 4.297634         Validation Loss: 4.534806
Epoch: 17         Training Loss: 4.302651         Validation Loss: 4.155287
Epoch: 18         Training Loss: 4.291013         Validation Loss: 4.157404
Epoch: 19         Training Loss: 4.250319         Validation Loss: 3.977001
Low Valid Loss detected... Saving model
Epoch: 20         Training Loss: 4.202672         Validation Loss: 4.021577
```

```
Epoch: 21         Training Loss: 4.182600         Validation Loss: 4.028958
Epoch: 22         Training Loss: 4.166903         Validation Loss: 4.434503
Epoch: 23         Training Loss: 4.146718         Validation Loss: 3.980272
Epoch: 24         Training Loss: 4.109214         Validation Loss: 4.106636
Epoch: 25         Training Loss: 4.099525         Validation Loss: 4.099988
Epoch: 26         Training Loss: 4.064845         Validation Loss: 3.828310
Low Valid Loss detected... Saving model
Epoch: 27         Training Loss: 4.033821         Validation Loss: 3.997260
Epoch: 28         Training Loss: 4.019220         Validation Loss: 3.982440
Epoch: 29         Training Loss: 3.965435         Validation Loss: 4.294441
Epoch: 30         Training Loss: 3.977075         Validation Loss: 4.204819
Epoch: 31         Training Loss: 3.947058         Validation Loss: 3.753407
Low Valid Loss detected... Saving model
Epoch: 32         Training Loss: 3.908624         Validation Loss: 3.618141
Low Valid Loss detected... Saving model
Epoch: 33         Training Loss: 3.883667         Validation Loss: 3.634668
Epoch: 34         Training Loss: 3.890865         Validation Loss: 3.677351
Epoch: 35         Training Loss: 3.850270         Validation Loss: 3.777857
Epoch: 36         Training Loss: 3.838882         Validation Loss: 3.884823
Epoch: 37         Training Loss: 3.827276         Validation Loss: 3.622040
Epoch: 38         Training Loss: 3.796183         Validation Loss: 3.833380
Epoch: 39         Training Loss: 3.761102         Validation Loss: 3.594259
Low Valid Loss detected... Saving model
Epoch: 40         Training Loss: 3.753093         Validation Loss: 3.558970
Low Valid Loss detected... Saving model
Epoch: 41         Training Loss: 3.724388         Validation Loss: 3.491988
Low Valid Loss detected... Saving model
Epoch: 42         Training Loss: 3.708239         Validation Loss: 3.521087
Epoch: 43         Training Loss: 3.688759         Validation Loss: 3.384214
Low Valid Loss detected... Saving model
Epoch: 44         Training Loss: 3.674289         Validation Loss: 3.340525
Low Valid Loss detected... Saving model
Epoch: 45         Training Loss: 3.645022         Validation Loss: 3.463500
Epoch: 46         Training Loss: 3.638303         Validation Loss: 3.357289
Epoch: 47         Training Loss: 3.639471         Validation Loss: 3.524066
Epoch: 48         Training Loss: 3.594732         Validation Loss: 3.502743
Epoch: 49         Training Loss: 3.600802         Validation Loss: 3.412562
Epoch: 50         Training Loss: 3.571666         Validation Loss: 3.620538
```

```
In [6]: #load the model that got the best validation accuracy
        model_scratch.load_state_dict(torch.load('model_scratch.pt'))
```

1.1.11 (IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Use the code cell below to calculate and print the test loss and accuracy. Ensure that your test accuracy is greater than 10%.

```
In [1]: import numpy as np
        def test(loaders, model, criterion, use_cuda):

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

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

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

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

In [8]: # call test function
        test(loaders, model_scratch, criterion_scratch, use_cuda)
```

```
Test Loss: 3.345171


Test Accuracy: 19% (161/836)
```

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

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

1.1.12 (IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

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

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

```
In [2]: import torch
        from torchvision import datasets,transforms
        from PIL import ImageFile #Help load truncated files better
        ImageFile.LOAD_TRUNCATED_IMAGES = True

        # Directories of train,valid and test datasets of Dogs
        train_dir = '/data/dog_images/train'
        valid_dir = '/data/dog_images/valid'
        test_dir = '/data/dog_images/test'


        #Apply different transformations for train and valid data to train and check the model b
        train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                               transforms.RandomResizedCrop(224),
                                               transforms.RandomHorizontalFlip(),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],[0.229

        valid_transforms = transforms.Compose([transforms.Resize(224),
                                               transforms.CenterCrop(224),
                                               transforms.ToTensor(),
                                               transforms.Normalize([0.485, 0.456, 0.406],[0.229

        #No need to drastically transform test data. Only resizes it to [224,224] and converts i
        test_transforms = transforms.Compose([transforms.Resize(224),
                                              transforms.CenterCrop(224),
                                              transforms.ToTensor(),
                                              transforms.Normalize([0.485, 0.456, 0.406],[0.229,


        #Loading Datasets from their directories.
        train_datasets = datasets.ImageFolder(train_dir, transform = train_transforms)
        valid_datasets = datasets.ImageFolder(valid_dir, transform=valid_transforms)
        test_datasets = datasets.ImageFolder(test_dir, transform=test_transforms)


