

dog_app

August 26, 2019

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/**/*.jpg"))
        dog_files = np.array(glob("/data/dog_images/**/*.jpg"))

        # 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.

```
In [2]: human_files[0]
```

```
Out[2]: '/data/lfw/Dan_Ackroyd/Dan_Ackroyd_0001.jpg'
```

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))

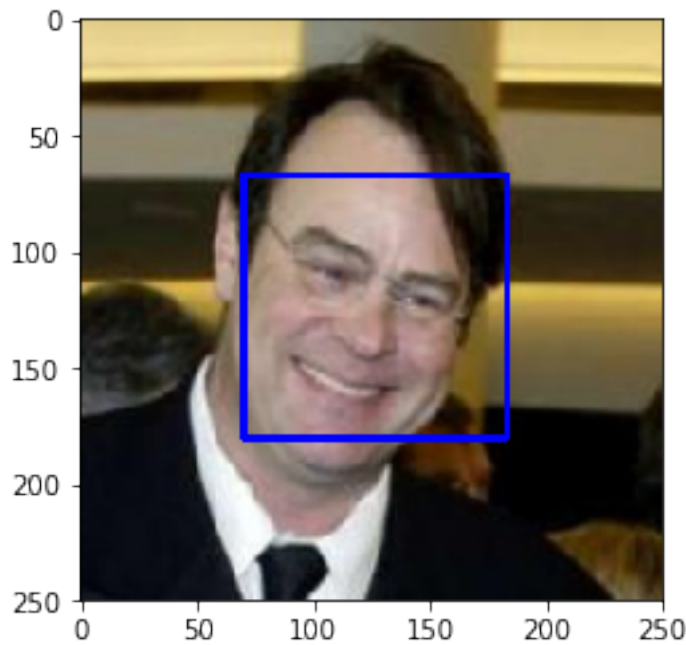
# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

# 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: Faces detected in 98.00% of the sample human dataset. Faces detected in 17.00% of the sample dog dataset.

```
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.

humans_count = 0
dogs_count = 0

for file in tqdm(human_files_short):
    if face_detector(file) == True:
        humans_count += 1

for file in tqdm(dog_files_short):
    if face_detector(file) == True:
        dogs_count += 1
```

```

print('%.1f%% images of the first 100 human_files were detected as human face.' % humans)
print('%.1f%% images of the first 100 dog_files were detected as human face.' % dogs_count)

100%|| 100/100 [00:02<00:00, 43.20it/s]
100%|| 100/100 [00:29<00:00, 3.34it/s]

98.0% images of the first 100 human_files were detected as human face.
17.0% images of the first 100 dog_files were detected as human face.

```

We suggest the face detector from OpenCV as a potential way to detect human images in your algorithm, but you are free to explore other approaches, especially approaches that make use of deep learning :). Please use the code cell below to design and test your own face detection algorithm. If you decide to pursue this *optional* task, report performance on `human_files_short` and `dog_files_short`.

```

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

```

Step 2: Detect Dogs

In this section, we use a [pre-trained model](#) to detect dogs in images.

1.1.3 Obtain Pre-trained VGG-16 Model

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

```

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

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

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

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

```

```

Downloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.torch/models/vgg16
100%|| 553433881/553433881 [00:12<00:00, 44541295.77it/s]

```

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

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

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

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

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

def transfrom(image_path):
    img =Image.open(image_path)

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

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

    img = Image.open(img_path)#.convert('RGB')
    img=transforms_in(img)

    # PyTorch pretrained models expect the Tensor dims to be (num input imgs, num color
    # Currently however, we have (num color channels, height, width); let's fix this by
```

```

# Insert the new axis at index 0 i.e. in front of the other axes/dims.
img = img.unsqueeze(0)

# Now that we have preprocessed our img, we need to convert it into a Variable
# PyTorch models expect inputs to be Variables. A PyTorch Variable is a wrapper around
img = Variable(img)
img = img.cuda()

# Returns a Tensor of shape (batch, num class labels)
prediction = VGG16(img)
prediction = prediction.cpu().data.numpy().argmax()

