Convolutional Neural Networks

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.

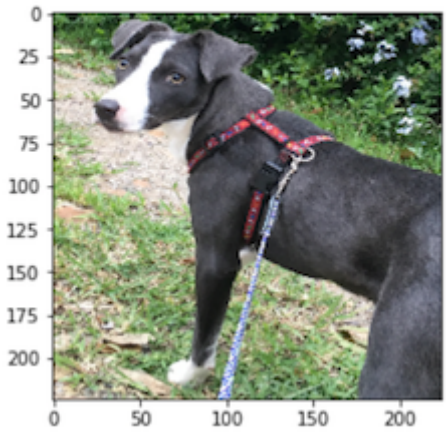
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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.

Why We're Here

In this notebook, you will make the first steps towards developing an algorithm that could be used as part of a mobile or web app. At the end of this project, your code will accept any user-supplied image as input. If a dog is detected in the image, it will provide an estimate of the dog's breed. If a human is detected, it will provide an estimate of the dog breed that is most resembling. The image below displays potential sample output of your finished project (... but we expect that each student's algorithm will behave differently!).

```
hello, dog!
your predicted breed is ...
American Staffordshire terrier
```



In this real-world setting, you will need to piece together a series of models to perform different tasks; for instance, the algorithm that detects humans in an image will be different from the CNN that infers dog breed. There are many points of possible failure, and no perfect algorithm exists. Your imperfect solution will nonetheless create a fun user experience!

The Road Ahead

We break the notebook into separate steps. Feel free to use the links below to navigate the notebook.

Step 0: Import Datasets

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

- Download the dog dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip). Unzip the folder and place it in this project's home directory, at the location `/dogImages`.
- Download the human dataset (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip). 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 (http://www.7-zip.org/) 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]:

```
!wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/dogImages.zip
!wget https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/lfw.zip
!wget https://r-aravind-somedata.s3.amazonaws.com/capstone.zip

!unzip dogImages.zip
!unzip lfw.zip
!unzip capstone.zip
```

```
Streaming output truncated to the last 5000 lines.
   creating: __MACOSX/lfw/Stella_McCartney/
  inflating: __MACOSX/lfw/Stella_McCartney/._Stella_McCartney_0001.j
pg
  inflating: __MACOSX/lfw/._Stella_McCartney
   creating: lfw/Stella_Tennant/
  inflating: lfw/Stella_Tennant/Stella_Tennant_0001.jpg
   creating: __MACOSX/lfw/Stella_Tennant/
  inflating: __MACOSX/lfw/Stella_Tennant/._Stella_Tennant_0001.jpg
  inflating: __MACOSX/lfw/._Stella_Tennant
   creating: lfw/Stellan_Skarsgard/
  inflating: lfw/Stellan_Skarsgard/Stellan_Skarsgard_0001.jpg
   creating: __MACOSX/lfw/Stellan_Skarsgard/
  inflating: __MACOSX/lfw/Stellan_Skarsgard/._Stellan_Skarsgard_000
1.jpg
  inflating: lfw/Stellan_Skarsgard/Stellan_Skarsgard_0002.jpg
  inflating: __MACOSX/lfw/Stellan_Skarsgard/._Stellan_Skarsgard_000
2.jpg
  inflating: __MACOSX/lfw/._Stellan_Skarsgard
   creating: lfw/Steny_Hoyer/
```

In [2]:

```
!ls
```

```
capstone.zip  dogImages.zip  images  lfw.zip   sample_data
dogImages     haarcascades   lfw     __MACOSX
```

In [3]:

```
import numpy as np
from glob import glob

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

# 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

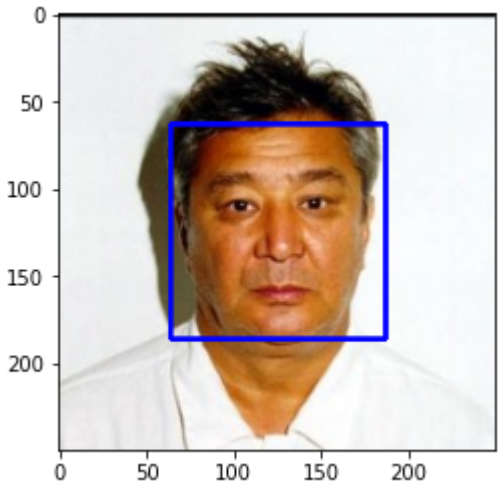
In this section, we use OpenCV's implementation of [Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html)](http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images.

OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). 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 [4]:

```
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[1])
# 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.

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 [0]:

```
# 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
```

(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 [0]:

```
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 = 0
human_like_dogs = 0

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

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

In [7]:

```
print("Percentage of detected human faces in first 100 images in human_files: {}%".
print("Percentage of detected human faces in first 100 images in dog_files: {}%".fo
```

```
Percentage of detected human faces in first 100 images in human_files:
100%
Percentage of detected human faces in first 100 images in dog_files: 1
0%
```

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 [0]:

```
### (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 (http://pytorch.org/docs/master/torchvision/models.html) to detect dogs in images.

Obtain Pre-trained VGG-16 Model

The code cell below downloads the VGG-16 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), 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 (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [9]:

```
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/.cache/torch/checkpoints/vgg16-397923af.pth

HBox(children=(FloatProgress(value=0.0, max=553433881.0), HTML(value
='')))
```

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.

(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 (http://pytorch.org/docs/stable/torchvision/models.html).

In [0]:

```
from PIL import Image
import torchvision.transforms as transforms

# Set PIL to be tolerant of image files that are truncated.
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

    img = Image.open(img_path)

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

    img = pre_processing(img).unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    with torch.no_grad():
        VGG16.eval()
        output = VGG16(img)
        prediction = torch.argmax(output).item()

    return prediction # predicted class index
```

(IMPLEMENTATION) Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), 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 [0]:

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

    prediction = VGG16_predict(img_path)

    return (151 <= prediction) and (prediction <= 268)
```

(IMPLEMENTATION) Assess the Dog Detector

Question 2: Use the code cell below to test the performance of your `dog_detector` function.

- What percentage of the images in `human_files_short` have a detected dog?
- What percentage of the images in `dog_files_short` have a detected dog?

Answer:

In [0]:

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

dog_like_humans = 0
dogs = 0

for file in human_files_short:
    if dog_detector(file) == True:
        dog_like_humans += 1

for file in dog_files_short:
    if dog_detector(file) == True:
        dogs +=1
```

In [13]:

```
print("Percentage of detected dog faces in first 100 images in human_files: {}%".fo
print("Percentage of detected dog faces in first 100 images in dog_files: {}%".form
```

```
Percentage of detected dog faces in first 100 images in human_files:
0%
Percentage of detected dog faces in first 100 images in dog_files: 10
0%
```

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 (http://pytorch.org/docs/master/torchvision/models.html#inception-v3), ResNet-50 (http://pytorch.org/docs/master/torchvision/models.html#id3), 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 [0]:

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

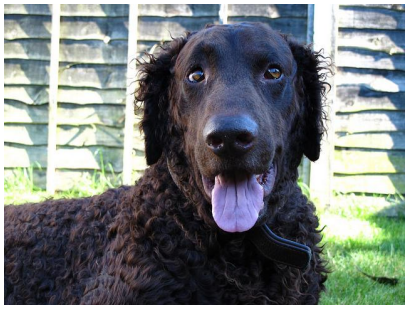
Step 3: Create a CNN to Classify Dog Breeds (from Scratch)

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
	

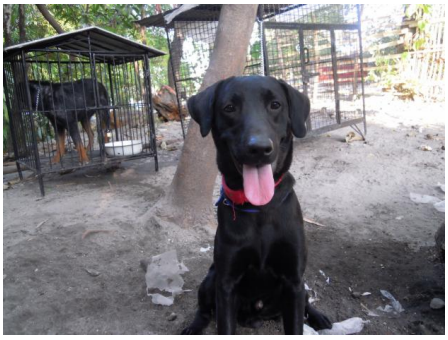
Likewise, recall that labradors come in yellow, chocolate, and black. Your vision-based algorithm will have to conquer this high intra-class variation to determine how to classify all of these different shades as the same breed.

Yellow Labrador | Chocolate Labrador | Black 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!