        #Making a sub-batch based dataloaders for efficient training and testing
        loaders_transfer = {'train':torch.utils.data.DataLoader(train_datasets, batch_size=32, s
                    'valid':torch.utils.data.DataLoader(valid_datasets, batch_size=32, shuffle=Tr
                    'test':torch.utils.data.DataLoader(test_datasets, batch_size =32, shuffle=Tru
```

1.1.13 (IMPLEMENTATION) Model Architecture

Use transfer learning to create a CNN to classify dog breed. Use the code cell below, and save your initialized model as the variable `model_transfer`.

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

        ## TODO: Specify model architecture

        #Gets vgg16 pretrained model from pytorch
        model_transfer = models.vgg16(pretrained=True)


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

        #Freezes all parameters in the pre-trained network
        for param in model_transfer.features.parameters():
            param.requires_grad = False

        #Makes our custom last fully-connected layer
        in_features = model_transfer.classifier[6].in_features
        last_fc = nn.Linear(in_features, 133)

        #Set the last layer to custom_fc layer
        model_transfer.classifier[6] = last_fc

        #if cuda is avaliable, transfers the model over to gpu for faster training
        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:

Firstly, for choosing the model architecture, I scoured the torchvision.models for possible models. I choose vgg16 model as it was trained on ImageNet. ImageNet is a massive dataset with over 14 million images around 1000 different classes. Hence, as vgg16 model was trained on that dataset, it can detect high-level features and preserve it and send it to the next layer. As some breeds of dogs are hard to differentiate even with a human eye, I thought that using a vgg16's feature detector would help preserve the small intricate details useful to train the model on.

Vgg16 has a two parts: feature detector and classifier. Feature detector is made out of CNN networks together with ReLU activations and MaxPooling layers. Classifier is is a chain of fully-connected layers that use the output from the feature detector and classify it into one of the 1000 classes. As the vgg16 model has a huge array of cnn, fc layers, I decided it would be inefficient to re-train the entire model to suit my needs. Instead, I decided to fine tune the model by freezing the model's feature detector's weight in place (with pre-trained weights), change the last layer's outfeatures to 133(no.of dog breeds in my dataset) and let the model's classifier train on my dog-breed images. I think this is going to work as as the ImageNet's database has a wide variety of classes with some portion centered around animals. My dog-breed database isn't too different from some portion of imagenet's classes. Hence, by only fine-tuning the model's classifier on my dog-breed dataset, I will reach an optimal solution.

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 [9]: import torch.optim as optim
        criterion_transfer = nn.CrossEntropyLoss()
        optimizer_transfer = optim.SGD(model_transfer.classifier.parameters(),lr=0.0015,momentum
```

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [5]: # train the model
        n_epochs=50
        model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer, c
```

```
Epoch: 1         Training Loss: 2.316349         Validation Loss: 0.651877
Low Valid Loss detected... Saving model
Epoch: 2         Training Loss: 1.290015         Validation Loss: 0.535039
Low Valid Loss detected... Saving model
Epoch: 3         Training Loss: 1.161189         Validation Loss: 0.495322
Low Valid Loss detected... Saving model
Epoch: 4         Training Loss: 1.074726         Validation Loss: 0.443386
Low Valid Loss detected... Saving model
Epoch: 5         Training Loss: 1.019462         Validation Loss: 0.433022
Low Valid Loss detected... Saving model
Epoch: 6         Training Loss: 0.985240         Validation Loss: 0.429575
Low Valid Loss detected... Saving model
Epoch: 7         Training Loss: 0.959299         Validation Loss: 0.499135
Epoch: 8         Training Loss: 0.922601         Validation Loss: 0.412784
Low Valid Loss detected... Saving model
Epoch: 9         Training Loss: 0.899207         Validation Loss: 0.457792
Epoch: 10         Training Loss: 0.871882         Validation Loss: 0.503775
Epoch: 11         Training Loss: 0.868175         Validation Loss: 0.387778
Low Valid Loss detected... Saving model
Epoch: 12         Training Loss: 0.841625         Validation Loss: 0.453466
Epoch: 13         Training Loss: 0.842980         Validation Loss: 0.422583
Epoch: 14         Training Loss: 0.814727         Validation Loss: 0.404163
Epoch: 15         Training Loss: 0.806877         Validation Loss: 0.362305
Low Valid Loss detected... Saving model
Epoch: 16         Training Loss: 0.768546         Validation Loss: 0.425365
Epoch: 17         Training Loss: 0.800780         Validation Loss: 0.345710
Low Valid Loss detected... Saving model
Epoch: 18         Training Loss: 0.788741         Validation Loss: 0.364102
Epoch: 19         Training Loss: 0.758962         Validation Loss: 0.387419
Epoch: 20         Training Loss: 0.735984         Validation Loss: 0.400528
Epoch: 21         Training Loss: 0.754520         Validation Loss: 0.403025
```

```
Epoch: 22         Training Loss: 0.755402         Validation Loss: 0.356529
Epoch: 23         Training Loss: 0.753549         Validation Loss: 0.356529
Epoch: 24         Training Loss: 0.724324         Validation Loss: 0.366815
Epoch: 25         Training Loss: 0.702088         Validation Loss: 0.396262
Epoch: 26         Training Loss: 0.731700         Validation Loss: 0.364357
Epoch: 27         Training Loss: 0.682460         Validation Loss: 0.394089
Epoch: 28         Training Loss: 0.712649         Validation Loss: 0.386379
Epoch: 29         Training Loss: 0.672475         Validation Loss: 0.357134
Epoch: 30         Training Loss: 0.693368         Validation Loss: 0.346681
Epoch: 31         Training Loss: 0.655731         Validation Loss: 0.352256
Epoch: 32         Training Loss: 0.662580         Validation Loss: 0.370489
Epoch: 33         Training Loss: 0.681083         Validation Loss: 0.378477
Epoch: 34         Training Loss: 0.673056         Validation Loss: 0.399207
Epoch: 35         Training Loss: 0.658535         Validation Loss: 0.357411
Epoch: 36         Training Loss: 0.632807         Validation Loss: 0.407068
Epoch: 37         Training Loss: 0.621175         Validation Loss: 0.371864
Epoch: 38         Training Loss: 0.623370         Validation Loss: 0.427159
Epoch: 39         Training Loss: 0.661687         Validation Loss: 0.434035
Epoch: 40         Training Loss: 0.602454         Validation Loss: 0.335837
Low Valid Loss detected... Saving model
Epoch: 41         Training Loss: 0.619770         Validation Loss: 0.513223
Epoch: 42         Training Loss: 0.596735         Validation Loss: 0.367060
Epoch: 43         Training Loss: 0.625916         Validation Loss: 0.357478
Epoch: 44         Training Loss: 0.621625         Validation Loss: 0.380867
Epoch: 45         Training Loss: 0.627297         Validation Loss: 0.387039
Epoch: 46         Training Loss: 0.616041         Validation Loss: 0.373547
Epoch: 47         Training Loss: 0.591085         Validation Loss: 0.359034
Epoch: 48         Training Loss: 0.601113         Validation Loss: 0.364374
Epoch: 49         Training Loss: 0.582133         Validation Loss: 0.382738
Epoch: 50         Training Loss: 0.592723         Validation Loss: 0.401473
```

```
In [4]: if torch.cuda.is_available():
            map_location=lambda storage, loc: storage.cuda()
        else:
            map_location='cpu'

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

In [5]: model_transfer #Checks the model is up and running

Out[5]: VGG(
          (features): Sequential(
            (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (1): ReLU(inplace)
            (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
            (3): ReLU(inplace)
```

```
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace)
    (2): Dropout(p=0.5)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace)
    (5): Dropout(p=0.5)
    (6): Linear(in_features=4096, out_features=133, bias=True)
  )
)
```

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

```
Test Loss: 0.474464
```

```
Test Accuracy: 86% (727/836)
```

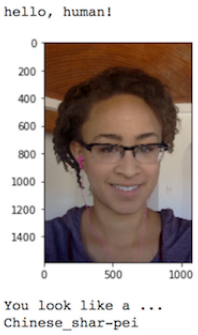
1.1.17 (IMPLEMENTATION) Predict Dog Breed with the Model

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 [6]: from PIL import Image


        ### 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].items[4]
        class_names = [item[4:].replace("_", " ") for item in train_datasets.classes]

        def predict_breed_transfer(img_path):
            # load the image and return the predicted breed
            img = Image.open(img_path)

            #transforms to change image into a tensor input for model
            trans = transforms.Compose([transforms.Resize((224,224)),
                                        transforms.ToTensor(),
                                        transforms.Normalize([0.485, 0.456, 0.406],
                                                             [0.229, 0.224, 0.225])])
            #applies transforms and gets the image into tensor format (no.of images, filters, le
            img = trans(img).unsqueeze(0)

            #Moves img to gpu if cuda is available
            if use_cuda:
                img = img.cuda()

            #Checks if cuda is available
            #Gets the output onto cpu, gets the probabilites, converts those probabilities into
            if use_cuda:
                output = model_transfer(img).cpu().detach().numpy().argmax()
            else:
                output = model_transfer(img).detach().numpy().argmax()

            return class_names[output] #gets the classname by searching in class_names dictionar
```

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



Sample Human Output

breed. - if a **human** is detected in the image, return the resembling dog breed. - if **neither** is detected in the image, provide output that indicates an error.

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

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

1.1.18 (IMPLEMENTATION) Write your Algorithm

In [13]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.xml')

# 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

### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    #gets the image's possible class index
    answer = VGG16_predict(img_path)

    #checks if the index's class is a dog
    if (answer > 151) & (answer <= 268):
        return True
    return False # true/false
```

```
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    #hum_img_path, dog_img_path = img_path
    name = 'Dog'
    if face_detector(img_path):
        name = 'Human'

    img = plt.imread(img_path)

    print('====================== Dog Breed Classifier [v1] by SD ======================
    print(f'Hello, {name}')
    plt.imshow(img,cmap='jet')
    plt.show()
    print('You look like a ...', predict_breed_transfer(img_path),'\n' )
```

Step 6: Test Your Algorithm

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

1.1.19 (IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

Answer: (Three possible points for improvement) Honestly, I was initially suprised about how high of an accuracy I got on my test earlier and before with the test function. I trained my initial CNN Network for 3 hours on Udacity's GPU only to get a disappointing 14% accuracy. However, utilizing pytorch's pretrained vgg16 model and some fine tuning, I was able to get around 88% test accuracy by only training it for 2 hours.

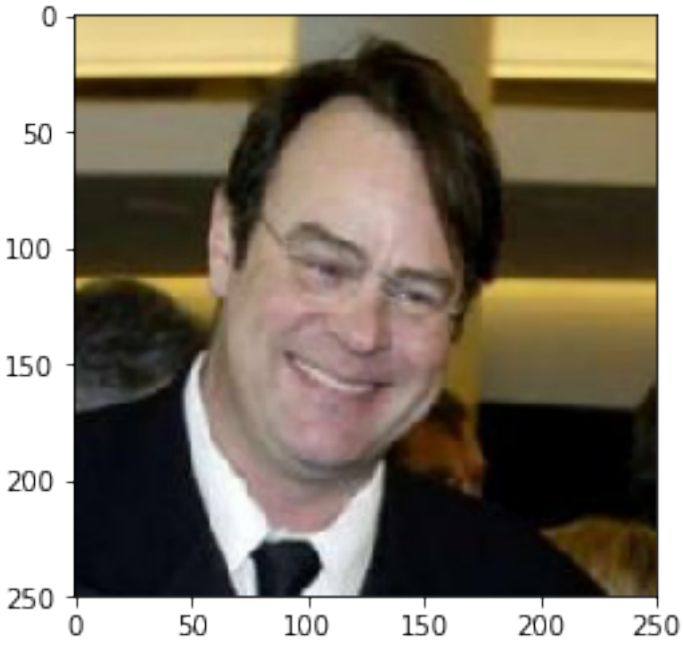
Although that was a drastic improvement from my earlier model, there are some ways I can improve them. Let me list them below.

- 1. **Get better optimizer or optimizer's hyperparameters**. For my model, I choose to use a Stocastic gradient descent optimizer with a learning rate of 0.0015 and momentum of 0.9. I choose this learning rate and momentum to accelerate my gradients into the optimal value. While I arrived near my best model at 40th of the 50th iteration with a valid loss of 0.336, my second best model arrived at a valid loss of 0.346 at the 17th out of 50th iteration. So it seems to me that I could have work
- 2. **Gather More data**. Although the dataset has a hefty amount of images, they are sorted into 133 different categories. In order to improve the model, we can consider gathering more and varying data about the 133 different dogs. Hence with more and varying data we could help the model find the differing intricate patterns in dogs and help classify them better.

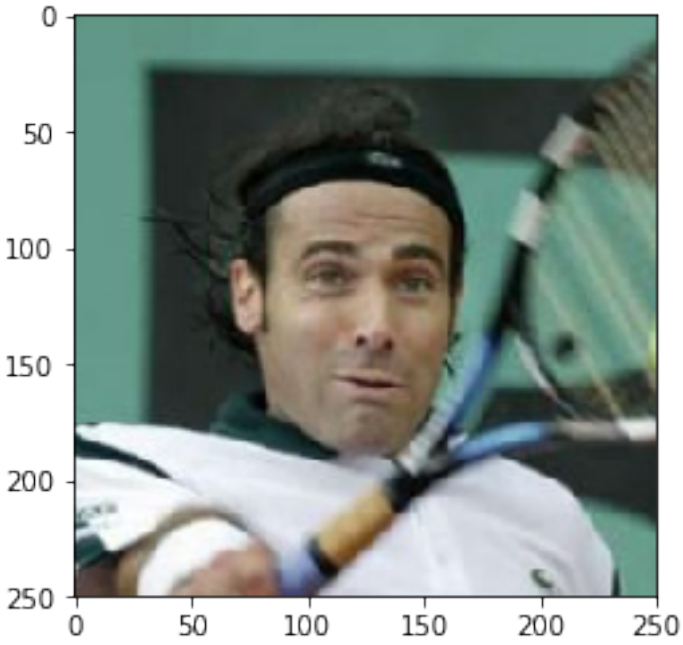
- 3. **Get more dog breeds data**. Although our dataset has data about 133 different dog breeds, real world is much more diverse with about 340 dog breeds. So in order for our model to generalize on new data, we could gather data from all the different dog breeds avaliable.

In [14]:
```
## 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
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/*/*/*"))

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

for file in np.hstack((human_files[:10], dog_files[:10])):
    run_app(file)
```

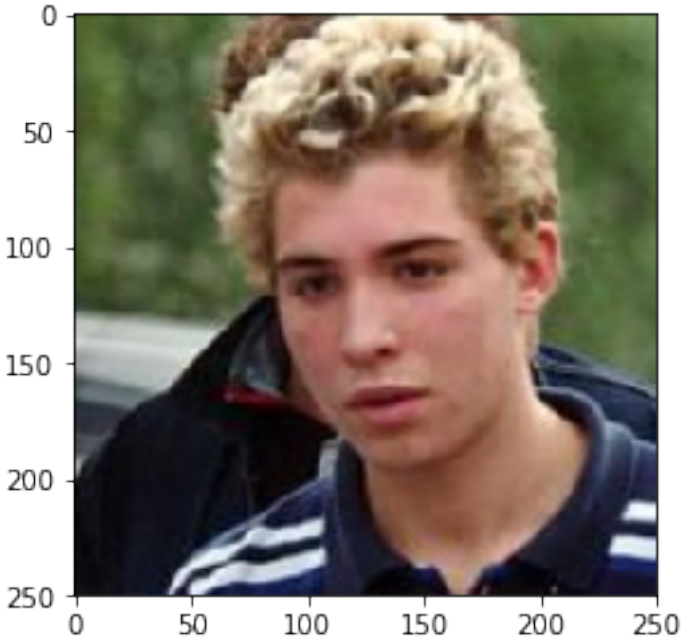
```
====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



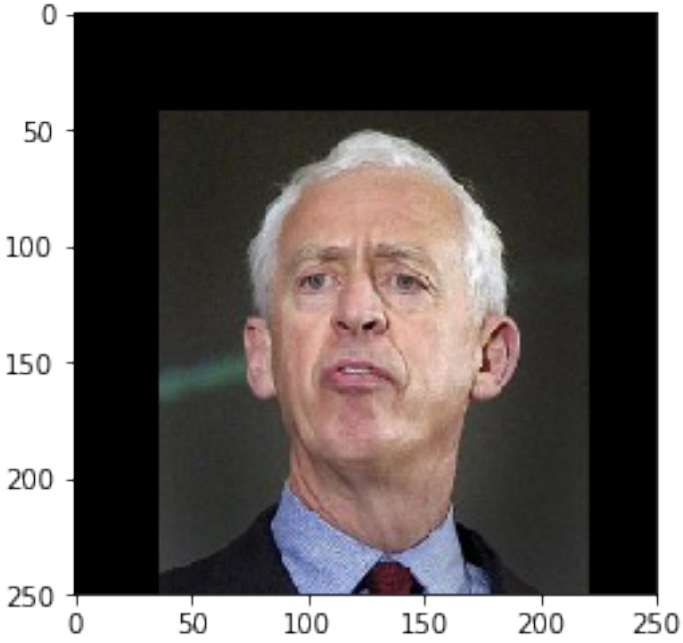
```
You look like a ... Silky terrier

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



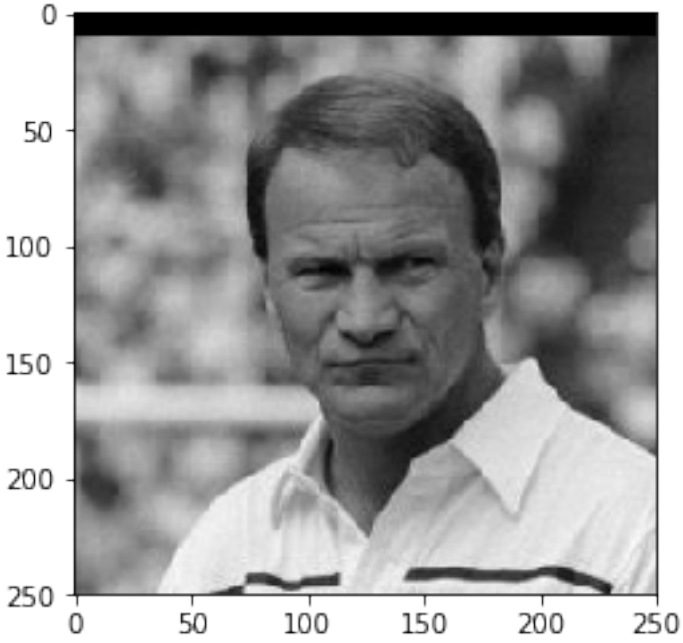

```
You look like a ... Bearded collie

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



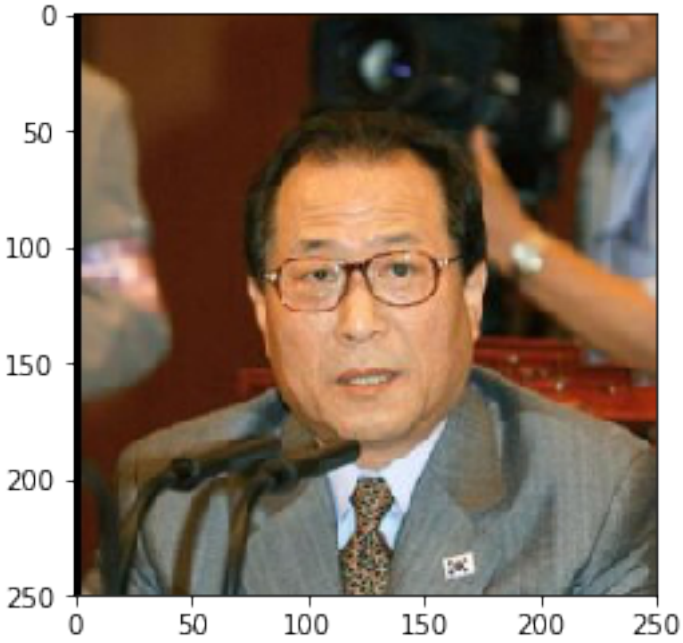

```
You look like a ... Afghan hound

======================== Dog Breed Classifier [v1] by SD ========================

Hello, Human
```



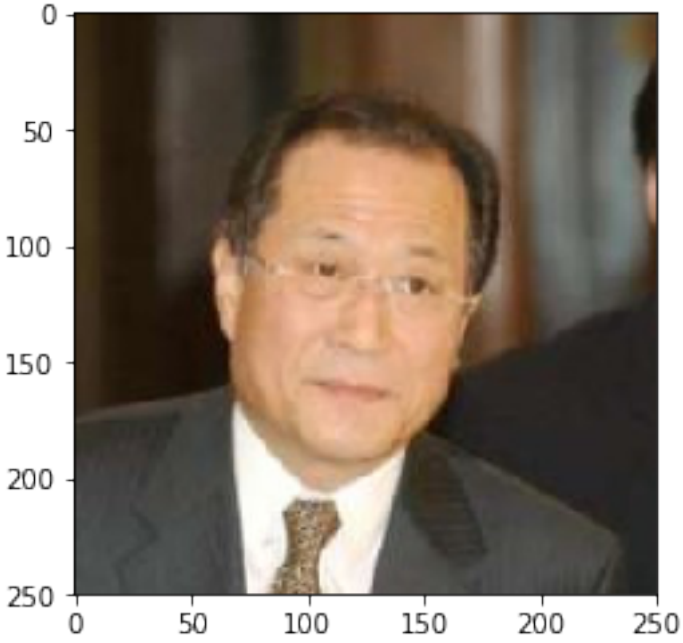
```
You look like a ... Pharaoh hound

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



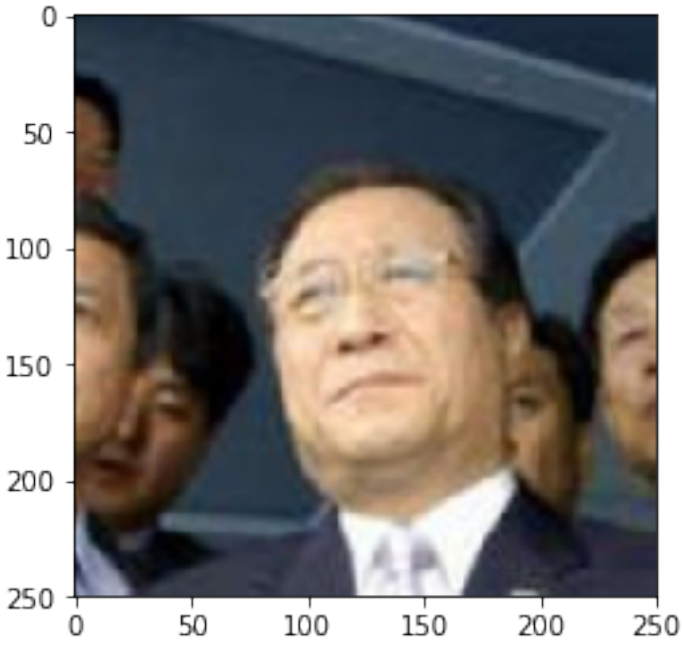

```
You look like a ... Bearded collie

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



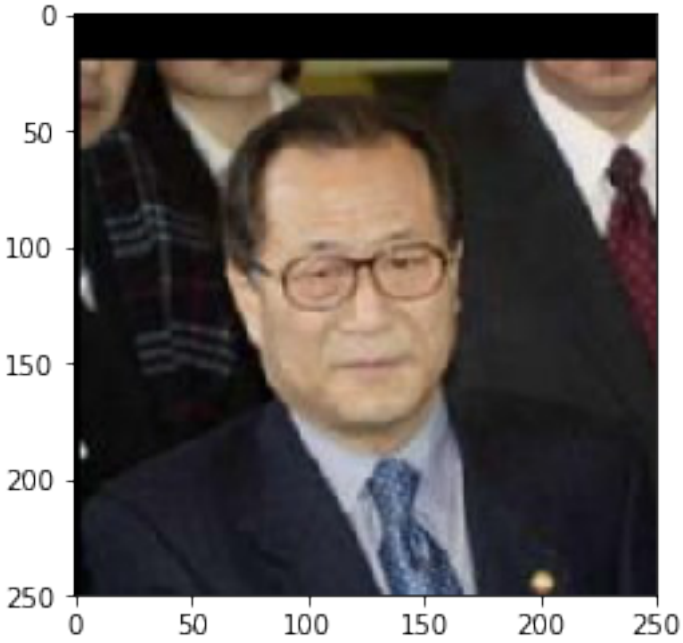

```
You look like a ... Smooth fox terrier

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



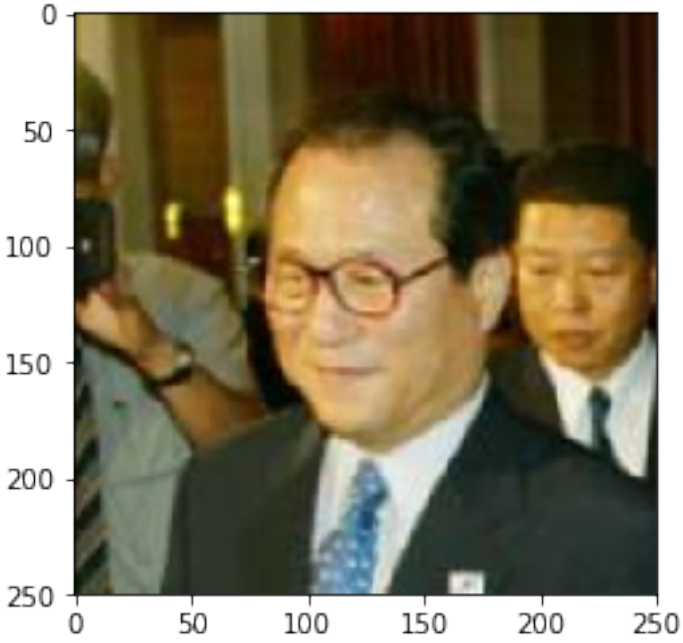
```
You look like a ... Basenji

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



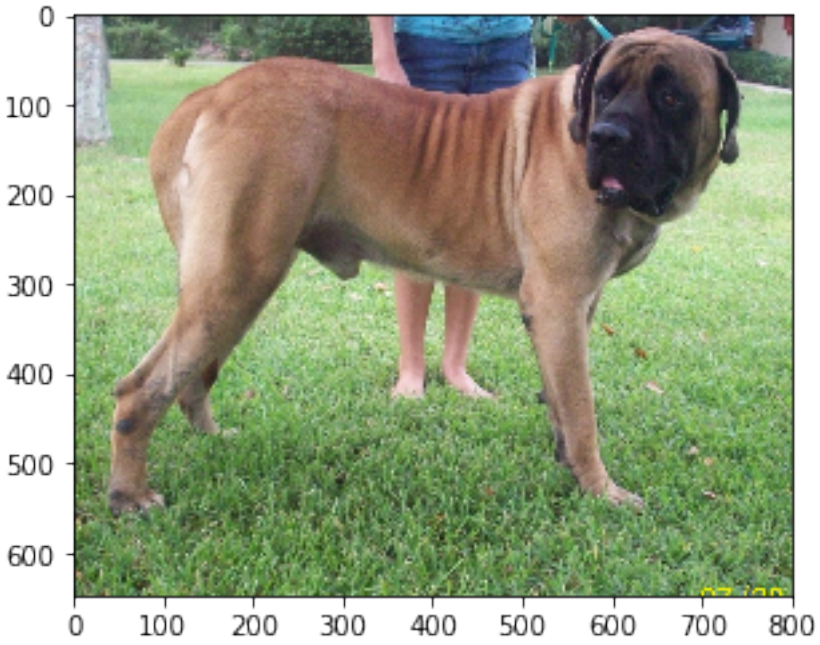

```
You look like a ... Silky terrier

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



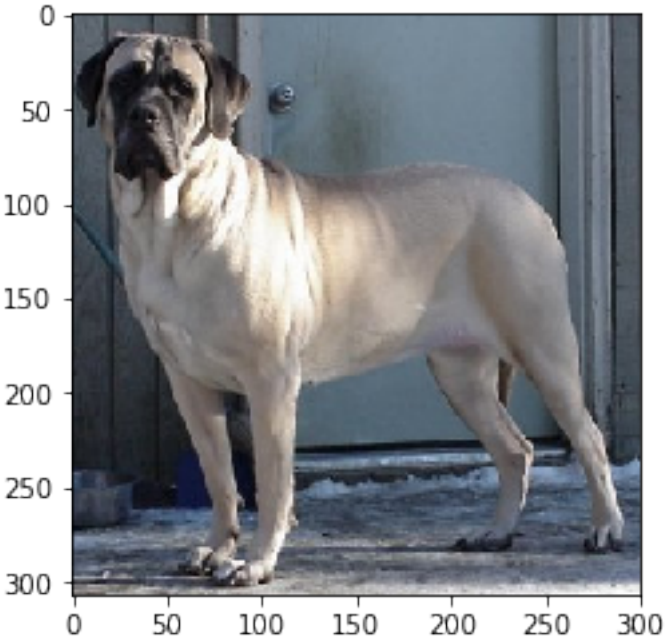

```
You look like a ... Doberman pinscher

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Human
```



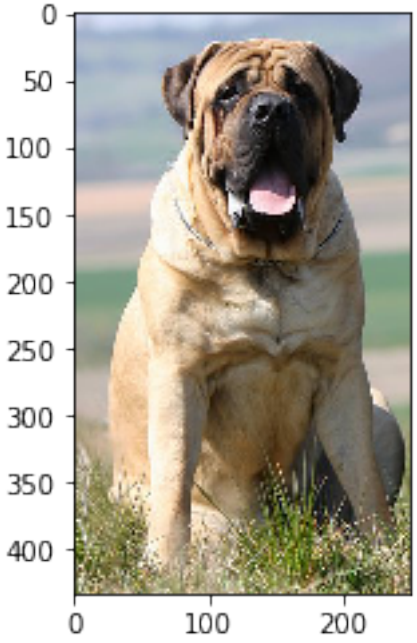
```
You look like a ... Dachshund

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```



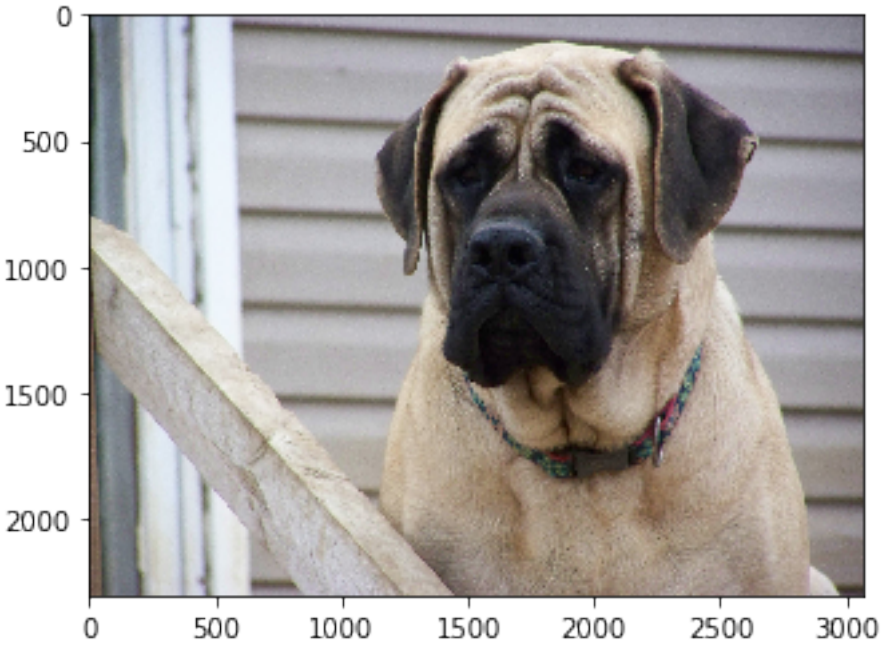

```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```



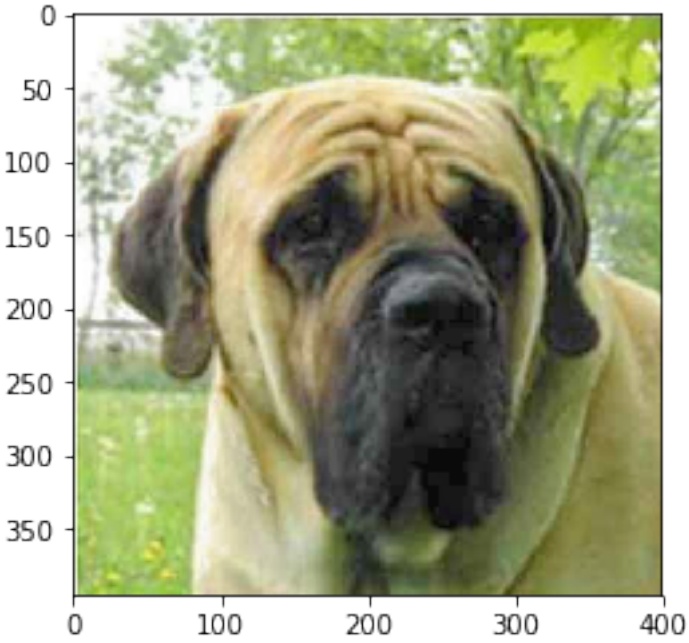

```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```



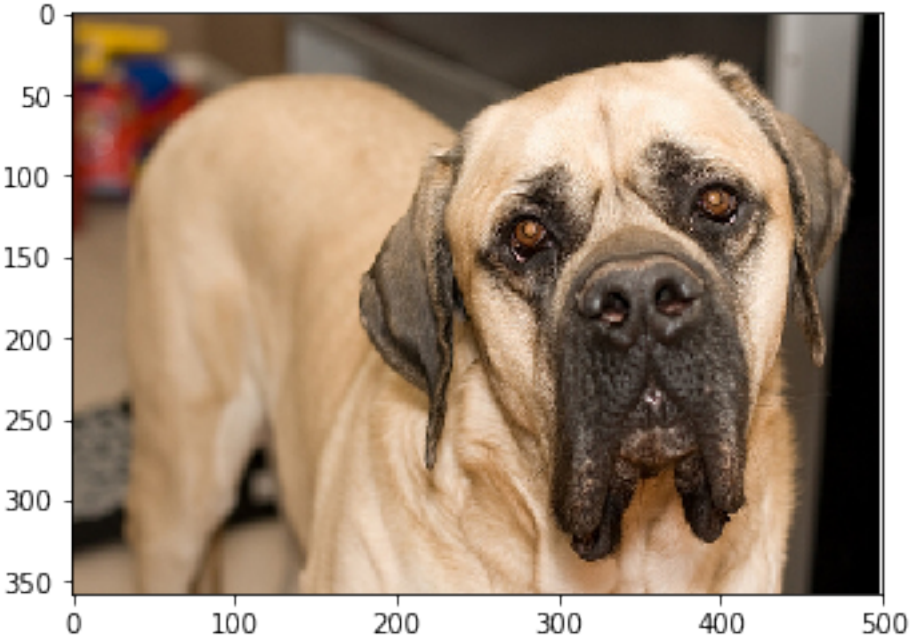
```
You look like a ... Cane corso

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```



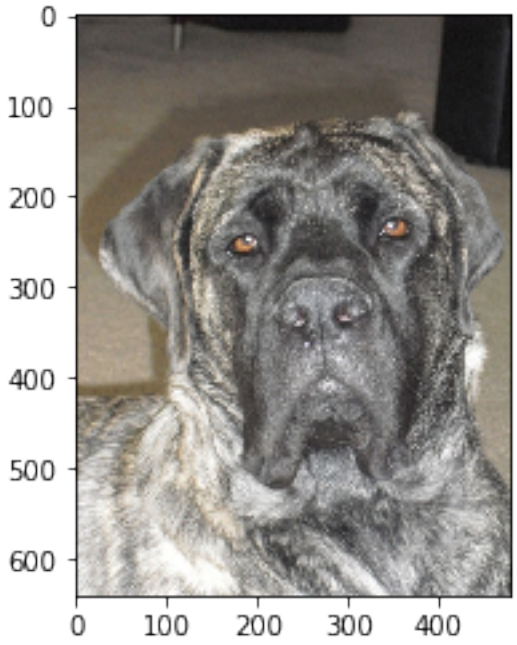

```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```




```
You look like a ... Bullmastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```

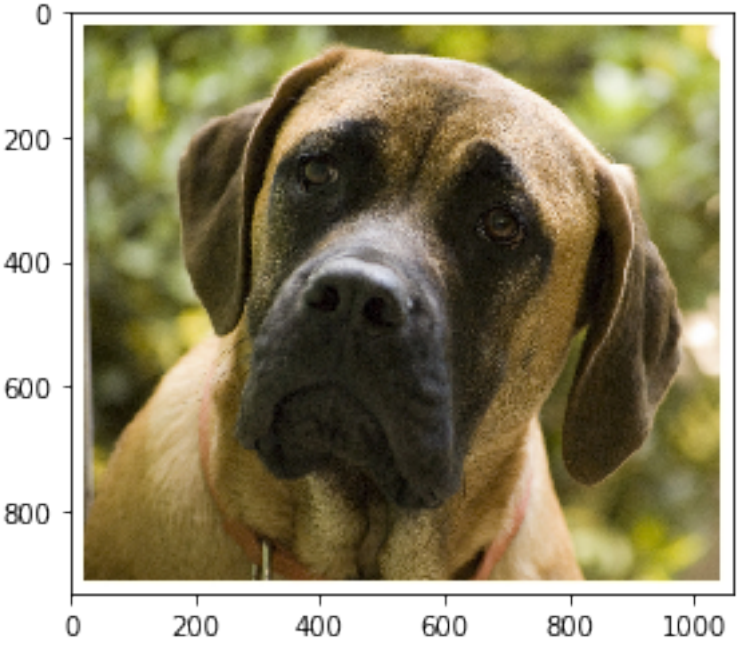


```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```

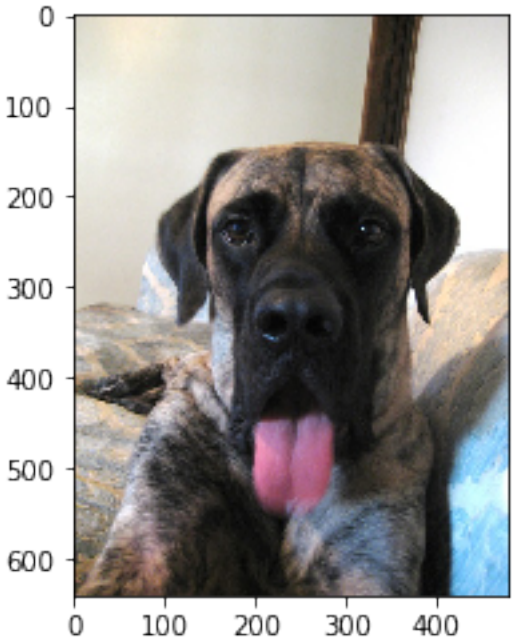



```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```


```
You look like a ... Mastiff

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```



```
You look like a ... Cane corso

====================== Dog Breed Classifier [v1] by SD ======================

Hello, Dog
```




```
You look like a ... Mastiff
```

```
In [12]: ############################################## THE END ###################################
```