## Return the *index* of the predicted class for that image

return prediction # predicted class index

```

```

In [8]: from PIL import Image
import glob
image = Image.open(human_files[0])

# summarize some details about the image
print(image.format)
print(image.mode)
print(image.size)

# show the image
image.show()

```

```

JPEG
RGB
(250, 250)

```

```

In [9]: # load and display an image with Matplotlib
from matplotlib import image
from matplotlib import pyplot

# load image as pixel array
data = image.imread(human_files[0])

# summarize shape of the pixel array
print(data.dtype)
print(data.shape)

# display the array of pixels as an image
pyplot.imshow(data)
pyplot.show()

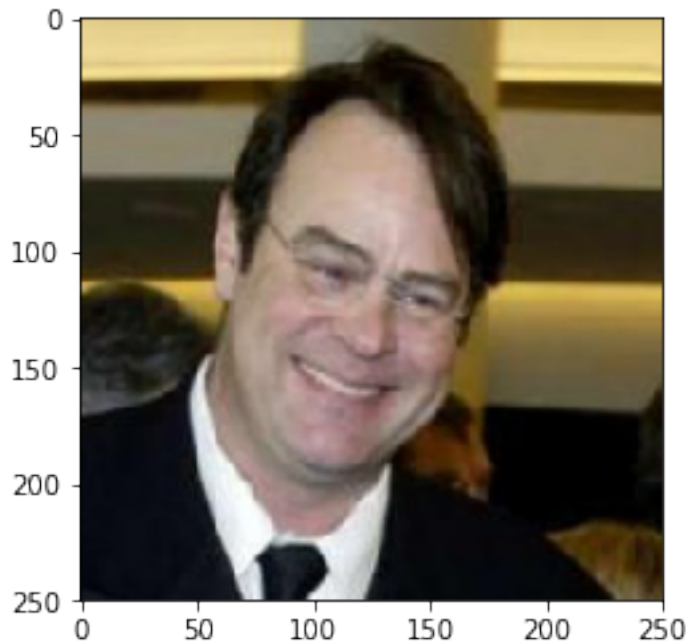
```

```

uint8

```

(250, 250, 3)



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

    ## TODO: Complete the function.

    pred=VGG16_predict(img_path)
    if pred >= 151 and pred <= 268:
        return True
    else:
        return False
```



```
In [10]: print(dog_detector(human_files[0]))
         print(dog_detector(dog_files[0]))
```

False

True

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: Detected dog in human files 2.0% Detected dog in dog files 100.0%

```
In [11]: ### TODO: Test the performance of the dog_detector function
         ### on the images in human_files_short and dog_files_short.
```

```
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.
```

```
#####Solution-1#####
```

```
## tqdm use for showing result in progress
```

```
humans_count = 0
```

```
dogs_count = 0
```

```
for file in tqdm(human_files_short):
```

```
    if dog_detector(file):
```

```
        humans_count += 1
```

```
for file in tqdm(dog_files_short):
```

```
    if dog_detector(file) == True:
```

```
        dogs_count += 1
```

```
#####Solution-2#####
```

```
##face_dec = np.vectorize(face_detector)
```

```
##human_face_detected =face_dec(human_files_short)
```

```
##dog_detected = face_dec(dog_files_short)
```

```
print('%0.1f%% images of the first 100 human_files were detected as dog face.' % humans_count)
```

```
print('%0.1f%% images of the first 100 dog_files were detected as dog face.' % dogs_count)
```

```
100%| 100/100 [00:03<00:00, 30.97it/s]
```

```
100%| 100/100 [00:04<00:00, 22.86it/s]
```

2.0% images of the first 100 human_files were detected as dog face.
100.0% images of the first 100 dog_files were detected as dog face.