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) for the training, validation, and test datasets of dog images (located at `dogImages/train`, `dogImages/valid`, and `dogImages/test`, respectively). You may find [this documentation on custom datasets (http://pytorch.org/docs/stable/torchvision/datasets.html)](http://pytorch.org/docs/stable/torchvision/datasets.html) to be a useful resource. If you are interested in augmenting your training and/or validation data, check out the wide variety of [transforms (http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)](http://pytorch.org/docs/stable/torchvision/transforms.html?highlight=transform)!

In [0]:

```
import os
from torchvision import datasets

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

data_dir = 'dogImages/'

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.Resize(255),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomResizedCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])
                                      ])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5],
                                                           [0.5, 0.5, 0.5])
                                     ])

batch_size = 128

train_data = datasets.ImageFolder(os.path.join(data_dir,'train'), transform=train_t
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffl

valid_data = datasets.ImageFolder(os.path.join(data_dir,'valid'), transform=valid_t
validloader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffl

test_data = datasets.ImageFolder(os.path.join(data_dir,'test'), transform=valid_tra
testloader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=False)
```

Question 3: Describe your chosen procedure for preprocessing the data.

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

- **Answer**:

1. I resized the image to 255x255px. Then I used random resized crop to crop it to 224x224px. Hence, input tensor shape is 224x224. I picked this because it is a common choice in many pre-trained models including the VGG16.
2. Augementation was done by random rotation of upto 30 degrees and a random horizontal flip. This will help to generalize the training data.

(IMPLEMENTATION) Model Architecture

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

In [0]:

```
import torch.nn as nn
import torch.nn.functional as F

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

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, padding=1)
        self.conv4 = nn.Conv2d(256, 512, kernel_size=3, padding=1)
        self.conv5 = nn.Conv2d(512, 512, kernel_size=3, padding=1)

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

        self.fc1 = nn.Linear(25088, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 133)
        self.dropout = nn.Dropout(0.5)


    def forward(self, x):
        ## Define forward behavior

        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = self.pool(F.relu(self.conv5(x)))

        x = x.view(-1, 25088)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x= self.fc3(x)

        return x

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

# instantiate the CNN
model_scratch = Net()

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

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

Answer:

- I decided to go with 5 convolutional layers with a kernel size of 3x3 and padding of 1.
- Between every convolutional layer I added a maxpool layer with kernel size 2x2 and stride of 2.
- Hence, after 5 convolutions we will end up with an output of 512x7x7 (512 feature maps of 7x7)
- Then it is flattened into a vector of length 25088 and fed to an fully connected layer.
- I used 3 fully connected layers with the output layer of 133 nodes (one for each of the 133 classes) and I used a relu activation function.
- I used dropout to prevent overfitting and to achieve generalization.

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/stable/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/stable/optim.html). Save the chosen loss function as `criterion_scratch`, and the optimizer as `optimizer_scratch` below.

In [0]:

```
import torch.optim as optim

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

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

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_scratch.pt'`.

In [34]:

```
use_cuda = torch.cuda.is_available()
if use_cuda:
    print("CUDA is available")
else:
    print("CUDA is not available")
```

```
CUDA is available
```

In [0]:

```
# To avoid memory error if it occurs
torch.cuda.empty_cache()
```

In [35]:

```
# the following import is required for training to be robust to truncated images
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 - trai

            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)

            loss.backward()
            optimizer.step()

            train_loss = train_loss + ((1 / (batch_idx + 1)) * (loss.data - train_l

        ######################
        # 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
            with torch.no_grad():
                output = model(data)

            loss = criterion(output, target)
            valid_loss = valid_loss + ((1 / (batch_idx + 1)) * (loss.data - valid_l


        # print training/validation statistics
        print(" ")
        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 from {:.6f} to {:.6f} | SAVING MODEL."
            valid_loss_min = valid_loss
            torch.save(model.state_dict(), save_path)

    # return trained model
    return model

loaders_scratch = {'train': trainloader,
                   'valid': validloader,
                   'test': testloader}

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

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

```
Epoch: 1        Training Loss: 4.884384         Validation Loss: 4.867
551
validation loss decreased from inf to 4.867551 | SAVING MODEL.

Epoch: 2        Training Loss: 4.838078         Validation Loss: 4.785
732
validation loss decreased from 4.867551 to 4.785732 | SAVING MODEL.

Epoch: 3        Training Loss: 4.736030         Validation Loss: 4.667
461
validation loss decreased from 4.785732 to 4.667461 | SAVING MODEL.

Epoch: 4        Training Loss: 4.684020         Validation Loss: 4.565
259
validation loss decreased from 4.667461 to 4.565259 | SAVING MODEL.

Epoch: 5        Training Loss: 4.598903         Validation Loss: 4.423
836
validation loss decreased from 4.565259 to 4.423836 | SAVING MODEL.

Epoch: 6        Training Loss: 4.540593         Validation Loss: 4.348
523
validation loss decreased from 4.423836 to 4.348523 | SAVING MODEL.

Epoch: 7        Training Loss: 4.467464         Validation Loss: 4.269
838
validation loss decreased from 4.348523 to 4.269838 | SAVING MODEL.

Epoch: 8        Training Loss: 4.423851         Validation Loss: 4.277
485

Epoch: 9        Training Loss: 4.375755         Validation Loss: 4.149
446
validation loss decreased from 4.269838 to 4.149446 | SAVING MODEL.

Epoch: 10       Training Loss: 4.311601         Validation Loss: 4.067
079
validation loss decreased from 4.149446 to 4.067079 | SAVING MODEL.
```

```
Epoch: 11       Training Loss: 4.254384         Validation Loss: 3.996
764
validation loss decreased from 4.067079 to 3.996764 | SAVING MODEL.

Epoch: 12       Training Loss: 4.261071         Validation Loss: 4.023
086

Epoch: 13       Training Loss: 4.208914         Validation Loss: 3.958
763
validation loss decreased from 3.996764 to 3.958763 | SAVING MODEL.

Epoch: 14       Training Loss: 4.138472         Validation Loss: 3.895
195
validation loss decreased from 3.958763 to 3.895195 | SAVING MODEL.

Epoch: 15       Training Loss: 4.137808         Validation Loss: 4.006
435

Epoch: 16       Training Loss: 4.107223         Validation Loss: 3.914
882

Epoch: 17       Training Loss: 4.051947         Validation Loss: 3.879
456
validation loss decreased from 3.895195 to 3.879456 | SAVING MODEL.

Epoch: 18       Training Loss: 4.044856         Validation Loss: 3.805
078
validation loss decreased from 3.879456 to 3.805078 | SAVING MODEL.

Epoch: 19       Training Loss: 4.009995         Validation Loss: 3.819
511

Epoch: 20       Training Loss: 4.000538         Validation Loss: 3.775
684
validation loss decreased from 3.805078 to 3.775684 | SAVING MODEL.

Epoch: 21       Training Loss: 3.982647         Validation Loss: 3.706
249
validation loss decreased from 3.775684 to 3.706249 | SAVING MODEL.

Epoch: 22       Training Loss: 3.959592         Validation Loss: 3.732
502

Epoch: 23       Training Loss: 3.941087         Validation Loss: 3.763
801

Epoch: 24       Training Loss: 3.885166         Validation Loss: 3.591
918
validation loss decreased from 3.706249 to 3.591918 | SAVING MODEL.

Epoch: 25       Training Loss: 3.868854         Validation Loss: 3.667
275
```

Out[35]:

```
<All keys matched successfully>
```

Type *Markdown* and LaTeX: α^2

(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 [36]:

```
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().nump
        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: 3.644251


Test Accuracy: 12% (107/836)
```

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.

(IMPLEMENTATION) Specify Data Loaders for the Dog Dataset

Use the code cell below to write three separate [data loaders (http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader)](http://pytorch.org/docs/master/data.html#torch.utils.data.DataLoader) 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 [0]:

```
## TODO: Specify data loaders

data_dir = 'dogImages/'

train_transforms = transforms.Compose([transforms.RandomRotation(30),
                                       transforms.Resize(255),
                                       transforms.RandomHorizontalFlip(),
                                       transforms.RandomResizedCrop(224),
                                       transforms.ToTensor(),
                                       transforms.Normalize([0.5, 0.5, 0.5],
                                                            [0.5, 0.5, 0.5])
                                      ])

valid_transforms = transforms.Compose([transforms.Resize(255),
                                      transforms.CenterCrop(224),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.5, 0.5, 0.5],
                                                           [0.5, 0.5, 0.5])
                                     ])

batch_size = 128

train_data = datasets.ImageFolder(os.path.join(data_dir,'train'), transform=train_t
trainloader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, shuffl

valid_data = datasets.ImageFolder(os.path.join(data_dir,'valid'), transform=valid_t
validloader = torch.utils.data.DataLoader(valid_data, batch_size=batch_size, shuffl

test_data = datasets.ImageFolder(os.path.join(data_dir,'test'), transform=valid_tra
testloader = torch.utils.data.DataLoader(test_data, batch_size=64, shuffle=False)
```

(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 [0]:

```
import torchvision.models as models
import torch.nn as nn

## TODO: Specify model architecture

model_transfer = models.resnet50(pretrained=True)

for parameter in model_transfer.parameters():
    parameter.requires_grad = False

input_features = model_transfer.fc.in_features

# new_classifier = nn.Sequential(nn.Linear(input_features, 4096),
#                                nn.ReLU(),
#                                nn.Dropout(0.5),
#                                nn.Linear(4096, 512),
#                                nn.ReLU(),
#                                nn.Dropout(0.5),
#                                nn.Linear(512, 133)
#                               )
model_transfer.fc = nn.Linear(input_features, 133)

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

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

Answer:

- I decided to pick the resnet50 model as it is better compared to VGG16. Resnet has a better time and space complexity than VGGnet owing to the lesser number of parameters. Resnet also accounts for the vanishing gradient problem by providing skip connections.
- I froze all the layers in the network and changed its classifier, replaced it with a new classifier that output is equal to the number of breeds

(IMPLEMENTATION) Specify Loss Function and Optimizer

Use the next code cell to specify a loss function (http://pytorch.org/docs/master/nn.html#loss-functions) and optimizer (http://pytorch.org/docs/master/optim.html). Save the chosen loss function as `criterion_transfer`, and the optimizer as `optimizer_transfer` below.

In [0]:

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

(IMPLEMENTATION) Train and Validate the Model

Train and validate your model in the code cell below. Save the final model parameters (http://pytorch.org/docs/master/notes/serialization.html) at filepath `'model_transfer.pt'`.

In [43]:

```
loaders_transfer = {'train': trainloader,
                    'valid': validloader,
                    'test': testloader}

# train the model
model_transfer = train(10, loaders_transfer, model_transfer, optimizer_transfer, cr

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

```
Epoch: 1        Training Loss: 3.268011         Validation Loss: 1.416
865
validation loss decreased from inf to 1.416865 | SAVING MODEL.

Epoch: 2        Training Loss: 1.597447         Validation Loss: 0.833
784
validation loss decreased from 1.416865 to 0.833784 | SAVING MODEL.

Epoch: 3        Training Loss: 1.216328         Validation Loss: 0.643
467
validation loss decreased from 0.833784 to 0.643467 | SAVING MODEL.

Epoch: 4        Training Loss: 1.076251         Validation Loss: 0.587
054
validation loss decreased from 0.643467 to 0.587054 | SAVING MODEL.

Epoch: 5        Training Loss: 1.003519         Validation Loss: 0.531
474
validation loss decreased from 0.587054 to 0.531474 | SAVING MODEL.

Epoch: 6        Training Loss: 0.904155         Validation Loss: 0.498
654
validation loss decreased from 0.531474 to 0.498654 | SAVING MODEL.

Epoch: 7        Training Loss: 0.872065         Validation Loss: 0.511
887

Epoch: 8        Training Loss: 0.838603         Validation Loss: 0.461
430
validation loss decreased from 0.498654 to 0.461430 | SAVING MODEL.

Epoch: 9        Training Loss: 0.809956         Validation Loss: 0.481
620

Epoch: 10       Training Loss: 0.818106         Validation Loss: 0.456
371
validation loss decreased from 0.461430 to 0.456371 | SAVING MODEL.
```

Out[43]:

```
<All keys matched successfully>
```

(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 [44]:

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

```
Test Loss: 0.473873


Test Accuracy: 85% (717/836)
```

(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 [0]:

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

# list of class names by index, i.e. a name can be accessed like class_names[0]
class_names = [item[4:].replace("_", " ") for item in train_data.classes]

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

    img = Image.open(img_path)

    pre_processing = transforms.Compose([transforms.Resize(255),
                                         transforms.CenterCrop(224),
                                         transforms.ToTensor(),
                                         transforms.Normalize([0.5, 0.5, 0.5],
                                                              [0.5, 0.5, 0.5])
                                        ])

    img = pre_processing(img).unsqueeze(0)

    if use_cuda:
        img = img.cuda()

    with torch.no_grad():
        model_transfer.eval()
        output = model_transfer(img)
        prediction = torch.argmax(output).item()

    return class_names[prediction]
```

Step 5: Write your Algorithm

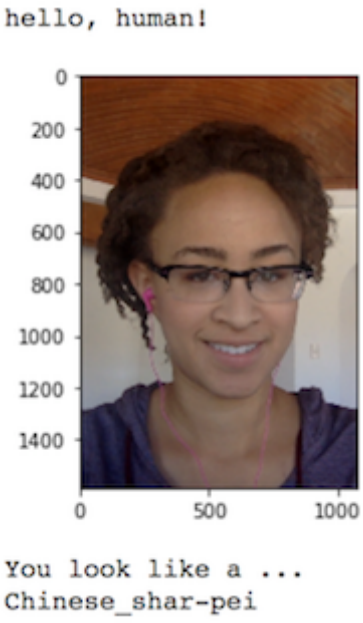
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- if a **dog** is detected in the image, return the predicted breed.

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

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_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!



(IMPLEMENTATION) Write your Algorithm

In [0]:

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

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

    plt.imshow(Image.open(img_path))
    plt.show()

    human_found = face_detector(img_path)
    dog_found = dog_detector(img_path)

    if human_found:
        print('Hello! You look like a ' + predict_breed_transfer(img_path))
    elif dog_found:
        print('This dog breed is ' + predict_breed_transfer(img_path))
    elif (not human_found) & (not dog_found):
        print('No Dog or Human found!!')
```

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?

(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.

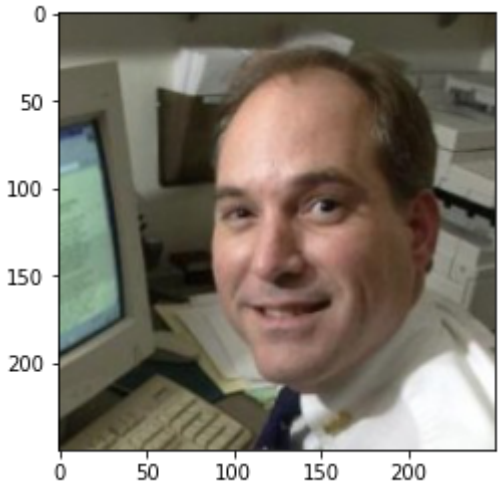
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer:

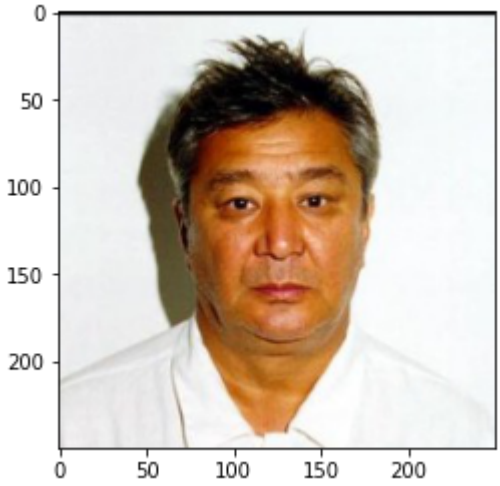
- A little to my surprise, My dog_detector gave a perfect result on the first 100 samples.
- I could improve the face_detector by training a neural network using transfer learning.
- I could improve the models (both scratch and tranfer learning) by training to for more epochs. (requires more computing power though)
- I could improve the models by using more generalized training and validation data.
- I could try to improve the models with weight initialisation.

In [53]:

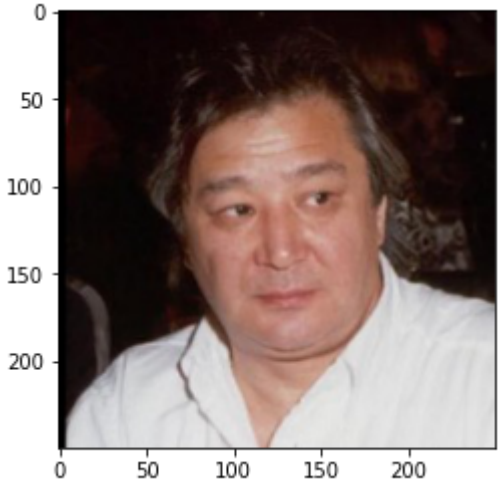
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

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



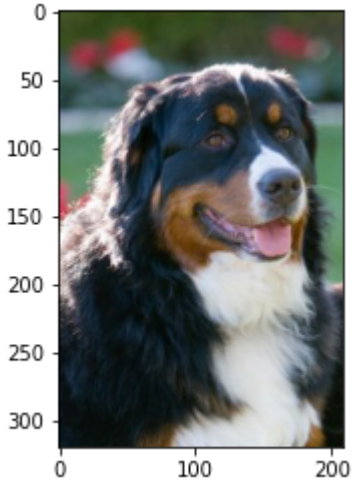
Hello! You look like a Chihuahua



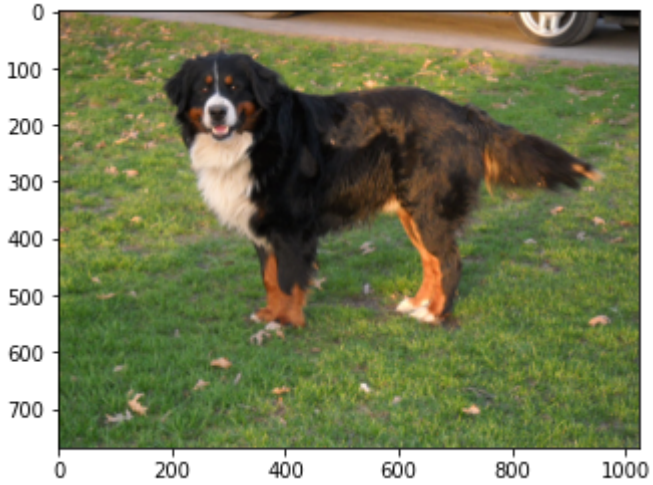
Hello! You look like a Dogue de bordeaux



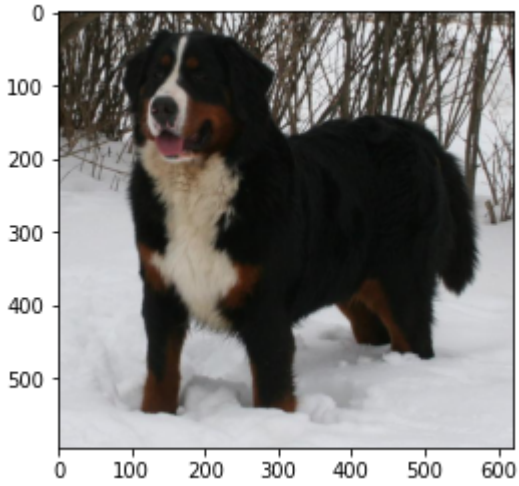
Hello! You look like a American water spaniel



This dog breed is Bernese mountain dog



This dog breed is Bernese mountain dog



This dog breed is Bernese mountain dog

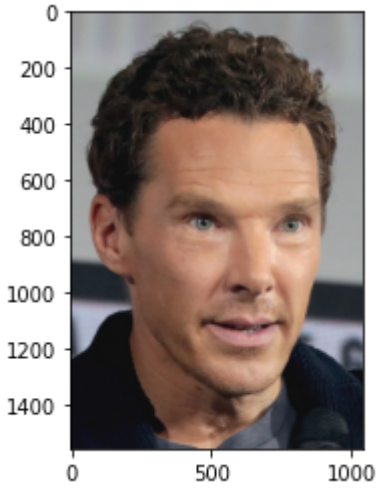
In [55]:

```
# Download my images from aws S3
!wget https://r-aravind-somedata.s3.amazonaws.com/my_images.zip
!unzip my_images.zip
```

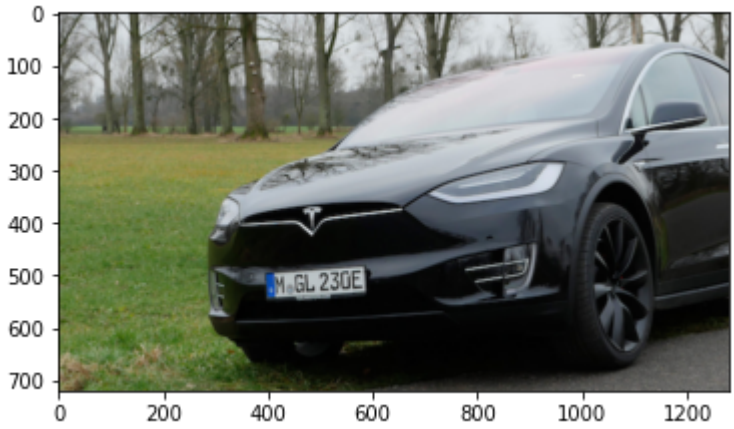
```
--2020-05-12 13:45:28--  https://r-aravind-somedata.s3.amazonaws.com/m
y_images.zip (https://r-aravind-somedata.s3.amazonaws.com/my_images.zi
p)
Resolving r-aravind-somedata.s3.amazonaws.com (r-aravind-somedata.s3.a
mazonaws.com)... 52.216.176.35
Connecting to r-aravind-somedata.s3.amazonaws.com (r-aravind-somedata.
s3.amazonaws.com)|52.216.176.35|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2157364 (2.1M) [application/zip]
Saving to: ‘my_images.zip’

my_images.zip       100%[===================>]   2.06M  4.01MB/s    in
0.5s

2020-05-12 13:45:29 (4.01 MB/s) - ‘my_images.zip’ saved [2157364/21573
64]

Archive:  my_images.zip
   creating: my_images/
  inflating: my_images/beckham.jpg
  inflating: my_images/dog.jpeg
  inflating: my_images/Benedict_Cumberbatch.jpg
  inflating: my_images/dog-puppy-on-garden-royalty-free-image-15869661
91.jpg
  inflating: my_images/Tesla_ModelX.jpg
  inflating: my_images/cat.jpg
```

In [56]:

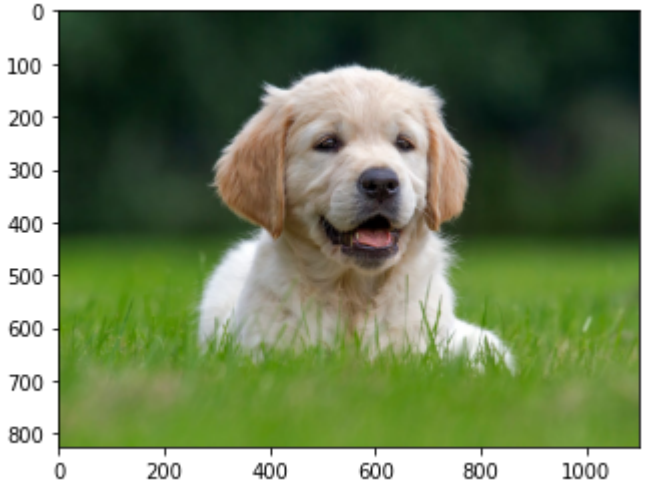
```
# test images on my computer
files = np.array(glob("my_images/*"))
for file in files:
    run_app(file)
```



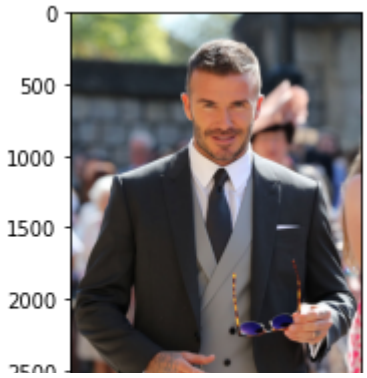
Hello! You look like a Havanese



No Dog or Human found!!



This dog breed is Golden retriever



Hello! You look like a Portuguese water dog

This dog breed is Golden retriever

No Dog or Human found!!

In [0]: