Artificial Intelligence Nanodegree

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 iPython 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 \n", "**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.

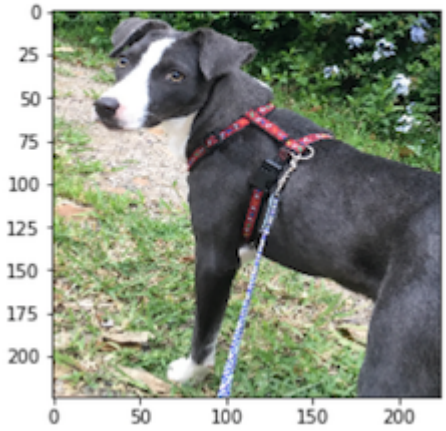
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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 IPython 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

Import Dog Dataset

In the code cell below, we import a dataset of dog images. We populate a few variables through the use of the `load_files` function from the scikit-learn library:

- `train_files`, `valid_files`, `test_files` - numpy arrays containing file paths to images
- `train_targets`, `valid_targets`, `test_targets` - numpy arrays containing onehot-encoded classification labels
- `dog_names` - list of string-valued dog breed names for translating labels

In [1]:
```
from sklearn.datasets import load_files
from keras.utils import np_utils
import numpy as np
from glob import glob

# define function to load train, test, and validation datasets
def load_dataset(path):
    data = load_files(path)
    dog_files = np.array(data['filenames'])
    dog_targets = np_utils.to_categorical(np.array(data['target']), 133)
    return dog_files, dog_targets

# load train, test, and validation datasets
train_files, train_targets = load_dataset('dogImages/train')
valid_files, valid_targets = load_dataset('dogImages/valid')
test_files, test_targets = load_dataset('dogImages/test')

# load list of dog names
dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*/"))]

# print statistics about the dataset
print('There are %d total dog categories.' % len(dog_names))
print('There are %s total dog images.\n' % len(np.hstack([train_files, valid_f
iles, test_files])))
print('There are %d training dog images.' % len(train_files))
print('There are %d validation dog images.' % len(valid_files))
print('There are %d test dog images.'% len(test_files))
```

```
Using TensorFlow backend.
```

```
There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array `human_files`.

In [2]:
```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

```
There are 13233 total human images.
```

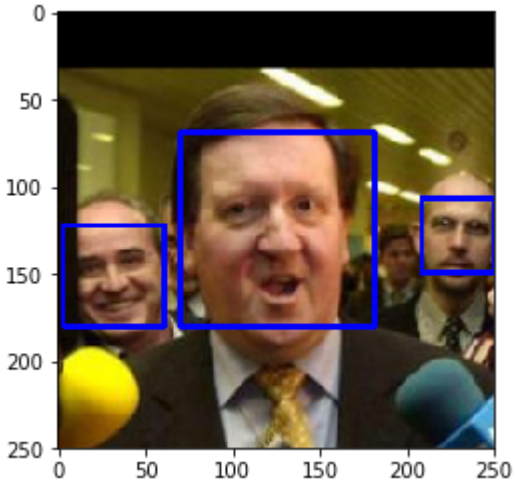
Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers (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 [3]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_al
t.xml')

# load color (BGR) image
img = cv2.imread(human_files[3])
# 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: 3
```



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 [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
```

(IMPLEMENTATION) Assess the Human Face Detector

Question 1: Use the code cell below to test the performance of the `face_detector` function.

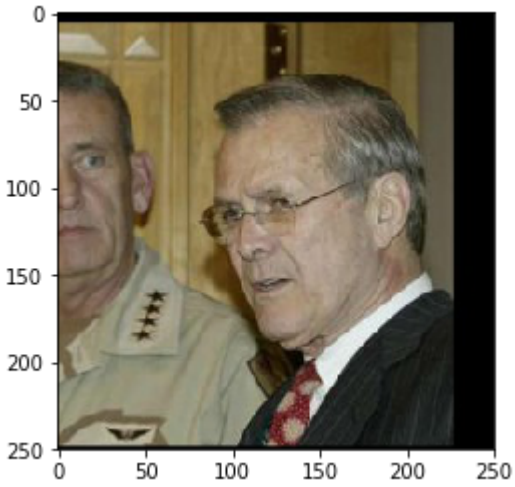
- What percentage of the first 100 images in `human_files` have a detected human face?
- What percentage of the first 100 images in `dog_files` have a detected human face?

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:

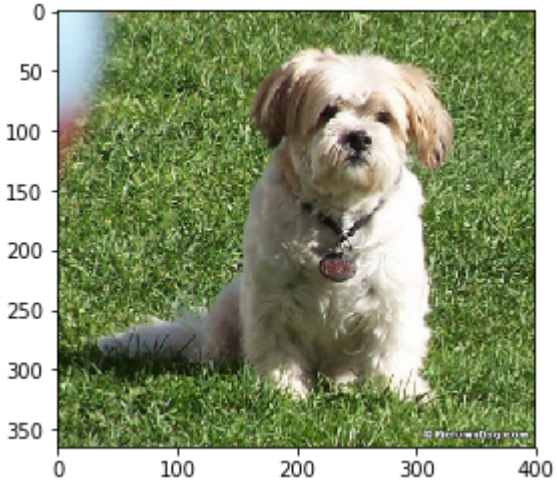
In [5]:
```
human_files_short = human_files[:100]
dog_files_short = train_files[:100]
# Do NOT modify the code above this line.

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

def disp_image(img_file):
    img = cv2.imread(img_file)
    # 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()

num_human=0
num_dog=0

print("incorectly detected images will be displayed:")
for h_file in human_files_short:
    if face_detector(h_file):
        num_human+=1
    else:
        print(h_file)
        disp_image(h_file)

for d_file in dog_files_short:
    if face_detector(d_file):
        num_dog+=1
        print(d_file)
        disp_image(d_file)
print('\nNum Human: ',num_human,'Num Dog: ',num_dog)
```

incorectly detected images will be displayed:
lfw/Donald_Rumsfeld/Donald_Rumsfeld_0083.jpg



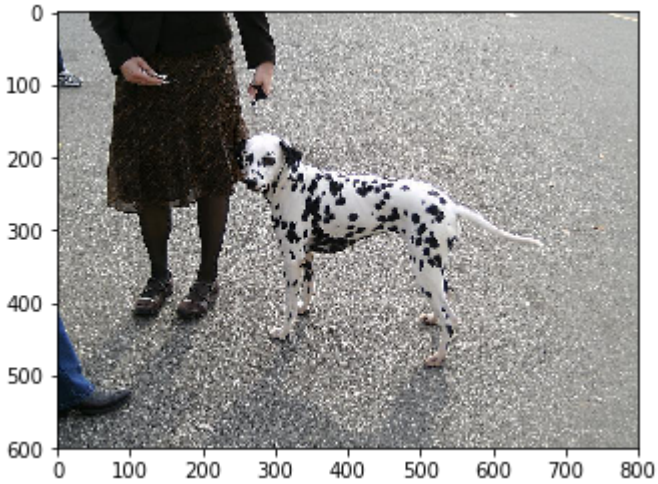
dogImages/train/095.Kuvasz/Kuvasz_06442.jpg

dogImages/train/099.Lhasa_apso/Lhasa_apso_06646.jpg

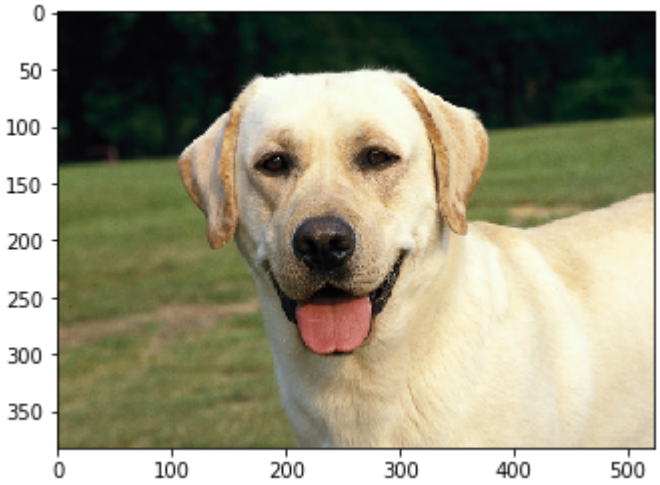


dogImages/train/009.American_water_spaniel/American_water_spaniel_00628.jpg

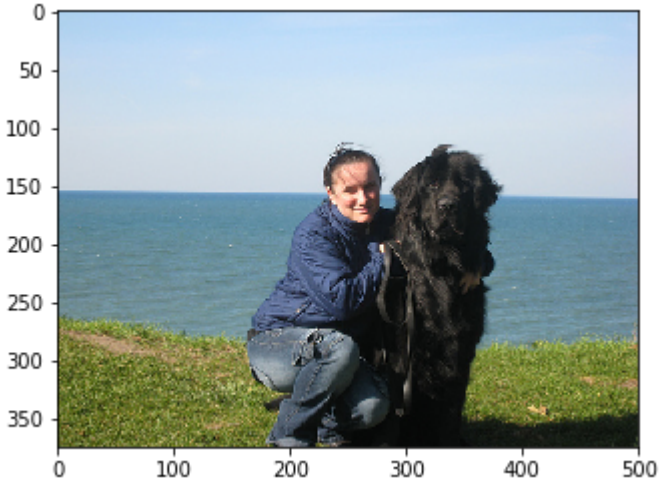
dogImages/train/057.Dalmatian/Dalmatian_04023.jpg



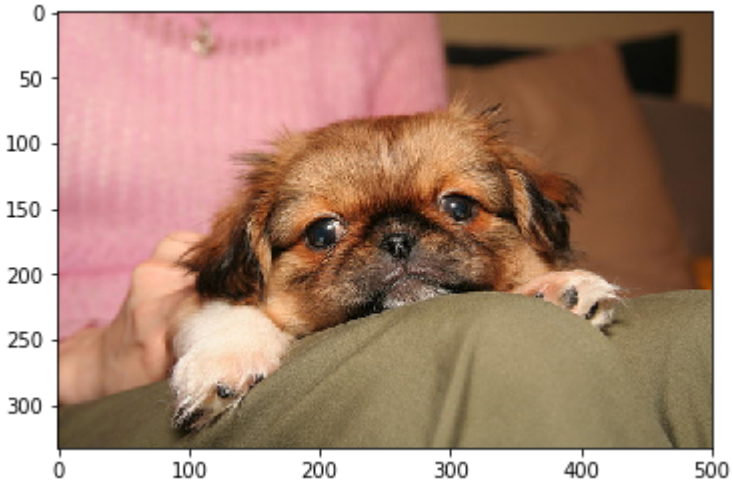
dogImages/train/096.Labrador_retriever/Labrador_retriever_06474.jpg



dogImages/train/106.Newfoundland/Newfoundland_06989.jpg

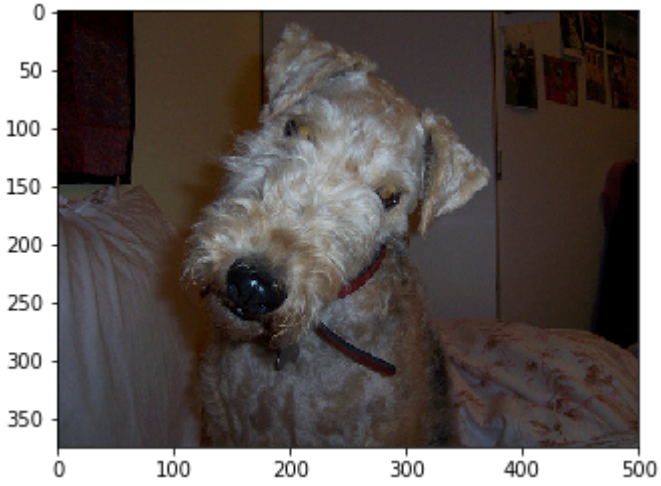


dogImages/train/117.Pekingese/Pekingese_07559.jpg

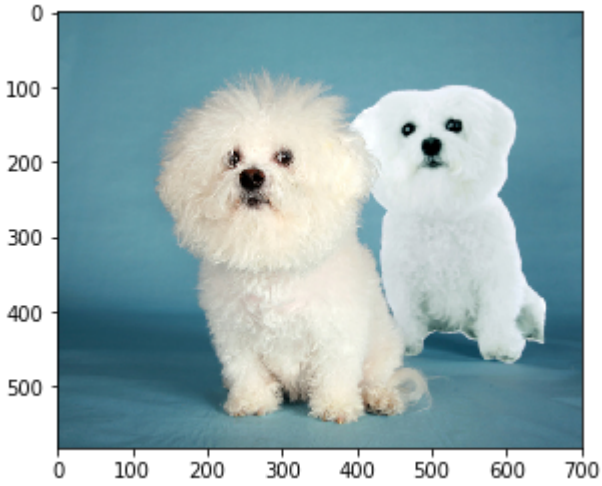


dogImages/train/039.Bull_terrier/Bull_terrier_02805.jpg

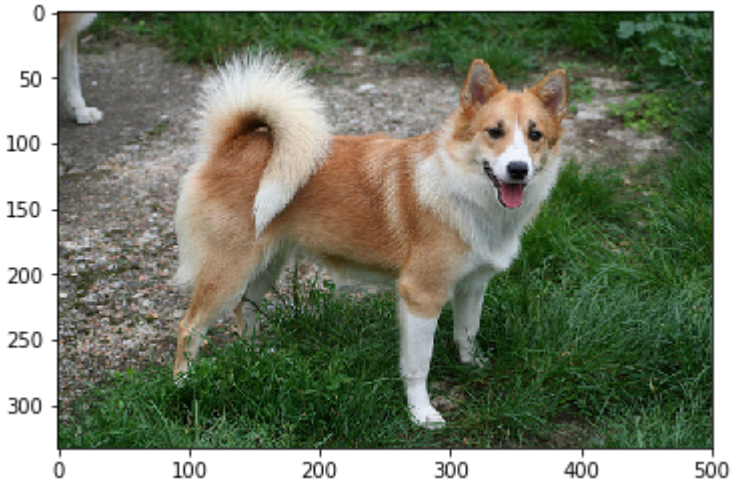
dogImages/train/097.Lakeland_terrier/Lakeland_terrier_06516.jpg




```
dogImages/train/024.Bichon_frise/Bichon_frise_01771.jpg
```




```
dogImages/train/084.Icelandic_sheepdog/Icelandic_sheepdog_05705.jpg
```




```
Num Human:  99 Num Dog:  11
```

99 humans were seen our of 100. 11 Dogs were seen, out of 100

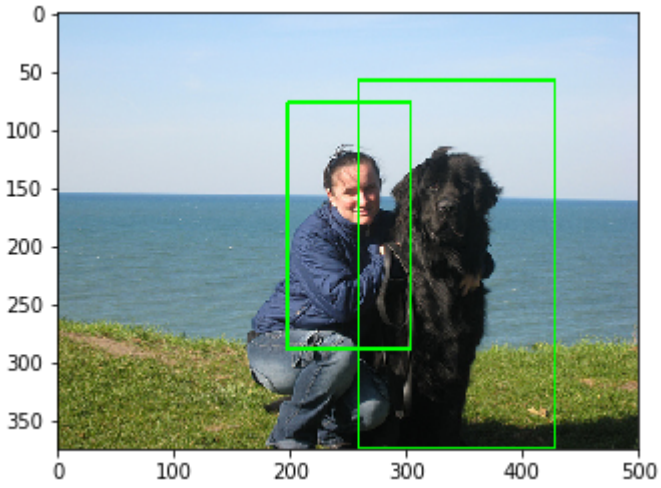
Question 2: This algorithmic choice necessitates that we communicate to the user that we accept human images only when they provide a clear view of a face (otherwise, we risk having unneccessarily frustrated users!). In your opinion, is this a reasonable expectation to pose on the user? If not, can you think of a way to detect humans in images that does not necessitate an image with a clearly presented face?

Answer:

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 each of the datasets.

In [6]:
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.

# A person trained object detector

# Take the image file name from the command line
file_name = 'lfw/Donald_Rumsfeld/Donald_Rumsfeld_0083.jpg'
file_name= 'dogImages/train/106.Newfoundland/Newfoundland_06989.jpg'
#file_name='test_images/atticus.jpg'
orig = cv2.imread(file_name)

# load color (BGR) image
#img = cv2.imread(human_files[3])
# convert BGR image to grayscale
im = cv2.cvtColor(orig, cv2.COLOR_BGR2GRAY)

#im = imutils.resize(im, width=min(400, im.shape[1]))

hog = cv2.HOGDescriptor()
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

(rects, weights) = hog.detectMultiScale(im, winStride=(4, 4),padding=(32, 32),
 scale=1.05 )

print('objects seen:',len(rects))
# draw the original bounding boxes
for (x, y, w, h) in rects:
    cv2.rectangle(orig, (x, y), (x + w, y + h), (0, 255, 0), 2)

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(orig, cv2.COLOR_BGR2RGB)

# display the image, along with bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
objects seen: 2
```



Answer: One specific issue seen with this detection method is the inability to detect any face in an image that is not turned to the camera and oriented up/down with the camera. While this may be a reasonable expectation, as this method is fast and can be easily implemented for a live camera view, a method that detects faces in other poses may be justified for other applications.

Above, in the code section, I have implemented a person detector, using a provided model, from openCV. This detector is based on the HOG method (https://dl.acm.org/citation.cfm?id=1069007)[1] This detector can be retrained on a face data set, and should have better ability to see faces in a larger number of circumstances, and may be useful as is in the context of detecting objects humans in general, rather than just faces, allowing us to make a determination of breed based on the entire returned object.

See this paper (https://arxiv.org/abs/1701.08289)[2] for one example of using deep learning to detect faces in more varied conditions. It continues the ideas below of using transfer learning and expands upo it with vaious techniques to achieve a better recognition rate

- [1] Dalal,Triggs,Histograms of Oriented Gradients for Human Detection(2005). DOI: 10.1109/CVPR.2005.177
- [2] Xudong Sun, Pengcheng Wu, Steven C.H. Hoi, Face Detection using Deep Learning: An Improved Faster RCNN Approach(2017). arXiv:1701.0828

Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006) model to detect dogs in images. Our first line of code downloads the ResNet-50 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). Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

In [7]:
```
from keras.applications.resnet50 import ResNet50

# define ResNet50 model
ResNet50_model = ResNet50(weights='imagenet')
```

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

$$(\text{nb\_samples}, \text{rows}, \text{columns}, \text{channels}),$$

where `nb_samples` corresponds to the total number of images (or samples), and `rows`, `columns`, and `channels` correspond to the number of rows, columns, and channels for each image, respectively.

The `path_to_tensor` function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

$$(1, 224, 224, 3).$$

The `paths_to_tensor` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

$$(\text{nb\_samples}, 224, 224, 3).$$

Here, `nb_samples` is the number of samples, or number of images, in the supplied array of image paths. It is best to think of `nb_samples` as the number of 3D tensors (where each 3D tensor corresponds to a different image) in your dataset!

In [8]:
```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D
 tensor
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths
)]
    return np.vstack(list_of_tensors)
```

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as $[103.939, 116.779, 123.68]$ and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function `preprocess_input`. If you're curious, you can check the code for `preprocess_input` here (https://github.com/fchollet/keras/blob/master/keras/applications/imagenet_utils.py).

Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the `predict` method, which returns an array whose i-th entry is the model's predicted probability that the image belongs to the i-th ImageNet category. This is implemented in the `ResNet50_predict_labels` function below.

By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [9]:
```
from keras.applications.resnet50 import preprocess_input, decode_predictions

def ResNet50_predict_labels(img_path):
    # returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))
```

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 ResNet-50 model, we need only check if the `ResNet50_predict_labels` function above returns a value between 151 and 268 (inclusive).

We 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 [10]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

(IMPLEMENTATION) Assess the Dog Detector

Question 3: 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 [11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

num_human=0
num_dog=0
for h_file in human_files_short:
    if dog_detector(h_file):
        print(h_file)
        disp_image(h_file)
    else:
        num_human+=1

for d_file in dog_files_short:
    if dog_detector(d_file):
        num_dog+=1
    else:
        print(d_file)
        disp_image(d_file)
print('\nNum human:',num_human,'Num dog:',num_dog)
```

```
Num human: 100 Num dog: 100
```

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

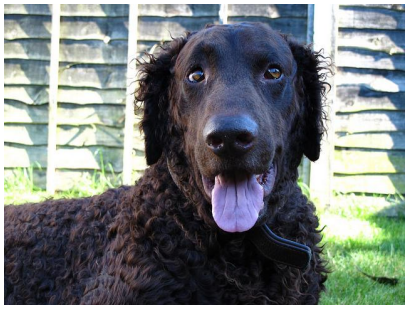
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of at least 1%. In Step 5 of this notebook, you will have the opportunity to use transfer learning to create a CNN that attains greatly improved accuracy.

Be careful with adding too many trainable layers! More parameters means longer training, which means you are more likely to need a GPU to accelerate the training process. Thankfully, Keras provides a handy estimate of the time that each epoch is likely to take; you can extrapolate this estimate to figure out how long it will take for your algorithm to train.

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 great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

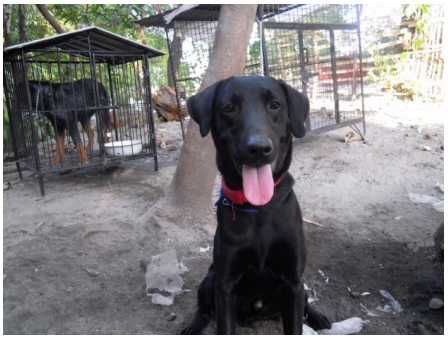
Brittany	**Welsh Springer Spaniel**
	

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**

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!

Pre-process the Data

We rescale the images by dividing every pixel in every image by 255.

In [12]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

# pre-process the data for Keras
train_tensors = paths_to_tensor(train_files).astype('float32')/255
valid_tensors = paths_to_tensor(valid_files).astype('float32')/255
test_tensors = paths_to_tensor(test_files).astype('float32')/255
```

```
100%|██████████| 6680/6680 [00:53<00:00, 124.12it/s]
100%|██████████| 835/835 [00:06<00:00, 138.53it/s]
100%|██████████| 836/836 [00:06<00:00, 139.12it/s]
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
model.summary()
```

We have imported some Python modules to get you started, but feel free to import as many modules as you need. If you end up getting stuck, here's a hint that specifies a model that trains relatively fast on CPU and attains >1% test accuracy in 5 epochs:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

Question 4: Outline the steps you took to get to your final CNN architecture and your reasoning at each step. If you chose to use the hinted architecture above, describe why you think that CNN architecture should work well for the image classification task.

Answer:

Using the hinted architecture as starting point to acheive the goal accuracy, I have enhanced it by referring to some of the succesful models such as LeNet, VGG, and Inception. We can observe that one of the primary differences they bring in is more layers. Beyond just adding more layers, I also noted that the kernel and stride are varied so that the layer could see a different(larger) portion of the image then the original. I used this idea for my final model by doing adding several convolutional layers(increasing filter size in the deeper layers), as well as by varying the kernel to 3x3.

To enhance the model, we could refer to the current state-of-the-art, and copy their architecture, but we would see large increases in training time compared to what we have used below.

In [13]:
```
#based on example model
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',

                         input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D())

model.add(Dense(133, activation='softmax'))


model.summary()
#Test accuracy: 2.5120%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 16)      208
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 112, 112, 32)      2080
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 56, 56, 64)        8256
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 28, 28, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [14]:
```
#Alt model, more layers

from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=16, kernel_size=2, padding='same', activation='relu',

                         input_shape=(224, 224, 3)))
model.add(Conv2D(filters=32, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=128, kernel_size=2, padding='same', activation='relu'
))
model.add(Conv2D(filters=256, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))

model.add(GlobalAveragePooling2D())
model.add(Dense(133, activation='softmax'))


model.summary()

#Test accuracy: 3.1100%
```

Layer (type)	Output Shape	Param #
conv2d_4 (Conv2D)	(None, 224, 224, 16)	208
conv2d_5 (Conv2D)	(None, 224, 224, 32)	2080
max_pooling2d_5 (MaxPooling2	(None, 112, 112, 32)	0
conv2d_6 (Conv2D)	(None, 112, 112, 64)	8256
max_pooling2d_6 (MaxPooling2	(None, 56, 56, 64)	0
conv2d_7 (Conv2D)	(None, 56, 56, 128)	32896
conv2d_8 (Conv2D)	(None, 56, 56, 256)	131328
max_pooling2d_7 (MaxPooling2	(None, 28, 28, 256)	0
global_average_pooling2d_2 (	(None, 256)	0
dense_2 (Dense)	(None, 133)	34181

Total params: 208,949.0
Trainable params: 208,949.0
Non-trainable params: 0.0

In [15]:
```
# similar to example, but varying kernel size/strides
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=16, kernel_size=1, padding='same', activation='relu',

                         input_shape=(224, 224, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3,strides =3, padding='same', activat
ion='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D())

model.add(Dense(133, activation='softmax'))


model.summary()
#Test accuracy: 3.4689%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 224, 224, 16)      64
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 38, 38, 32)        4640
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 19, 19, 32)        0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 19, 19, 64)        8256
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 9, 9, 64)          0
_________________________________________________________________
global_average_pooling2d_3 ( (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               8645
=================================================================
Total params: 21,605.0
Trainable params: 21,605.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [16]:
```
#Alt model, more layers, varying kernel size/strides

from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
model.add(Conv2D(filters=16, kernel_size=1, padding='same', activation='relu',

                         input_shape=(224, 224, 3)))
model.add(Conv2D(filters=32, kernel_size=1, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=64, kernel_size=3, strides=3, padding='same', activat
ion='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=128, kernel_size=2,strides =2, padding='same', activa
tion='relu'))
model.add(Conv2D(filters=256, kernel_size=2, padding='same', activation='relu'
))
model.add(MaxPooling2D(pool_size=2))

model.add(GlobalAveragePooling2D())
model.add(Dense(133, activation='softmax'))


model.summary()

#est accuracy: 5.5024%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_12 (Conv2D)           (None, 224, 224, 16)      64
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 224, 224, 32)      544
_________________________________________________________________
max_pooling2d_11 (MaxPooling (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_14 (Conv2D)           (None, 38, 38, 64)        18496
_________________________________________________________________
max_pooling2d_12 (MaxPooling (None, 19, 19, 64)        0
_________________________________________________________________
conv2d_15 (Conv2D)           (None, 10, 10, 128)       32896
_________________________________________________________________
conv2d_16 (Conv2D)           (None, 10, 10, 256)       131328
_________________________________________________________________
max_pooling2d_13 (MaxPooling (None, 5, 5, 256)         0
_________________________________________________________________
global_average_pooling2d_4 ( (None, 256)               0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               34181
=================================================================
Total params: 217,509.0
Trainable params: 217,509.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [17]:
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=[
'accuracy'])
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

In [18]:
```
from keras.callbacks import ModelCheckpoint

### TODO: specify the number of epochs that you would like to use to train the
 model.

epochs = 5

### Do NOT modify the code below this line.

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratc
h.hdf5',
                               verbose=1, save_best_only=True)

model.fit(train_tensors, train_targets,
          validation_data=(valid_tensors, valid_targets),
          epochs=epochs, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8856 - acc: 0.
0080Epoch 00000: val_loss improved from inf to 4.87001, saving model to saved
_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 253s - loss: 4.8855 - acc: 0.007
9 - val_loss: 4.8700 - val_acc: 0.0108
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8611 - acc: 0.
0117Epoch 00001: val_loss improved from 4.87001 to 4.80635, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 257s - loss: 4.8612 - acc: 0.011
7 - val_loss: 4.8064 - val_acc: 0.0228
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.7293 - acc: 0.
0249Epoch 00002: val_loss improved from 4.80635 to 4.71867, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 258s - loss: 4.7284 - acc: 0.025
0 - val_loss: 4.7187 - val_acc: 0.0275
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.5573 - acc: 0.
0353Epoch 00003: val_loss improved from 4.71867 to 4.52053, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 259s - loss: 4.5579 - acc: 0.035
2 - val_loss: 4.5205 - val_acc: 0.0407
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.4253 - acc: 0.
0458Epoch 00004: val_loss improved from 4.52053 to 4.43800, saving model to s
aved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 260s - loss: 4.4240 - acc: 0.045
8 - val_loss: 4.4380 - val_acc: 0.0491
```

Out[18]:
```
<keras.callbacks.History at 0x7f53801a9c50>
```

Load the Model with the Best Validation Loss

In [19]:
```
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
```

Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 1%.

In [20]:
```
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(tensor, axis=0
))) for tensor in test_tensors]

# report test accuracy
test_accuracy = 100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_tar
gets, axis=1))/len(dog_breed_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 5.1435%
```

Step 4: Use a CNN to Classify Dog Breeds

To reduce training time without sacrificing accuracy, we show you how to train a CNN using transfer learning. In the following step, you will get a chance to use transfer learning to train your own CNN.

Obtain Bottleneck Features

In [21]:
```
bottleneck_features = np.load('bottleneck_features/DogVGG16Data.npz')
train_VGG16 = bottleneck_features['train']
valid_VGG16 = bottleneck_features['valid']
test_VGG16 = bottleneck_features['test']
```

Model Architecture

The model uses the the pre-trained VGG-16 model as a fixed feature extractor, where the last convolutional output of VGG-16 is fed as input to our model. We only add a global average pooling layer and a fully connected layer, where the latter contains one node for each dog category and is equipped with a softmax.

In [22]:
```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_5 ( (None, 512)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [23]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metr
ics=['accuracy'])
```

Train the Model

In [24]:
```
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.VGG16.hdf5'
,
                               verbose=1, save_best_only=True)

VGG16_model.fit(train_VGG16, train_targets,
          validation_data=(valid_VGG16, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6580/6680 [============================>.] - ETA: 0s - loss: 12.1254 - acc:
0.1236Epoch 00000: val_loss improved from inf to 10.46719, saving model to
saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 12.0932 - acc: 0.12
54 - val_loss: 10.4672 - val_acc: 0.2299
Epoch 2/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.9557 - acc:
0.2955Epoch 00001: val_loss improved from 10.46719 to 9.92555, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.9715 - acc: 0.294
5 - val_loss: 9.9256 - val_acc: 0.2862
Epoch 3/20
6540/6680 [============================>.] - ETA: 0s - loss: 9.5627 - acc:
0.3547Epoch 00002: val_loss improved from 9.92555 to 9.68321, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5672 - acc: 0.354
2 - val_loss: 9.6832 - val_acc: 0.3257
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 9.3309 - acc:
0.3811Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.3323 - acc: 0.381
0 - val_loss: 9.6936 - val_acc: 0.3353
Epoch 5/20
6660/6680 [============================>.] - ETA: 0s - loss: 9.2433 - acc:
0.3997Epoch 00004: val_loss improved from 9.68321 to 9.55334, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.2393 - acc: 0.399
4 - val_loss: 9.5533 - val_acc: 0.3437
Epoch 6/20
6460/6680 [============================>.] - ETA: 0s - loss: 9.0467 - acc:
0.4155Epoch 00005: val_loss improved from 9.55334 to 9.30099, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.0508 - acc: 0.415
0 - val_loss: 9.3010 - val_acc: 0.3581
Epoch 7/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.9005 - acc:
0.4271Epoch 00006: val_loss improved from 9.30099 to 9.23519, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.9116 - acc: 0.426
6 - val_loss: 9.2352 - val_acc: 0.3701
Epoch 8/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.8201 - acc:
0.4351Epoch 00007: val_loss improved from 9.23519 to 9.15520, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7994 - acc: 0.436
2 - val_loss: 9.1552 - val_acc: 0.3629
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.6445 - acc:
0.4486Epoch 00008: val_loss improved from 9.15520 to 9.10952, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.6500 - acc: 0.448
4 - val_loss: 9.1095 - val_acc: 0.3725
Epoch 10/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.5640 - acc:
0.4508Epoch 00009: val_loss improved from 9.10952 to 8.99027, saving model
```

```
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5428 - acc: 0.452
1 - val_loss: 8.9903 - val_acc: 0.3689
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 8.2087 - acc:
0.4692Epoch 00010: val_loss improved from 8.99027 to 8.63772, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.2013 - acc: 0.469
5 - val_loss: 8.6377 - val_acc: 0.3904
Epoch 12/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.0025 - acc:
0.4908- ETA: 1s - loEpoch 00011: val_loss improved from 8.63772 to 8.58152,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0290 - acc: 0.489
2 - val_loss: 8.5815 - val_acc: 0.3952
Epoch 13/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.0056 - acc:
0.4924Epoch 00012: val_loss improved from 8.58152 to 8.55483, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.9743 - acc: 0.494
5 - val_loss: 8.5548 - val_acc: 0.4012
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.8976 - acc:
0.5021Epoch 00013: val_loss improved from 8.55483 to 8.50316, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.9006 - acc: 0.501
9 - val_loss: 8.5032 - val_acc: 0.4072
Epoch 15/20
6560/6680 [============================>.] - ETA: 0s - loss: 7.8461 - acc:
0.5049Epoch 00014: val_loss improved from 8.50316 to 8.42033, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.8534 - acc: 0.504
2 - val_loss: 8.4203 - val_acc: 0.4108
Epoch 16/20
6640/6680 [============================>.] - ETA: 0s - loss: 7.7700 - acc:
0.5120Epoch 00015: val_loss improved from 8.42033 to 8.26814, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.7742 - acc: 0.511
8 - val_loss: 8.2681 - val_acc: 0.4096
Epoch 17/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.6843 - acc:
0.5133Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 7.6831 - acc: 0.513
3 - val_loss: 8.2698 - val_acc: 0.4096
Epoch 18/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.4757 - acc:
0.5230Epoch 00017: val_loss improved from 8.26814 to 8.26495, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.4729 - acc: 0.522
9 - val_loss: 8.2649 - val_acc: 0.3952
Epoch 19/20
6580/6680 [============================>.] - ETA: 0s - loss: 7.3064 - acc:
0.5325Epoch 00018: val_loss improved from 8.26495 to 8.01406, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.3087 - acc: 0.532
2 - val_loss: 8.0141 - val_acc: 0.4359
Epoch 20/20
```

```
6460/6680 [============================>.] - ETA: 0s - loss: 7.2717 - acc:
0.5393Epoch 00019: val_loss improved from 8.01406 to 7.95947, saving model
to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.2392 - acc: 0.541
3 - val_loss: 7.9595 - val_acc: 0.4263
```

Out[24]: <keras.callbacks.History at 0x7f5272fddc88>

Load the Model with the Best Validation Loss

In [25]:
```
VGG16_model.load_weights('saved_models/weights.best.VGG16.hdf5')
```

Test the Model

Now, we can use the CNN to test how well it identifies breed within our test dataset of dog images. We print the test accuracy below.

In [26]:
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axi
s=0))) for feature in test_VGG16]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(test_targets
, axis=1))/len(VGG16_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 44.9761%
```

Predict Dog Breed with the Model

In [27]:
```
from extract_bottleneck_features import *

def VGG16_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_VGG16(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG16_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

Step 5: 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.

In Step 4, we used transfer learning to create a CNN using VGG-16 bottleneck features. In this section, you must use the bottleneck features from a different pre-trained model. To make things easier for you, we have pre-computed the features for all of the networks that are currently available in Keras:

- VGG-19 (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogVGG19Data.npz) bottleneck features
- ResNet-50 (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogResnet50Data.npz) bottleneck features
- Inception (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogInceptionV3Data.npz) bottleneck features
- Xception (https://s3-us-west-1.amazonaws.com/udacity-aind/dog-project/DogXceptionData.npz) bottleneck features

The files are encoded as such:

```
Dog{network}Data.npz
```

where `{network}`, in the above filename, can be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`. Pick one of the above architectures, download the corresponding bottleneck features, and store the downloaded file in the `bottleneck_features/` folder in the repository.

(IMPLEMENTATION) Obtain Bottleneck Features

In the code block below, extract the bottleneck features corresponding to the train, test, and validation sets by running the following:

```
bottleneck_features = np.load('bottleneck_features/Dog{network}Data.npz')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

In [97]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features = np.load('bottleneck_features/DogXceptionData.npz')
train_Xception = bottleneck_features['train']
valid_Xception = bottleneck_features['valid']
test_Xception = bottleneck_features['test']
```

(IMPLEMENTATION) Model Architecture

Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
<your model's name>.summary()
```

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 have chosen to use the Xception model, with transfer learning, for the final CNN architecture. This model uses a more advanced architecture with significantly more layers, and so serves as a good base. We use a GAP(Global Average Pooling) layer to flattern the result, and then finaly make the final prediction with a single Dense layer.

In [98]:
```
### TODO: Define your architecture.
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

# Base model provided in example above
Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[1
:]))
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()

#Test accuracy: 86.2440%, lr=default(.001), multiple trial
#Test accuracy: 85.5263%%, lr=.0001, one trial
#Test accuracy: 85.8852%, lr=.0005, one trial
#Test accuracy: 80.9809%, lr=.01, one trial
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_13  (None, 2048)              0
_________________________________________________________________
dense_16 (Dense)             (None, 133)               272517
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [31]:
```
### TODO: Define your architecture.
# add another fully connected layer with dropout
Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[1
:]))
Xception_model.add(Dense(532, activation='relu'))
Xception_model.add(Dropout(0.2))
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()

#Test accuracy: 83.4928%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_7 ( (None, 2048)              0
_________________________________________________________________
dense_7 (Dense)              (None, 532)               1090068
_________________________________________________________________
dropout_1 (Dropout)          (None, 532)               0
_________________________________________________________________
dense_8 (Dense)              (None, 133)               70889
=================================================================
Total params: 1,160,957.0
Trainable params: 1,160,957.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [32]:
```
### TODO: Define your architecture.
# add another fully connected layer with dropout
Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[1
:]))
Xception_model.add(Dense(532, activation='relu'))
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()

#Test accuracy: 82.6555%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_8 ( (None, 2048)              0
_________________________________________________________________
dense_9 (Dense)              (None, 532)               1090068
_________________________________________________________________
dense_10 (Dense)             (None, 133)               70889
=================================================================
Total params: 1,160,957.0
Trainable params: 1,160,957.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [33]:
```
### TODO: Define your architecture.
# add another fully connected layer with dropout
Xception_model = Sequential()
Xception_model.add(GlobalAveragePooling2D(input_shape=train_Xception.shape[1
:]))
Xception_model.add(Dense(266, activation='relu'))
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()

#Test accuracy: 81.1005%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_9 ( (None, 2048)              0
_________________________________________________________________
dense_11 (Dense)             (None, 266)               545034
_________________________________________________________________
dense_12 (Dense)             (None, 133)               35511
=================================================================
Total params: 580,545.0
Trainable params: 580,545.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [34]:
```
### TODO: Define your architecture.
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

# Base model provided in example above
Xception_model = Sequential()
Xception_model.add(Conv2D(filters=64, kernel_size=5,strides=5, padding='same',
 activation='relu',input_shape=train_Xception.shape[1:]))
Xception_model.add(MaxPooling2D(pool_size=2))
Xception_model.add(Conv2D(filters=256, kernel_size=1, strides=1, padding='sam
e', activation='relu'))
Xception_model.add(MaxPooling2D(pool_size=1))
Xception_model.add(GlobalAveragePooling2D())
Xception_model.add(Dense(133, activation='softmax'))

Xception_model.summary()

#Test accuracy: 79.1866%
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_17 (Conv2D)           (None, 2, 2, 64)          3276864
_________________________________________________________________
max_pooling2d_14 (MaxPooling (None, 1, 1, 64)          0
_________________________________________________________________
conv2d_18 (Conv2D)           (None, 1, 1, 256)         16640
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 1, 1, 256)         0
_________________________________________________________________
global_average_pooling2d_10  (None, 256)               0
_________________________________________________________________
dense_13 (Dense)             (None, 133)               34181
=================================================================
Total params: 3,327,685.0
Trainable params: 3,327,685.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

Answer: As in the example, we compile the model. Opportunity for change here lies with the optimizer, where we may choose to use a different optimizer, or change the learning rate. various rates were sampled, with the best to date being the default, .001

In [99]:
```
### TODO: Compile the model.
from keras import optimizers

rmsprop= optimizers.rmsprop(lr=.001)
Xception_model.compile(loss='categorical_crossentropy', optimizer=rmsprop, met
rics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

You are welcome to augment the training data (https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html), but this is not a requirement.

Answer: Here we have added 'Early Stopping' to the model, as well as upped the epochs/batch size slightly.

In [100]:
```
### TODO: Train the model.
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.Xception.hd
f5',
                               verbose=1,monitor='val_acc', save_best_only=Tru
e)

from keras.callbacks import EarlyStopping
e_stop=EarlyStopping(monitor='val_acc', min_delta=0.00009, patience=11, verbos
e=2, mode='auto')

model_history=Xception_model.fit(train_Xception, train_targets,
              validation_data=(valid_Xception, valid_targets),
              epochs=32, batch_size=32, callbacks=[checkpointer,e_stop], verbo
se=1)
```

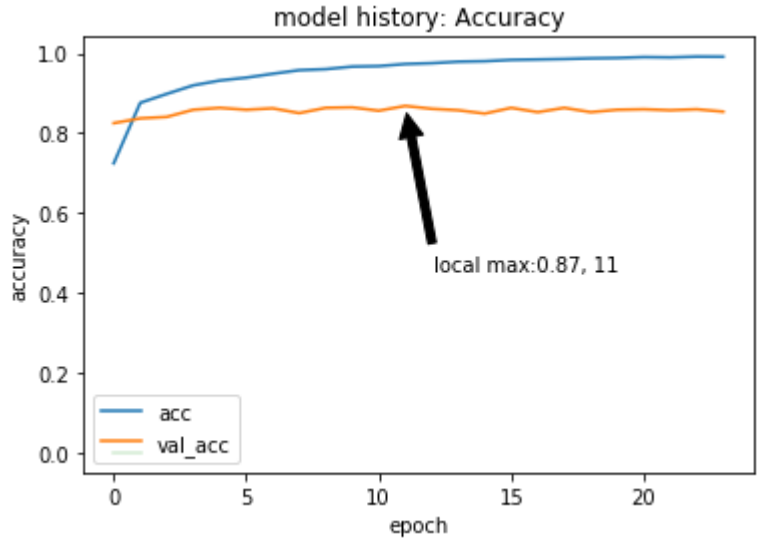
```
Train on 6680 samples, validate on 835 samples
Epoch 1/32
6656/6680 [============================>.] - ETA: 0s - loss: 1.1570 - acc:
0.7246- ETA: 1s - loss: 1.2627 - aEpoch 00000: val_acc improved from -inf t
o 0.82515, saving model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 16s - loss: 1.1558 - acc: 0.72
46 - val_loss: 0.5341 - val_acc: 0.8251
Epoch 2/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.3945 - acc:
0.8761Epoch 00001: val_acc improved from 0.82515 to 0.83713, saving model t
o saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 5s - loss: 0.3938 - acc: 0.876
0 - val_loss: 0.5060 - val_acc: 0.8371
Epoch 3/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.3028 - acc:
0.8989- ETA: 2s - loss: 0.2890 - acc: - ETAEpoch 00002: val_acc improved fr
om 0.83713 to 0.84072, saving model to saved_models/weights.best.Xception.h
df5
6680/6680 [==============================] - 5s - loss: 0.3052 - acc: 0.898
4 - val_loss: 0.4903 - val_acc: 0.8407
Epoch 4/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.2481 - acc:
0.9202Epoch 00003: val_acc improved from 0.84072 to 0.85868, saving model t
o saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 5s - loss: 0.2487 - acc: 0.919
8 - val_loss: 0.4717 - val_acc: 0.8587
Epoch 5/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.2114 - acc:
0.9319- ETA: 1s - lossEpoch 00004: val_acc improved from 0.85868 to 0.8634
7, saving model to saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 5s - loss: 0.2113 - acc: 0.931
9 - val_loss: 0.4730 - val_acc: 0.8635
Epoch 6/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.1884 - acc:
0.9389- ETA: 3 - ETA: 1s - loss: 0.1 - ETA: 0s - loss: 0.1861 - accEpoch 00
005: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.1878 - acc: 0.939
1 - val_loss: 0.4796 - val_acc: 0.8587
Epoch 7/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.1587 - acc:
0.9491Epoch 00006: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.1600 - acc: 0.948
8 - val_loss: 0.4792 - val_acc: 0.8623
Epoch 8/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.1381 - acc:
0.9579Epoch 00007: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.1384 - acc: 0.957
8 - val_loss: 0.5188 - val_acc: 0.8503
Epoch 9/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.1216 - acc:
0.9602- Epoch 00008: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.1213 - acc: 0.960
3 - val_loss: 0.5044 - val_acc: 0.8635
Epoch 10/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.1097 - acc:
0.9666- ETAEpoch 00009: val_acc improved from 0.86347 to 0.86467, saving mo
del to saved_models/weights.best.Xception.hdf5
```

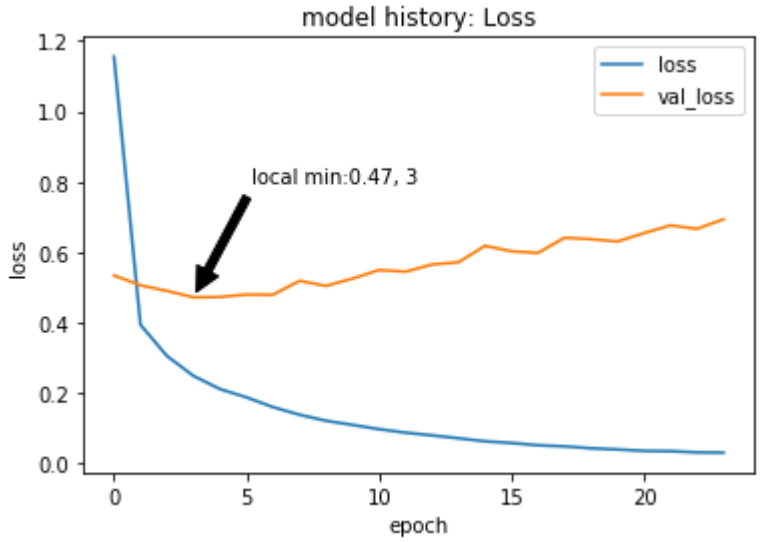
```
6680/6680 [==============================] - 5s - loss: 0.1095 - acc: 0.966
8 - val_loss: 0.5251 - val_acc: 0.8647
Epoch 11/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0974 - acc:
0.9680- ETA: 2s - loss: 0.0 - ETA: 1s - loss: 0.Epoch 00010: val_acc did no
t improve
6680/6680 [==============================] - 5s - loss: 0.0974 - acc: 0.968
0 - val_loss: 0.5494 - val_acc: 0.8563
Epoch 12/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0875 - acc:
0.9733Epoch 00011: val_acc improved from 0.86467 to 0.86826, saving model t
o saved_models/weights.best.Xception.hdf5
6680/6680 [==============================] - 5s - loss: 0.0875 - acc: 0.973
2 - val_loss: 0.5448 - val_acc: 0.8683
Epoch 13/32
6624/6680 [============================>.] - ETA: 0s - loss: 0.0792 - acc:
0.9755Epoch 00012: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0798 - acc: 0.975
3 - val_loss: 0.5651 - val_acc: 0.8611
Epoch 14/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0717 - acc:
0.9790Epoch 00013: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0716 - acc: 0.979
0 - val_loss: 0.5716 - val_acc: 0.8575
Epoch 15/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0630 - acc:
0.9802- ETA: 2s - loss: 0.0653 - acc: 0. - ETA: 2s - loss: 0.0637 - acc:  -
ETEpoch 00014: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0629 - acc: 0.980
2 - val_loss: 0.6183 - val_acc: 0.8491
Epoch 16/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0583 - acc:
0.9835- ETA: 1s - loss: 0.Epoch 00015: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0581 - acc: 0.983
5 - val_loss: 0.6027 - val_acc: 0.8635
Epoch 17/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0518 - acc:
0.9845Epoch 00016: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0517 - acc: 0.984
6 - val_loss: 0.5980 - val_acc: 0.8527
Epoch 18/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0486 - acc:
0.9857Epoch 00017: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0486 - acc: 0.985
8 - val_loss: 0.6412 - val_acc: 0.8635
Epoch 19/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0432 - acc:
0.9874Epoch 00018: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0431 - acc: 0.987
4 - val_loss: 0.6371 - val_acc: 0.8527
Epoch 20/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0399 - acc:
0.9883Epoch 00019: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0398 - acc: 0.988
3 - val_loss: 0.6306 - val_acc: 0.8587
Epoch 21/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0358 - acc:
```

```
0.9905Epoch 00020: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0357 - acc: 0.990
6 - val_loss: 0.6544 - val_acc: 0.8599
Epoch 22/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0347 - acc:
0.9901- ETA: 0s - loss: 0.0344 Epoch 00021: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0350 - acc: 0.989
8 - val_loss: 0.6764 - val_acc: 0.8575
Epoch 23/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0315 - acc:
0.9917Epoch 00022: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0314 - acc: 0.991
8 - val_loss: 0.6666 - val_acc: 0.8599
Epoch 24/32
6656/6680 [============================>.] - ETA: 0s - loss: 0.0305 - acc:
0.9916Epoch 00023: val_acc did not improve
6680/6680 [==============================] - 5s - loss: 0.0307 - acc: 0.991
5 - val_loss: 0.6937 - val_acc: 0.8539
Epoch 00023: early stopping
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [101]:
```
#added a visualization of the training
import matplotlib.pyplot as plt

plt.plot(model_history.history['acc'])
plt.plot(model_history.history['val_acc'])
plt.plot((0,0))
plt.title('model history: Accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['acc', 'val_acc'], loc='lower left')
max_val_acc=max(model_history.history['val_acc'])
max_val_acc_index=model_history.history['val_acc'].index(max_val_acc)
text_acc='local max:{:.2}, {}'.format(max_val_acc,max_val_acc_index)
plt.annotate(text_acc, xy=(max_val_acc_index,max_val_acc),  xycoords='data',
            xytext=(0.8, 0.5), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
plt.show()

plt.plot(model_history.history['loss'])
plt.plot(model_history.history['val_loss'])
plt.title('model history: Loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['loss', 'val_loss'], loc='upper right')
min_val_loss=min(model_history.history['val_loss'])
min_val_loss_index=model_history.history['val_loss'].index(min_val_loss)
text_loss='local min:{:.2}, {}'.format(min_val_loss,min_val_loss_index)
plt.annotate(text_loss, xy=(min_val_loss_index,min_val_loss),  xycoords='data'
,
            xytext=(0.5, 0.7), textcoords='axes fraction',
            arrowprops=dict(facecolor='black', shrink=0.05),
            horizontalalignment='right', verticalalignment='top',
            )
plt.show()
```





In [102]:
```
### TODO: Load the model weights with the best validation loss.
Xception_model.load_weights('saved_models/weights.best.Xception.hdf5')
```

(IMPLEMENTATION) Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 60%.

In [103]:
```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
Xception_predictions = [np.argmax(Xception_model.predict(np.expand_dims(featur
e, axis=0))) for feature in test_Xception]

# report test accuracy
test_accuracy = 100*np.sum(np.array(Xception_predictions)==np.argmax(test_targ
ets, axis=1))/len(Xception_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 86.0048%
```

(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.

Similar to the analogous function in Step 5, your function should have three steps:

1. Extract the bottleneck features corresponding to the chosen CNN model.
2. Supply the bottleneck features as input to the model to return the predicted vector. Note that the argmax of this prediction vector gives the index of the predicted dog breed.
3. Use the `dog_names` array defined in Step 0 of this notebook to return the corresponding breed.

The functions to extract the bottleneck features can be found in `extract_bottleneck_features.py`, and they have been imported in an earlier code cell. To obtain the bottleneck features corresponding to your chosen CNN architecture, you need to use the function

```
extract_{network}
```

where `{network}`, in the above filename, should be one of `VGG19`, `Resnet50`, `InceptionV3`, or `Xception`.

In [104]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

from extract_bottleneck_features import *

def Xception_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Xception(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = Xception_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]


def Xception_predict_mutlibreed(img_path): # return the top 3 breeds
    # extract bottleneck features
    bottleneck_feature = extract_Xception(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = Xception_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    multibreed={}
    #print(predicted_vector)
    for i in (range(len(predicted_vector[0]))):
            multibreed[dog_names[i]]=predicted_vector[0][i]*100
    return sorted(multibreed, key=multibreed.get, reverse=True)[:3]
```

Step 6: Write your Algorithm

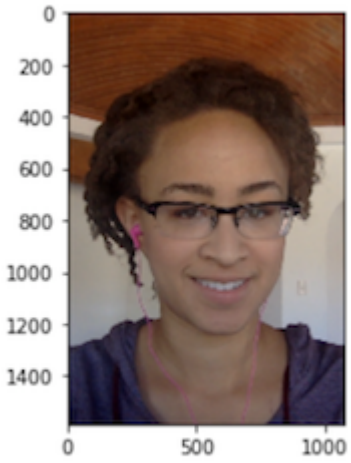
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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 5 to predict dog breed.

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

```
hello, human!
```




```
You look like a ...
Chinese_shar-pei
```

(IMPLEMENTATION) Write your Algorithm

In [105]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
def breed_detector(img_file_path):
    if face_detector(img_file_path):
        print('Human!:')
    elif dog_detector(img_file_path):
        print('Dog!:')
    else:
        print('Whut! Neither a dog or a human! ERROR! Prediction made anyway:'
) #just make a prediction
    return(Xception_predict_breed(img_file_path))

def multibreed_detector(img_file_path):
    if face_detector(img_file_path):
        print('Human!:')
    elif dog_detector(img_file_path):
        print('Dog!:')
    else:
        print('Whut! Neither a dog or a human! ERROR! Prediction made anyway:'
) #just make a prediction
    return(Xception_predict_mutlibreed(img_file_path))
```

Step 7: 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:

The algorithm was tested on pictures from my personal libary, and performed well, as expected!

We can note that the dog images in particular were correctly labled, despite suffering from relatively poor image quality

The human labeling was also consistent. The images were of family memebers, with the family members that are noted for their resemblance sharing a multibreed label. One human image was not detected, but this was due to the constraints of the image: It was from an low angle and suffered from some lighting issues, and is predicted by the human recognition method used. Despite this, the breed classifier correctly labeled the image similar to the other family members, who, as mentioned already, have been noted to bear a resemblance to the pictured human.

For good measure, I have also run the test on examples from the images dir, to allow for better visual confirmation of results. These performed in a reasonable manor for each image, with the majority of the images that were in the original data set matching their label, and other images being classified in a reasonable manor

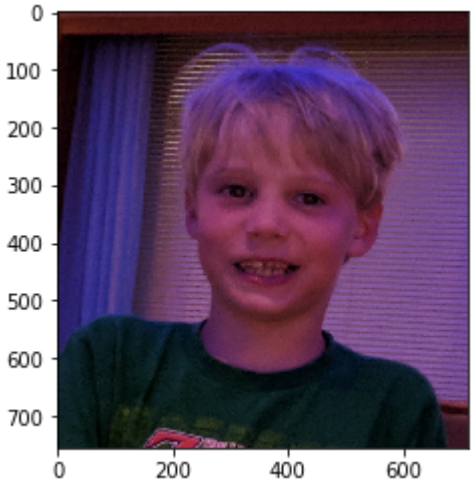
Several improvements to the model can be made:

1. Improve the human face detector for better off-angle prediction, as mentioned in that section
2. When testing, only use the rectangle that actually includeds the human/dog. The current human detector could be modified to do this quickly, but the current dog detector is unsuited to this task currently.
3. Improve the model for higher accuracy. This may be done by continued optimization of the hyper-parameters, and/or by updating the model. Other tools to improve the model would be larger training sets, and techniques such as hard-negative mining, that allow the model to augment the existing training sets
4. The logic for human/dog should be inverted, as the dog detector was more acurate for dogs. This is dependant on the current implementation, but would need to be re-evaluated in the case of modified detectors.

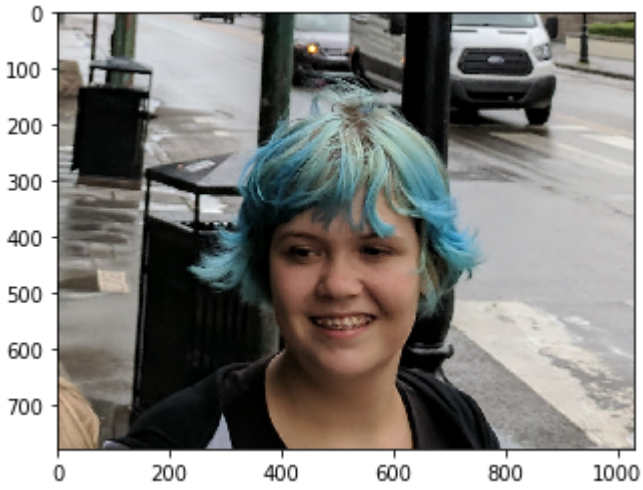
Note : Capture some test images from the first webcam on your system by running "python facecam.py haarcascades\haarcascade_frontalface_alt.xml" from the project directory. Press 's' to save an image, and 'q' to quit. if during capture, you see the green bounding box, the image will be detected as human. If the box isn't seen, the image won't be detected as human. The model doesn't perform quickly enough to add dogg breed to the live video, but that may be a possible enhancement.

In [ ]:
```
## 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.

#ADDED
#test the classifications :-)

test_img_paths =glob("test_images/*")+glob("images/*")


for file in tqdm(test_img_paths):
    print(file)
    print(breed_detector(file))
    print(multibreed_detector(file))
    disp_image(file)
```

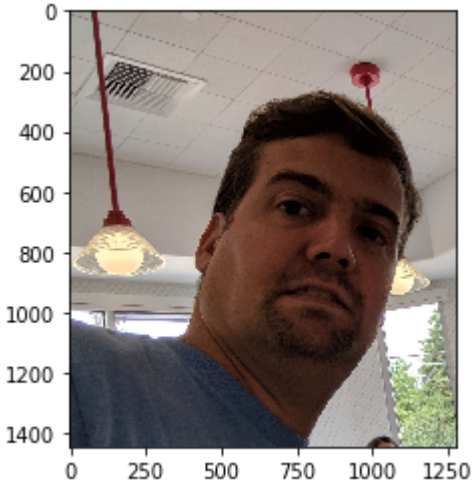
```
  0%|          | 0/19 [00:00<?, ?it/s]
test_images/Lincoln.jpg
Human!:
Dachshund
Human!:
['Dachshund', 'Cavalier_king_charles_spaniel', 'Cardigan_welsh_corgi']
```




```
  5%|▌         | 1/19 [00:36<10:49, 36.06s/it]
test_images/Sasha.jpg
Human!:
Chinese_crested
Human!:
['Chinese_crested', 'Cavalier_king_charles_spaniel', 'Irish_water_spaniel']
```

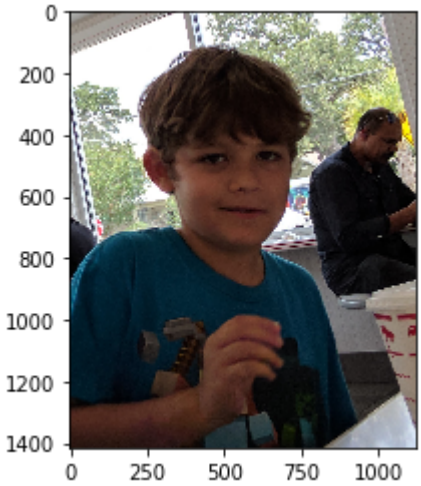



```
 11%|█         | 2/19 [01:11<10:09, 35.85s/it]
test_images/Luke.jpg
Whut! Neither a dog or a human! ERROR! Prediction made anyway:
Lowchen
Whut! Neither a dog or a human! ERROR! Prediction made anyway:
['Lowchen', 'Brussels_griffon', 'Cavalier_king_charles_spaniel']
```



16%|▉ | 3/19 [01:49<09:41, 36.37s/it]

```
test_images/atticus.jpg
Human!:
Cavalier_king_charles_spaniel
Human!:
['Cavalier_king_charles_spaniel', 'Dachshund', 'Kuvasz']
```

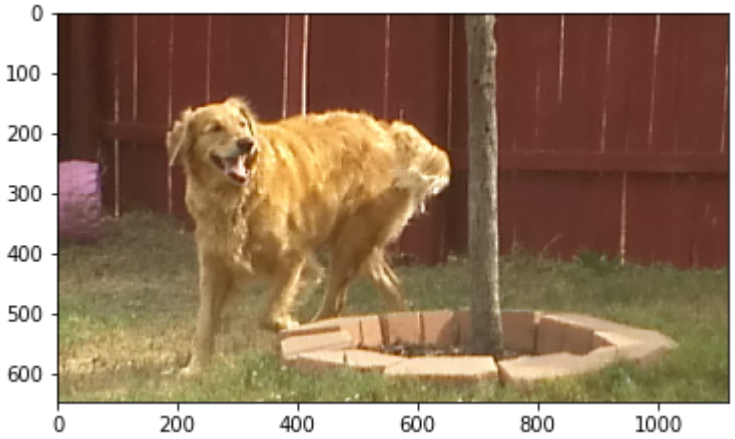


21%|▉ | 4/19 [02:26<09:11, 36.75s/it]

```
test_images/carrie.jpg
Human!:
Brussels_griffon
Human!:
['Brussels_griffon', 'Petit_basset_griffon_vendeen', 'Dachshund']
```

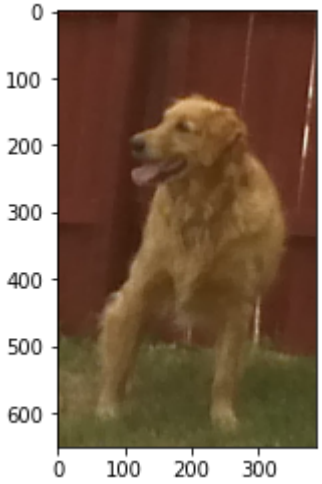
26%|██▌ | 5/19 [03:12<09:12, 39.47s/it]

```
test_images/sarah-2.jpg
Dog!:
Golden_retriever
Dog!:
['Golden_retriever', 'Anatolian_shepherd_dog', 'Nova_scotia_duck_tolling_retr
iever']
```



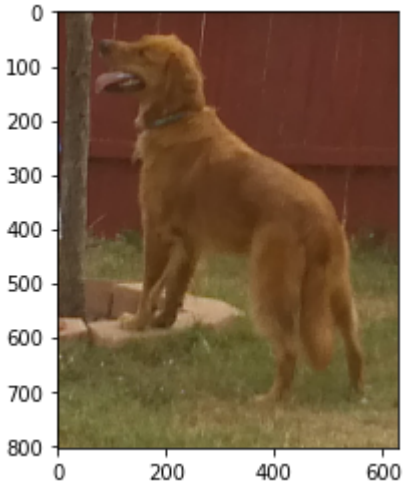
32%|███▏ | 6/19 [03:55<08:48, 40.64s/it]

```
test_images/sarah.jpg
Dog!:
Golden_retriever
Dog!:
['Golden_retriever', 'Otterhound', 'Nova_scotia_duck_tolling_retriever']
```



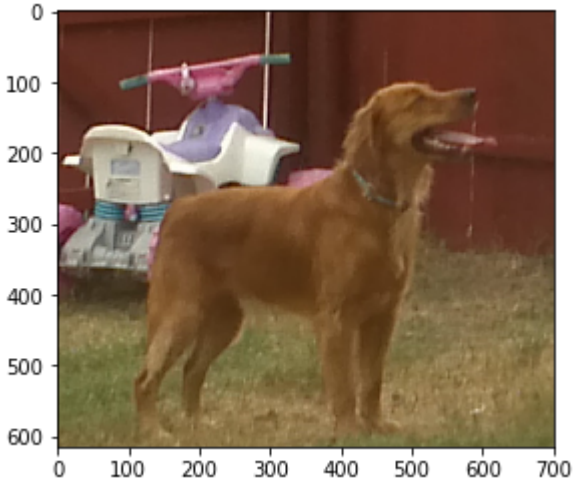
37%|███▊ | 7/19 [04:37<08:10, 40.90s/it]

```
test_images/loki-2.jpg
Dog!:
Golden_retriever
Dog!:
['Golden_retriever', 'Nova_scotia_duck_tolling_retriever', 'Irish_setter']
```



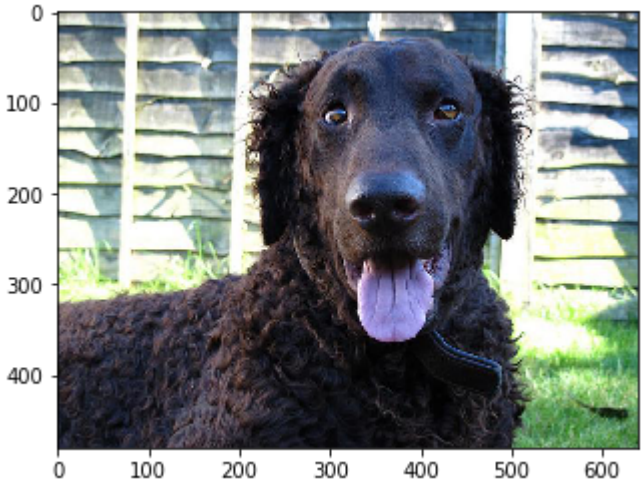
42%|████▏ | 8/19 [05:20<07:36, 41.53s/it]

```
test_images/Loki.jpg
Dog!:
Golden_retriever
Dog!:
['Golden_retriever', 'Irish_setter', 'Nova_scotia_duck_tolling_retriever']
```



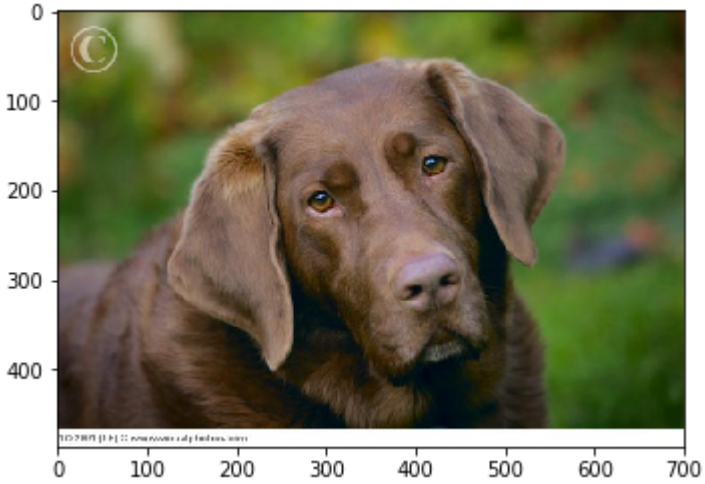
47%|████▋ | 9/19 [06:04<07:03, 42.37s/it]

images/Curly-coated_retriever_03896.jpg
Dog!:
Curly-coated_retriever
Dog!:
['Curly-coated_retriever', 'American_water_spaniel', 'Poodle']



53%|█████▎ | 10/19 [06:50<06:30, 43.35s/it]

images/Labrador_retriever_06455.jpg
Dog!:
Labrador_retriever
Dog!:
['Labrador_retriever', 'Chesapeake_bay_retriever', 'German_shorthaired_pointe
r']

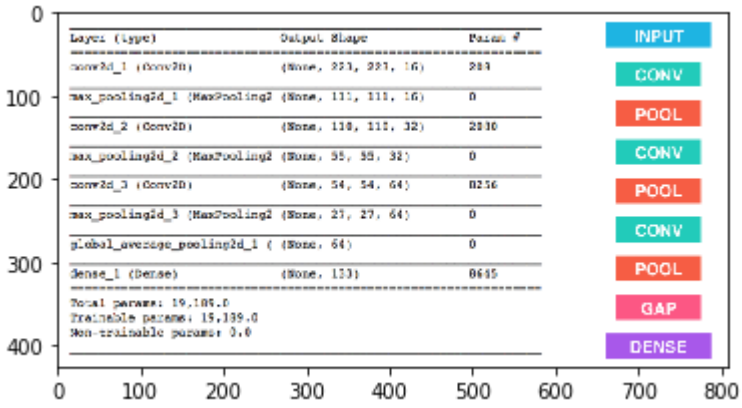


58%|█████▊ | 11/19 [07:37<05:55, 44.39s/it]

images/Labrador_retriever_06449.jpg
Dog!:
Labrador_retriever
Dog!:
['Labrador_retriever', 'Great_dane', 'Greyhound']

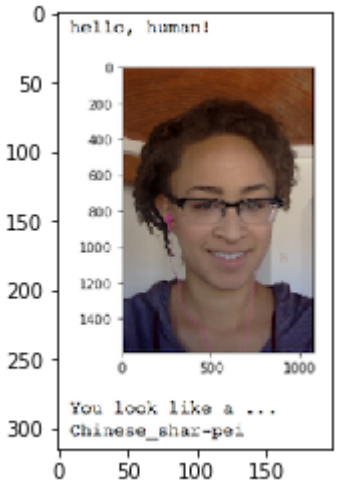
63%|██████▎ | 12/19 [08:25<05:19, 45.65s/it]

images/sample_cnn.png
Whut! Neither a dog or a human! ERROR! Prediction made anyway:
American_staffordshire_terrier
Whut! Neither a dog or a human! ERROR! Prediction made anyway:
['American_staffordshire_terrier', 'Chinese_shar-pei', 'Greyhound']



68%|██████▊ | 13/19 [09:17<04:45, 47.64s/it]

images/sample_human_output.png
Human!:
American_staffordshire_terrier
Human!:
['American_staffordshire_terrier', 'Cardigan_welsh_corgi', 'Chinese_shar-pe
i']



74%|███████▍ | 14/19 [10:08<04:01, 48.36s/it]

images/sample_dog_output.png
Whut! Neither a dog or a human! ERROR! Prediction made anyway:

EOF