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

```
In [ ]: ### (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 imbalanced, 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 [12]: import os
         from torchvision import datasets
         from torchvision import utils

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

         data_dir = "/data/dog_images/"
         num_workers = 0
         batch_size = 10
         data_transforms = {
             'train' : transforms.Compose([
                 transforms.Resize(256),
                 transforms.RandomResizedCrop(224),
                 transforms.RandomHorizontalFlip(), # augmented by randomly flip and rotate
                 transforms.RandomRotation(15),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
             ]),

             # no need of image augmentation for the validation test set
             'valid' : transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
             ]),

             # test dataset flips can be found out in fast.ai
             'test' : transforms.Compose([
                 transforms.Resize(256),
                 transforms.CenterCrop(224),
                 transforms.ToTensor(),
                 transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
             ]),
         }
```

```

train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

image_datasets = {
    'train' : datasets.ImageFolder(root=train_dir,transform=data_transforms['train']),
    'valid' : datasets.ImageFolder(root=valid_dir,transform=data_transforms['valid']),
    'test' : datasets.ImageFolder(root=test_dir,transform=data_transforms['test'])
}

# Loading Dataset
loaders_scratch = {
    'train' : torch.utils.data.DataLoader(image_datasets['train'],batch_size = batch_size),
    'valid' : torch.utils.data.DataLoader(image_datasets['valid'],batch_size = batch_size),
    'test' : torch.utils.data.DataLoader(image_datasets['test'],batch_size = batch_size)
}

```

```
In [13]: print(loaders_scratch['train'])
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7f941c9608d0>
```

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 have used transforms as ToTensor and RandomResizeCrop(224). The resize transform will resize the images with fixed size 224 and transform into tensors. Yes I decided to augment the dataset by using random horizontal flip and random rotation.

1.1.8 (IMPLEMENTATION) Model Architecture

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

```

In [14]: 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
        ## Follow the architecture of VGG-16
        ## Syntax: Attri 1) in_channels 2) out_channels (the no. of the output produced
        ## 3) kernel_size (k_h*k_w) 4) padding/stride

```

```

self.conv1 = nn.Conv2d(3, 16, 3)
self.conv2 = nn.Conv2d(16, 32, 3)
self.conv3 = nn.Conv2d(32, 64, 3)
self.conv4 = nn.Conv2d(64, 128, 3)
self.conv5 = nn.Conv2d(128, 256, 3)

self.fc1 = nn.Linear(256 * 6 * 6, 133)
self.max_pool = nn.MaxPool2d(2, 2, ceil_mode=True)

self.dropout = nn.Dropout(0.20)

self.conv_bn1 = nn.BatchNorm2d(224, 3)
self.conv_bn2 = nn.BatchNorm2d(16)
self.conv_bn3 = nn.BatchNorm2d(32)
self.conv_bn4 = nn.BatchNorm2d(64)
self.conv_bn5 = nn.BatchNorm2d(128)
self.conv_bn6 = nn.BatchNorm2d(256)

def forward(self, x):
    ## Define forward behavior
    x = F.relu(self.conv1(x))
    x = self.max_pool(x)
    x = self.conv_bn2(x)

    x = F.relu(self.conv2(x))
    x = self.max_pool(x)
    x = self.conv_bn3(x)

    x = F.relu(self.conv3(x))
    x = self.max_pool(x)
    x = self.conv_bn4(x)

    x = F.relu(self.conv4(x))
    x = self.max_pool(x)
    x = self.conv_bn5(x)

    x = F.relu(self.conv5(x))
    x = self.max_pool(x)
    x = self.conv_bn6(x)

    x = x.view(-1, 256 * 6 * 6)

    x = self.dropout(x)
    x = self.fc1(x)
    return x

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

```

```

# instantiate the CNN
model_scratch = Net()

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

```

In [17]: model_scratch

```

Out[17]: Net(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (conv5): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=9216, out_features=133, bias=True)
  (max_pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
  (dropout): Dropout(p=0.2)
  (conv_bn1): BatchNorm2d(224, eps=3, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn3): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv_bn6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
)

```

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

Answer: I used the 5 convolution layers all with the convolution of kernel size = 3, stride = 1 and padding = 0 Relu activations are used after each convolution layers except the last one. Max pooling layers of 2x2 are applied. Batch normalization is applied after each max pool layer. Dropout is applied with the probability of 0.2.

First layer will take three inputs for RGB because the in_channel is 3 and produces 16 output, the next layer to be a convolutional layer with 16 filters. Input = 224x224 RGB image Kernel Size = 3x3 Padding = 1 for 3x3 kernel MaxPooling = 2x2 with stride of 2 pixels, which will reduce the size of image and by the result the number of parameters will be half. Activation Function = ReLU (prevent from vanishing gradient). Batch Normalization 2D is a technique to provide inputs that are zero mean or variance 1 to prevent overfitting.

Layer 1: (3,16) input channels =3 , output channels = 16 Layer 2: (16,32) input channels = 16 , output channels = 32 Layer 3: (32,64) input channels =32 , output channels = 64 Layer 4: (64,128) input channels =64 , output channels = 128 Layer 5: (128,256) input channels =128 , output channels = 256

One fully connected layer with 9216 input channels and 133 output channel as dog breeds.

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 [15]: import torch.optim as optim

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

        ### TODO: select optimizer
        optimizer_scratch = optim.SGD(model_scratch.parameters(), lr=0.001, momentum=0.9)
```

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 [16]: ## Without following 2 line of code your this section is facing error "image file is tr
        ## This error seems to be a PIL error.
        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 of all optimized variables
                    optimizer.zero_grad()
                    ## forward pass
                    output = model(data)
                    ## calculate batch loss
                    loss = criterion(output, target)
                    ## backward pass
                    loss.backward()
```

```

        ## parameter update
        optimizer.step()
        ## update training loss
        train_loss += loss.item() * data.size(0)

#####
# 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

    ## forward pass
    output = model(data)
    ## batch loss
    loss = criterion(output, target)
    ## update validation loss
    valid_loss += loss.item() * data.size(0)
    ## calculate average losses
    train_loader = loaders['train']
    valid_loader = loaders['valid']
    train_loss = train_loss / len(train_loader.dataset)
    valid_loss = valid_loss / len(valid_loader.dataset)

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

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

if valid_loss <= valid_loss_min:
    print('Validation loss decreased ({:.6f} --> {:.6f}). Saving model...'.
          format(valid_loss_min, valid_loss))
    torch.save(model.state_dict(), save_path)
    valid_loss_min = valid_loss

# return trained model
return model

```



```

# train the model
model_scratch = train(25, loaders_scratch, model_scratch, optimizer_scratch,
                      criterion_scratch, use_cuda, 'model_scratch.pt')

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

```

```

Epoch: 1      Training Loss: 0.000000      Validation Loss: 0.030570
Validation loss decreased (inf --> 0.030570).    Saving model...
Epoch: 2      Training Loss: 0.000000      Validation Loss: 0.029544
Validation loss decreased (0.030570 --> 0.029544).    Saving model...
Epoch: 3      Training Loss: 0.000000      Validation Loss: 0.028224
Validation loss decreased (0.029544 --> 0.028224).    Saving model...
Epoch: 4      Training Loss: 0.000000      Validation Loss: 0.025340
Validation loss decreased (0.028224 --> 0.025340).    Saving model...
Epoch: 5      Training Loss: 0.000000      Validation Loss: 0.026206
Epoch: 6      Training Loss: 0.000000      Validation Loss: 0.025948
Epoch: 7      Training Loss: 0.000000      Validation Loss: 0.025712
Epoch: 8      Training Loss: 0.000000      Validation Loss: 0.024014
Validation loss decreased (0.025340 --> 0.024014).    Saving model...
Epoch: 9      Training Loss: 0.000000      Validation Loss: 0.023901
Validation loss decreased (0.024014 --> 0.023901).    Saving model...
Epoch: 10     Training Loss: 0.000000      Validation Loss: 0.027760
Epoch: 11     Training Loss: 0.000000      Validation Loss: 0.026494
Epoch: 12     Training Loss: 0.000000      Validation Loss: 0.016700
Validation loss decreased (0.023901 --> 0.016700).    Saving model...
Epoch: 13     Training Loss: 0.000000      Validation Loss: 0.019778
Epoch: 14     Training Loss: 0.000000      Validation Loss: 0.018475
Epoch: 15     Training Loss: 0.000000      Validation Loss: 0.018986
Epoch: 16     Training Loss: 0.000000      Validation Loss: 0.017581
Epoch: 17     Training Loss: 0.000000      Validation Loss: 0.018158
Epoch: 18     Training Loss: 0.000000      Validation Loss: 0.013152
Validation loss decreased (0.016700 --> 0.013152).    Saving model...
Epoch: 19     Training Loss: 0.000000      Validation Loss: 0.014911
Epoch: 20     Training Loss: 0.000000      Validation Loss: 0.013610
Epoch: 21     Training Loss: 0.000000      Validation Loss: 0.015076
Epoch: 22     Training Loss: 0.000000      Validation Loss: 0.009138
Validation loss decreased (0.013152 --> 0.009138).    Saving model...
Epoch: 23     Training Loss: 0.000000      Validation Loss: 0.009323
Epoch: 24     Training Loss: 0.000000      Validation Loss: 0.011187
Epoch: 25     Training Loss: 0.000000      Validation Loss: 0.012439

```

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

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

    model.eval()
    for batch_idx, (data, target) in enumerate(loaders['test']):
        # move to GPU
        if use_cuda:
            data, target = data.cuda(), target.cuda()

        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)

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

        # update average test loss
        test_loss = test_loss + ((1 / (batch_idx + 1)) * (loss.data - test_loss))

        # convert output probabilities to predicted class
        pred = output.data.max(1, keepdim=True)[1]

        # compare predictions to true label
        correct += np.sum(np.squeeze(pred.eq(target.data.view_as(pred))).cpu().numpy())
        total += data.size(0)

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

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

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

```

Test Loss: 2.928003

Test Accuracy: 29% (245/836)

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 [18]: ## TODO: Specify data loaders
```

```
import os
from torchvision import datasets
from torchvision import utils

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

data_dir = "/data/dog_images/"
num_workers = 0
batch_size = 10
data_transforms = {
    'train' : transforms.Compose([
        transforms.Resize(256),
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(), # randomly flip and rotate
        transforms.RandomRotation(15),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),

    # no need of image augmentation for the validation test set
    'valid' : transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),

    # test dataset flips can be found out in fast.ai
    'test' : transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
    ]),
}
```

```

train_dir = data_dir + '/train'
valid_dir = data_dir + '/valid'
test_dir = data_dir + '/test'

image_datasets = {
    'train' : datasets.ImageFolder(root=train_dir,transform=data_transforms['train']),
    'valid' : datasets.ImageFolder(root=valid_dir,transform=data_transforms['valid']),
    'test' : datasets.ImageFolder(root=test_dir,transform=data_transforms['test'])
}

# Loading Dataset
loaders_transfer = {
    'train' : torch.utils.data.DataLoader(image_datasets['train'],batch_size = batch_size),
    'valid' : torch.utils.data.DataLoader(image_datasets['valid'],batch_size = batch_size),
    'test' : torch.utils.data.DataLoader(image_datasets['test'],batch_size = batch_size)
}

```

```
In [19]: print(loaders_transfer['train'])
```

```
<torch.utils.data.dataloader.DataLoader object at 0x7f941c062ac8>
```

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

## TODO: Specify model architecture

## Load VGG-16 model
model_transfer = models.vgg16(pretrained=True)

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

## Get the input of the last layer of VGG-16
n_inputs = model_transfer.classifier[6].in_features

## Create a new layer(n_inputs -> 133). #The new layer's requires_grad will be automatic
last_layer = nn.Linear(n_inputs, 133)

## Change the last layer to the new layer.
model_transfer.classifier[6] = last_layer

## Print the model.

```

```

    print(model_transfer)

    if use_cuda:
        model_transfer = model_transfer.cuda()

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)

```

```
)  
)
```

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 used VGG-16 model which is already architected and trained to extract the feature. I updated the final fully connected layer to take inputs and produce 133 breed as a output.

As we know, we can use pretrained image classifier model for our own image classification problem. The VGG-16 is trained and precisely tuned layers so that we don't need to worry about feature extraction layers or simply convolutional layers. This model is suitable because my dog breed problem has RGB images with different sizes. As we can see the VGG-16 has 3 channels as input which is for RGB images (3 channels) and produce output of 64, the architecture makes sense to start with dog breed classifier.

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

1.1.15 (IMPLEMENTATION) Train and Validate the Model

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

```
In [22]: # train the model  
         n_epochs = 5  
         model_transfer = train(n_epochs, loaders_transfer, model_transfer, optimizer_transfer,  
                                criterion_transfer, use_cuda, 'model_transfer.pt')  
  
         # load the model that got the best validation accuracy (uncomment the line below)  
         #model_transfer.load_state_dict(torch.load('model_transfer.pt'))
```

```
Epoch: 1      Training Loss: 0.000000      Validation Loss: 0.016176  
Validation loss decreased (inf --> 0.016176).    Saving model...  
Epoch: 2      Training Loss: 0.000000      Validation Loss: 0.007787  
Validation loss decreased (0.016176 --> 0.007787).    Saving model...  
Epoch: 3      Training Loss: 0.000000      Validation Loss: 0.001960  
Validation loss decreased (0.007787 --> 0.001960).    Saving model...  
Epoch: 4      Training Loss: 0.000000      Validation Loss: 0.001841  
Validation loss decreased (0.001960 --> 0.001841).    Saving model...  
Epoch: 5      Training Loss: 0.000000      Validation Loss: 0.000712  
Validation loss decreased (0.001841 --> 0.000712).    Saving model...
```

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

Test Loss: 0.548933

Test Accuracy: 83% (698/836)

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 [25]: ### 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]

         # Load the trained model 'model_transfer.pt'
         model_transfer.load_state_dict(torch.load('model_transfer.pt', map_location='cpu'))

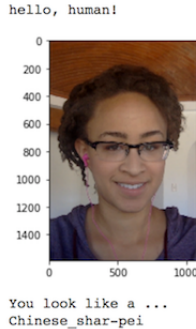
         def predict_breed_transfer(img_path):
             # load the image and return the predicted breed
             image = Image.open(img_path).convert('RGB')
             prediction_transform = transforms.Compose([transforms.Resize(size=(224, 224)),
                                                         transforms.ToTensor(),
                                                         transforms.Normalize(mean=[0.485, 0.456, 0.406], st

             # discard the transparent, alpha channel (that's the :3) and add the batch dimension
             image = prediction_transform(image)[:3,:,:].unsqueeze(0)
             image = image.cuda()

             model_transfer.eval()
             idx = torch.argmax(model_transfer(image))
             return class_names[idx]
```

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!

1.1.18 (IMPLEMENTATION) Write your Algorithm

```
In [26]: ### TODO: Write your algorithm.
        ### Feel free to use as many code cells as needed.

from PIL import Image
def run_app(img_path):
    ## handle cases for a human face, dog, and neither
    img = Image.open(img_path)
    plt.imshow(img)
    plt.show()

    if dog_detector(img_path) is True:
        prediction = predict_breed_transfer(img_path)
        print("A dog has been detected which most likely to be {0} breed".format(prediction))
    elif face_detector(img_path) > 0:
        prediction = predict_breed_transfer(img_path)
        print("This is a Human who looks like a {0}".format(prediction))
    else:
        print("Neither Human nor Dog")
```

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.

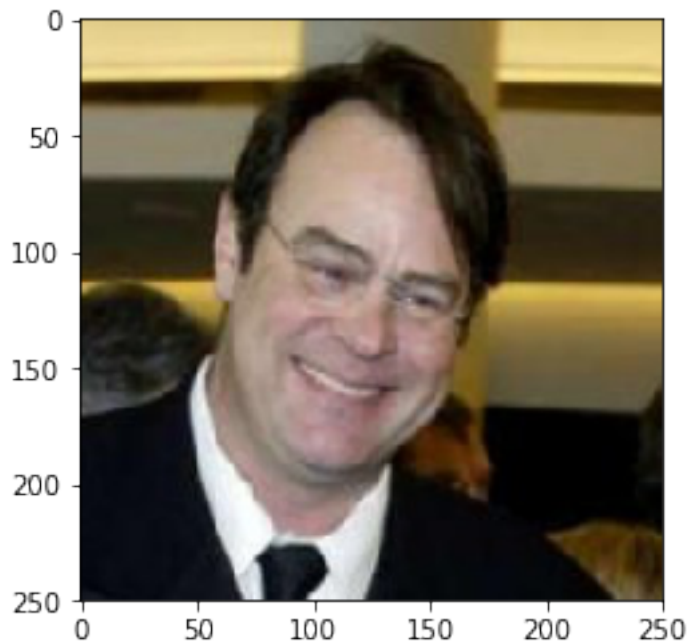
Answer: Yes, output is better than I expected.

Following are the key points to improve this algorithm:

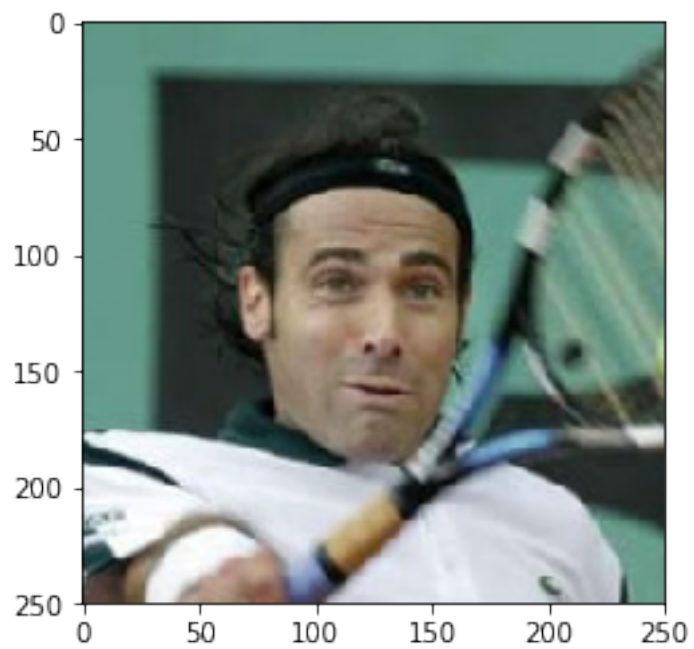
1.Large datasets will improve the performance of the model 2.Hyper-parameter tuning and training will improve 3.ensembles modeling

```
In [27]: ## 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.

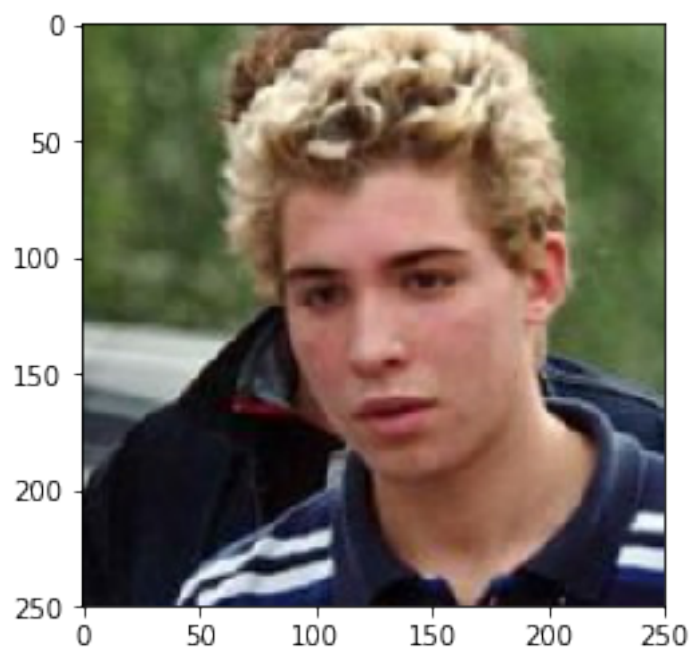
         ## suggested code, below
         for file in np.hstack((human_files[:3], dog_files[:3])):
             run_app(file)
```



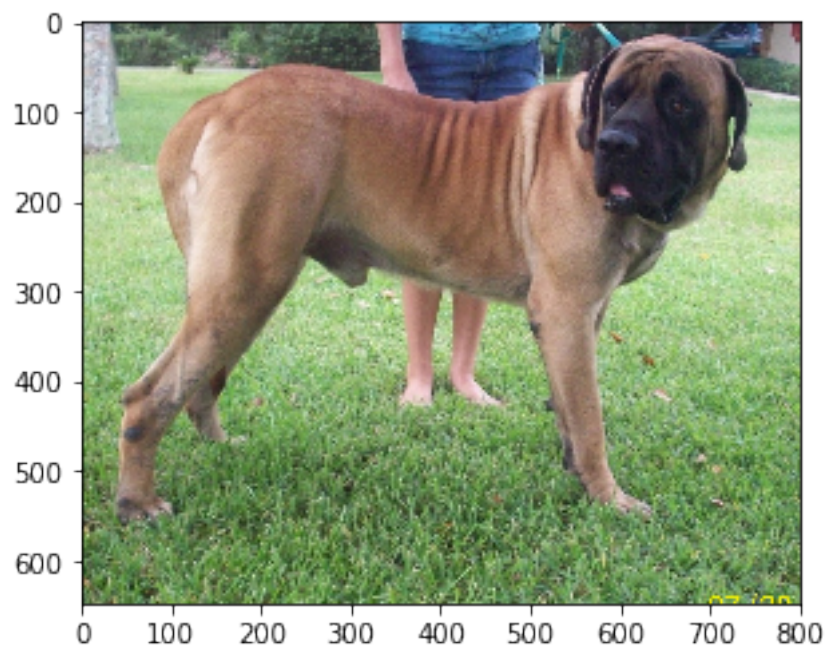
This is a Human who looks like a Beagle



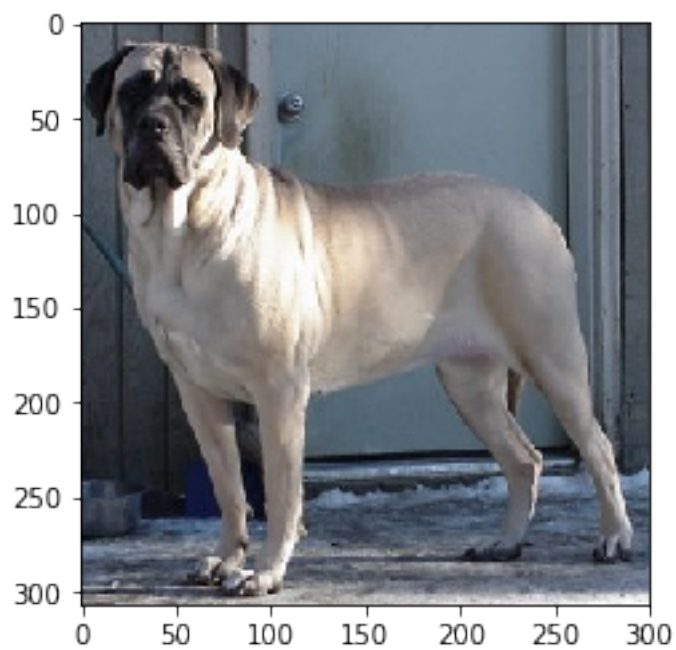
This is a Human who looks like a Komondor



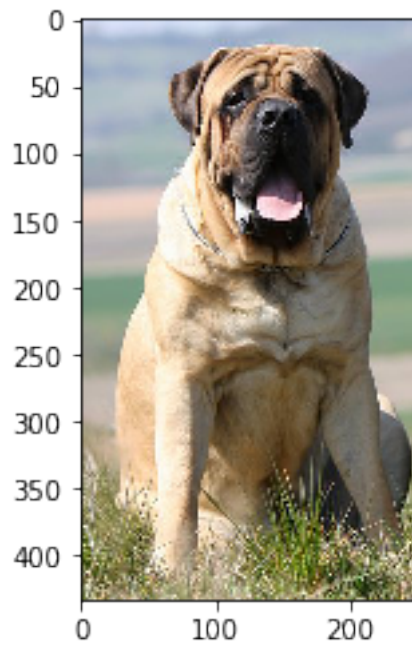
This is a Human who looks like a Irish water spaniel



A dog has been detected which most likely to be Bullmastiff breed



A dog has been detected which most likely to be Bullmastiff breed



A dog has been detected which most likely to be Bullmastiff breed

In []: