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.

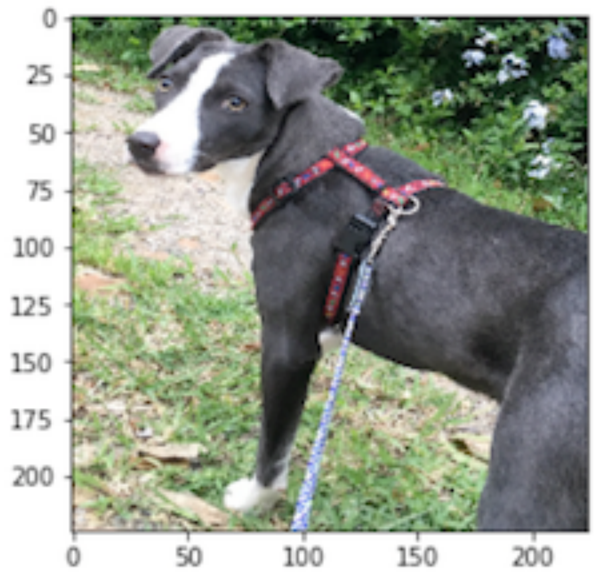
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Use a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 6: Write your Algorithm
- Step 7: Test Your Algorithm

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))
```

```
/Users/yoon/anaconda3/lib/python3.6/site-packages/h5py/__init__.py
:36: FutureWarning: Conversion of the second argument of issubdtyp
e from `float` to `np.floating` is deprecated. In future, it will
be treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
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

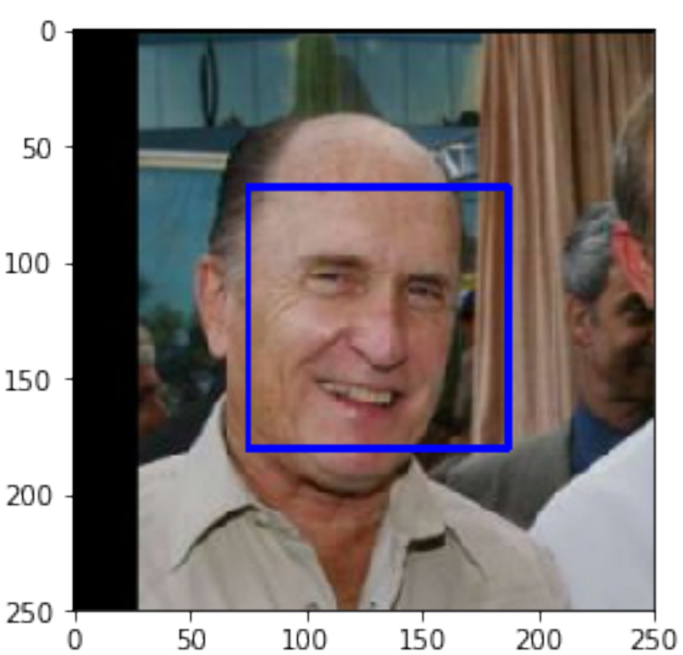
We use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). 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_alt
.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: 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 [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: I simply searched first 100 image files in 'human_files_short' and 'dog_files_short' respectively and counted the number of True value. That count is the percentage of the first 100 images since I searched 100 images in total.

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.
count_in_humanFile = 0
count_in_dogFile = 0

for i in range(len(human_files_short)):
    if face_detector(human_files_short[i]):
        count_in_humanFile += 1
    elif face_detector(dog_files_short[i]):
        count_in_dogFile += 1
print("Humans files = ", count_in_humanFile, "%", " " "Dog files = ", count_in
_dogFile, "%")
```

```
Humans files =  100 %  Dog files =  0 %
```

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: If the algorithm can only detect on a certain pose of human's face, it would be able to detect very limited variations of human faces. Recently, google has published a paper called FaceNet[1] and it showed strong invariance in terms of illumination and pose.

[1] Florian Schroff, Dmitry Kalenichenko, and James Philbin. "FaceNet: A Unified Embedding for Face Recognition and Clustering" published in '2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)'.`

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.
```

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: The code is the same as 'face_detector' above

In [11]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
count_in_humanFile = 0
count_in_dogFile = 0

for i in range(len(human_files_short)):
    if dog_detector(human_files_short[i]):
        count_in_humanFile += 1
    elif dog_detector(dog_files_short[i]):
        count_in_dogFile += 1

print("Human files = ", count_in_humanFile, "%", "Dog files = ", count_in_dogF
ile, "%")
```

```
Human files =  0 % Dog files =  100 %
```

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

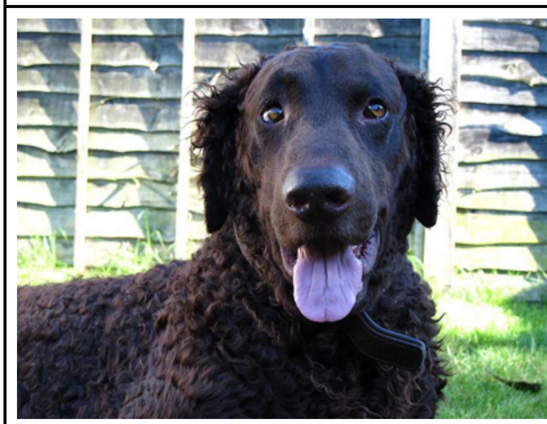
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

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

print(train_tensors.shape, valid_tensors.shape, test_tensors.shape)
```

```
100%|██████████| 6680/6680 [01:42<00:00, 65.28it/s]
100%|██████████| 835/835 [00:13<00:00, 63.76it/s]
100%|██████████| 836/836 [00:13<00:00, 63.22it/s]

(6680, 224, 224, 3) (835, 224, 224, 3) (836, 224, 224, 3)
```

(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: I chose model similar to sample architecture above. Instead of conducting GlobalAveragePooling, I erased that step and inserted flattening layer. In my opinion, the model losses too much image information by global average pooling and it decreases validation accuracy. In my experiment, I got test accuracy about 4.1% with global average pooling and about 9.4% without it. In addition, there was no big differences of training time (38.4 sec vs 39.2sec in 5 epoch).

In my opinion, the reason why CNN is good for image classification is that CNN is really good at detecting specific features such as edge, pattern, and etc. By modeling human eye's receptive field, kernels can be sensitive to locally connected pixels of image and this local sensitivity made computer classify images well in terms of objects.

In [13]:

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

model = Sequential()

### TODO: Define your architecture.

""" Model #1 """
model.add(Conv2D(filters=16, kernel_size=3, strides=2, padding='same', activat
ion='relu', input_shape=(224,224,3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=3, strides=2, padding='same', activat
ion='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=3, strides=2, padding='same', activat
ion='relu'))
model.add(MaxPooling2D(pool_size=2))
#model.add(GlobalAveragePooling2D())
model.add(Flatten())
model.add(Dense(133, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 112, 112, 16)      448
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 56, 56, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 32)        4640
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 3, 3, 64)          0
_________________________________________________________________
flatten_2 (Flatten)          (None, 576)               0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               76741
=================================================================
Total params: 100,325
Trainable params: 100,325
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [14]:

```
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 [15]:

```
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
6680/6680 [==============================] - 34s 5ms/step - loss:
4.8332 - acc: 0.0174 - val_loss: 4.7222 - val_acc: 0.0491

Epoch 00001: val_loss improved from inf to 4.72221, saving model t
o saved_models/weights.best.from_scratch.hdf5
Epoch 2/5
6680/6680 [==============================] - 34s 5ms/step - loss:
4.5010 - acc: 0.0503 - val_loss: 4.3596 - val_acc: 0.0515

Epoch 00002: val_loss improved from 4.72221 to 4.35957, saving mod
el to saved_models/weights.best.from_scratch.hdf5
Epoch 3/5
6680/6680 [==============================] - 35s 5ms/step - loss:
4.0970 - acc: 0.0931 - val_loss: 4.2102 - val_acc: 0.0850

Epoch 00003: val_loss improved from 4.35957 to 4.21021, saving mod
el to saved_models/weights.best.from_scratch.hdf5
Epoch 4/5
6680/6680 [==============================] - 34s 5ms/step - loss:
3.8246 - acc: 0.1373 - val_loss: 4.1565 - val_acc: 0.0862

Epoch 00004: val_loss improved from 4.21021 to 4.15648, saving mod
el to saved_models/weights.best.from_scratch.hdf5
Epoch 5/5
6680/6680 [==============================] - 34s 5ms/step - loss:
3.5898 - acc: 0.1728 - val_loss: 4.0893 - val_acc: 0.0970

Epoch 00005: val_loss improved from 4.15648 to 4.08929, saving mod
el to saved_models/weights.best.from_scratch.hdf5
```

Out[15]:

```
<keras.callbacks.History at 0x1277a1f28>
```

Load the Model with the Best Validation Loss

In [16]:

```
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 [17]:

```
# 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: 10.4067%
```

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 [18]:

```
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 [19]:

```
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_1 ( (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229
Trainable params: 68,229
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [20]:

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

Train the Model

In [21]:

```
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
6680/6680 [==============================] - 1s 219us/step - loss:
12.7366 - acc: 0.1046 - val_loss: 11.2111 - val_acc: 0.1808

Epoch 00001: val_loss improved from inf to 11.21108, saving model
to saved_models/weights.best.VGG16.hdf5
Epoch 2/20
6680/6680 [==============================] - 1s 135us/step - loss:
10.5786 - acc: 0.2515 - val_loss: 10.4928 - val_acc: 0.2575

Epoch 00002: val_loss improved from 11.21108 to 10.49285, saving m
odel to saved_models/weights.best.VGG16.hdf5
Epoch 3/20
```

```
6680/6680 [==============================] - 1s 135us/step - loss:
9.9552 - acc: 0.3138 - val_loss: 9.9257 - val_acc: 0.2898

Epoch 00003: val_loss improved from 10.49285 to 9.92574, saving mo
del to saved_models/weights.best.VGG16.hdf5
Epoch 4/20
6680/6680 [==============================] - 1s 136us/step - loss:
9.4041 - acc: 0.3596 - val_loss: 9.5504 - val_acc: 0.3186

Epoch 00004: val_loss improved from 9.92574 to 9.55036, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 5/20
6680/6680 [==============================] - 1s 135us/step - loss:
9.0338 - acc: 0.3886 - val_loss: 9.3344 - val_acc: 0.3281

Epoch 00005: val_loss improved from 9.55036 to 9.33439, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 6/20
6680/6680 [==============================] - 1s 136us/step - loss:
8.7852 - acc: 0.4145 - val_loss: 9.1846 - val_acc: 0.3569

Epoch 00006: val_loss improved from 9.33439 to 9.18465, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 7/20
6680/6680 [==============================] - 1s 135us/step - loss:
8.6111 - acc: 0.4278 - val_loss: 9.0342 - val_acc: 0.3593

Epoch 00007: val_loss improved from 9.18465 to 9.03417, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 8/20
6680/6680 [==============================] - 1s 136us/step - loss:
8.4471 - acc: 0.4479 - val_loss: 9.0141 - val_acc: 0.3629

Epoch 00008: val_loss improved from 9.03417 to 9.01411, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 9/20
6680/6680 [==============================] - 1s 137us/step - loss:
8.3929 - acc: 0.4587 - val_loss: 8.9673 - val_acc: 0.3737

Epoch 00009: val_loss improved from 9.01411 to 8.96735, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 10/20
6680/6680 [==============================] - 1s 136us/step - loss:
8.3500 - acc: 0.4627 - val_loss: 8.8882 - val_acc: 0.3737

Epoch 00010: val_loss improved from 8.96735 to 8.88820, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 11/20
6680/6680 [==============================] - 1s 135us/step - loss:
8.1116 - acc: 0.4722 - val_loss: 8.6653 - val_acc: 0.3880

Epoch 00011: val_loss improved from 8.88820 to 8.66533, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 12/20
6680/6680 [==============================] - 1s 140us/step - loss:
7.9236 - acc: 0.4894 - val_loss: 8.5031 - val_acc: 0.3940
```

```
Epoch 00012: val_loss improved from 8.66533 to 8.50312, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 13/20
6680/6680 [==============================] - 1s 135us/step - loss:
7.7420 - acc: 0.5004 - val_loss: 8.2861 - val_acc: 0.4072

Epoch 00013: val_loss improved from 8.50312 to 8.28611, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 14/20
6680/6680 [==============================] - 1s 136us/step - loss:
7.5907 - acc: 0.5124 - val_loss: 8.1952 - val_acc: 0.4204

Epoch 00014: val_loss improved from 8.28611 to 8.19515, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 15/20
6680/6680 [==============================] - 1s 136us/step - loss:
7.5136 - acc: 0.5208 - val_loss: 8.1737 - val_acc: 0.4180

Epoch 00015: val_loss improved from 8.19515 to 8.17368, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 16/20
6680/6680 [==============================] - 1s 132us/step - loss:
7.3991 - acc: 0.5226 - val_loss: 8.0451 - val_acc: 0.4120

Epoch 00016: val_loss improved from 8.17368 to 8.04515, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 17/20
6680/6680 [==============================] - 1s 132us/step - loss:
7.1639 - acc: 0.5410 - val_loss: 7.9250 - val_acc: 0.4371

Epoch 00017: val_loss improved from 8.04515 to 7.92501, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 18/20
6680/6680 [==============================] - 1s 132us/step - loss:
7.1187 - acc: 0.5496 - val_loss: 7.9622 - val_acc: 0.4251

Epoch 00018: val_loss did not improve from 7.92501
Epoch 19/20
6680/6680 [==============================] - 1s 133us/step - loss:
6.9798 - acc: 0.5475 - val_loss: 7.8176 - val_acc: 0.4359

Epoch 00019: val_loss improved from 7.92501 to 7.81765, saving mod
el to saved_models/weights.best.VGG16.hdf5
Epoch 20/20
6680/6680 [==============================] - 1s 139us/step - loss:
6.7506 - acc: 0.5663 - val_loss: 7.6048 - val_acc: 0.4407

Epoch 00020: val_loss improved from 7.81765 to 7.60483, saving mod
el to saved_models/weights.best.VGG16.hdf5
```

Out[21]:

```
<keras.callbacks.History at 0x127676390>
```

Load the Model with the Best Validation Loss

In [22]:

```
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 [23]:

```
# 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: 45.2153%
```

Predict Dog Breed with the Model

In [24]:

```
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)]
```

In [25]:

```
# dog name guessing test
test = VGG16_predict_breed('images/Labrador_retriever_06457.jpg')
print(test)
```

```
Labrador_retriever
```

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 [26]:

```
### TODO: Obtain bottleneck features from another pre-trained CNN.
bottleneck_features = np.load('bottleneck_features/DogInceptionV3Data.npz')
train_InceptionV3 = bottleneck_features['train']
valid_InceptionV3 = bottleneck_features['valid']
test_InceptionV3 = bottleneck_features['test']

print(train_InceptionV3.shape, valid_InceptionV3.shape, test_InceptionV3.shape
)
```

```
(6680, 5, 5, 2048) (835, 5, 5, 2048) (836, 5, 5, 2048)
```

(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: Based on [1], I implemented GlobalAveragePooling(GAP) after bottleneck_features. This GAP is used to reduce overfitting. After that, I add dropout layer as I experimented that using droptout(0.2) derives a little better Test Accuracy (80.7416% with dropout(0.2) VS 80.5024 % with dropout(0.3) and 78.9474% with no dropout). Then I applied softmax layer for classification.

- Note: Test Accuracy may vary slightly when new program run executed. However, Dropout(0.2) still shows better result than other methods.

[1] Min Lin, Qiang Chen, Shuicheng Yan, "Network In Network"

In [27]:

```
### TODO: Define your architecture.
InceptionV3_model = Sequential()

#InceptionV3_model.add(MaxPooling2D(pool_size=2, input_shape=train_InceptionV3
.shape[1:]))
InceptionV3_model.add(GlobalAveragePooling2D(input_shape=train_InceptionV3.sha
pe[1:]))
#model.add(Dense(2048))
#InceptionV3_model.add(Dropout(0.3))
InceptionV3_model.add(Dropout(0.2))
InceptionV3_model.add(Dense(133, activation='softmax'))

InceptionV3_model.summary()

# visualizing model
"""
from IPython.display import SVG
from keras.utils.vis_utils import model_to_dot

%matplotlib inline

SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
"""
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_2 ( (None, 2048)              0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 2048)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 133)               272517    
=================================================================
Total params: 272,517
Trainable params: 272,517
Non-trainable params: 0
_________________________________________________________________
```

Out[27]:

```
"\nfrom IPython.display import SVG\nfrom keras.utils.vis_utils imp
ort model_to_dot\n\n%matplotlib inline\n\nSVG(model_to_dot(model,
show_shapes=True).create(prog='dot', format='svg'))\n"
```

(IMPLEMENTATION) Compile the Model

In [28]:

```
### TODO: Compile the model.
InceptionV3_model.compile(loss='categorical_crossentropy', optimizer='rmsprop'
, 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 [29]:

```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.InceptionV3
.hdf5',
                               verbose=1, save_best_only=True)

InceptionV3_model.fit(train_InceptionV3, train_targets,
          validation_data=(valid_InceptionV3, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6680/6680 [==============================] - 4s 524us/step - loss:
1.2284 - acc: 0.6934 - val_loss: 0.6445 - val_acc: 0.8000

Epoch 00001: val_loss improved from inf to 0.64453, saving model t
o saved_models/weights.best.InceptionV3.hdf5
Epoch 2/20
6680/6680 [==============================] - 2s 237us/step - loss:
0.5036 - acc: 0.8464 - val_loss: 0.6403 - val_acc: 0.8287

Epoch 00002: val_loss improved from 0.64453 to 0.64028, saving mod
el to saved_models/weights.best.InceptionV3.hdf5
Epoch 3/20
6680/6680 [==============================] - 2s 236us/step - loss:
0.4163 - acc: 0.8763 - val_loss: 0.6371 - val_acc: 0.8467

Epoch 00003: val_loss improved from 0.64028 to 0.63707, saving mod
el to saved_models/weights.best.InceptionV3.hdf5
Epoch 4/20
6680/6680 [==============================] - 2s 234us/step - loss:
0.3535 - acc: 0.8940 - val_loss: 0.6919 - val_acc: 0.8395

Epoch 00004: val_loss did not improve from 0.63707
Epoch 5/20
6680/6680 [==============================] - 2s 238us/step - loss:
0.3134 - acc: 0.9112 - val_loss: 0.7254 - val_acc: 0.8443

Epoch 00005: val_loss did not improve from 0.63707
Epoch 6/20
6680/6680 [==============================] - 1s 222us/step - loss:
0.2767 - acc: 0.9174 - val_loss: 0.6916 - val_acc: 0.8551

Epoch 00006: val_loss did not improve from 0.63707
Epoch 7/20
6680/6680 [==============================] - 1s 222us/step - loss:
```

```
0.2425 - acc: 0.9272 - val_loss: 0.7125 - val_acc: 0.8527

Epoch 00007: val_loss did not improve from 0.63707
Epoch 8/20
6680/6680 [==============================] - 1s 221us/step - loss:
0.2287 - acc: 0.9349 - val_loss: 0.7283 - val_acc: 0.8575

Epoch 00008: val_loss did not improve from 0.63707
Epoch 9/20
6680/6680 [==============================] - 1s 221us/step - loss:
0.2069 - acc: 0.9406 - val_loss: 0.7399 - val_acc: 0.8443

Epoch 00009: val_loss did not improve from 0.63707
Epoch 10/20
6680/6680 [==============================] - 1s 224us/step - loss:
0.1823 - acc: 0.9455 - val_loss: 0.7564 - val_acc: 0.8563

Epoch 00010: val_loss did not improve from 0.63707
Epoch 11/20
6680/6680 [==============================] - 2s 229us/step - loss:
0.1662 - acc: 0.9534 - val_loss: 0.8012 - val_acc: 0.8575

Epoch 00011: val_loss did not improve from 0.63707
Epoch 12/20
6680/6680 [==============================] - 2s 233us/step - loss:
0.1610 - acc: 0.9525 - val_loss: 0.7862 - val_acc: 0.8503

Epoch 00012: val_loss did not improve from 0.63707
Epoch 13/20
6680/6680 [==============================] - 2s 235us/step - loss:
0.1424 - acc: 0.9588 - val_loss: 0.8088 - val_acc: 0.8491

Epoch 00013: val_loss did not improve from 0.63707
Epoch 14/20
6680/6680 [==============================] - 2s 227us/step - loss:
0.1345 - acc: 0.9644 - val_loss: 0.8540 - val_acc: 0.8419

Epoch 00014: val_loss did not improve from 0.63707
Epoch 15/20
6680/6680 [==============================] - 2s 227us/step - loss:
0.1280 - acc: 0.9618 - val_loss: 0.8879 - val_acc: 0.8539

Epoch 00015: val_loss did not improve from 0.63707
Epoch 16/20
6680/6680 [==============================] - 2s 226us/step - loss:
0.1163 - acc: 0.9639 - val_loss: 0.8632 - val_acc: 0.8491

Epoch 00016: val_loss did not improve from 0.63707
Epoch 17/20
6680/6680 [==============================] - 2s 229us/step - loss:
0.1094 - acc: 0.9674 - val_loss: 0.8808 - val_acc: 0.8527

Epoch 00017: val_loss did not improve from 0.63707
Epoch 18/20
6680/6680 [==============================] - 2s 229us/step - loss:
0.1076 - acc: 0.9690 - val_loss: 0.8415 - val_acc: 0.8647
```

```
Epoch 00018: val_loss did not improve from 0.63707
Epoch 19/20
6680/6680 [==============================] - 2s 227us/step - loss: 0.0930 - acc: 0.9738 - val_loss: 0.9066 - val_acc: 0.8515

Epoch 00019: val_loss did not improve from 0.63707
Epoch 20/20
6680/6680 [==============================] - 2s 238us/step - loss: 0.0983 - acc: 0.9722 - val_loss: 0.8894 - val_acc: 0.8551

Epoch 00020: val_loss did not improve from 0.63707
```

Out[29]:

```
<keras.callbacks.History at 0x1289beba8>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [30]:

```
### TODO: Load the model weights with the best validation loss.
InceptionV3_model.load_weights('saved_models/weights.best.InceptionV3.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 [31]:

```
### TODO: Calculate classification accuracy on the test dataset.
# get index of predicted dog breed for each image in test set
InceptionV3_predictions = [np.argmax(InceptionV3_model.predict(np.expand_dims(feature, axis=0))) for feature in test_InceptionV3]

# report test accuracy
test_accuracy = 100*np.sum(np.array(InceptionV3_predictions)==np.argmax(test_targets, axis=1))/len(InceptionV3_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 80.1435%
```

(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 [32]:

```
### 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 InceptionV3_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_InceptionV3(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = InceptionV3_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

In [33]:

```
# dog name guessing test
test = InceptionV3_predict_breed('images/Labrador_retriever_06457.jpg')
print(test)
```

```
Labrador_retriever
```

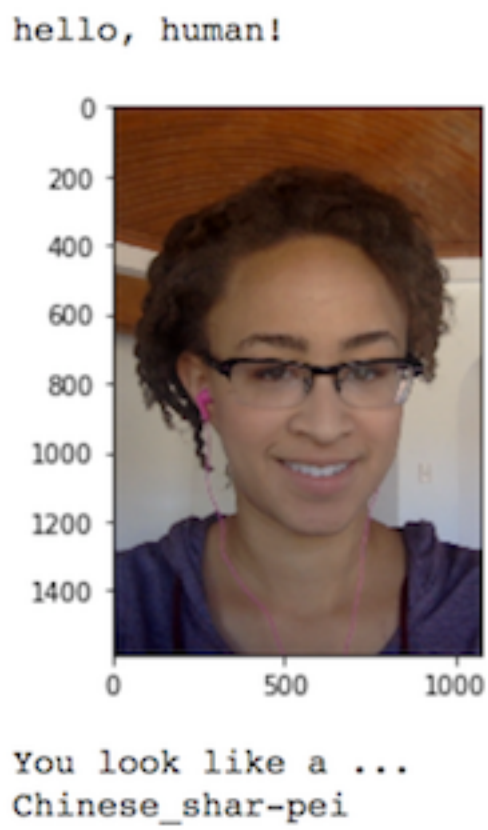
Step 6: Write your Algorithm

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.

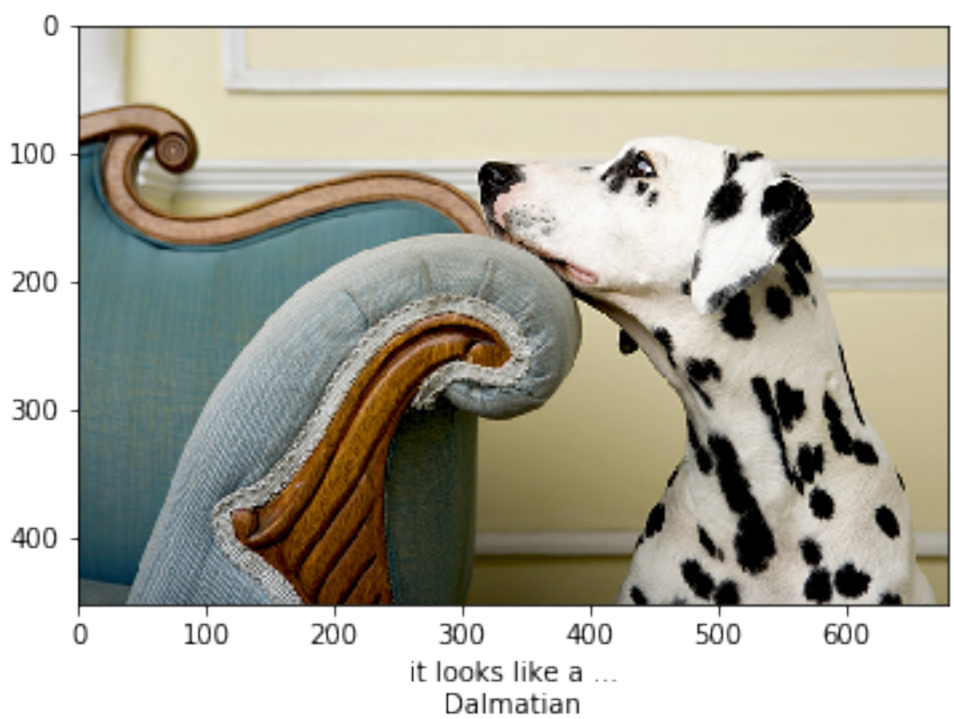
Some sample output for our algorithm is provided below, but feel free to design your own user experience!



(IMPLEMENTATION) Write your Algorithm

In [34]:

```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
###
img_path = 'images/dalmatian_3.jpg'
# load the image
img = cv2.imread(img_path)
# convert BGR image to RGB
img_RGB = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# add subplot
fig = plt.figure()
ax = fig.add_subplot(111)

dog_name = InceptionV3_predict_breed(img_path)
# classify image category
if face_detector(img_path):
    ax.set_title('Hello, human!')
    # dog name guessing test
    ax.set_xlabel('and you look like a ' + dog_name)
else:
    if dog_detector(img_path):
        ax.set_xlabel('it looks like a ...\n'+ dog_name)
    else:
        ax.set_xlabel('ERROR: neither human nor dog')

# convert BGR image to RGB for plotting
cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
# display the image
plt.imshow(cv_rgb)
plt.show()
```



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!

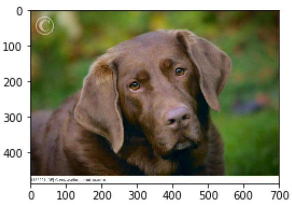
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

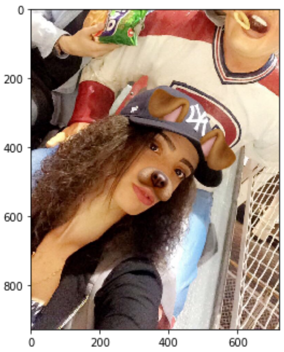
Answer: I am surprised with the result because the algorithm can distinguish human and dog really well. However, there are couple of drawbacks with this training. First, it seems that it can not recognize human well if there are more than 2 people(2nd row 1st image). Although 2nd row 4th image was said to be a 'cavalier king charles spaniel', I have no idea who is similar with this dog breed. I think the main cause is because the training data are consisted of only one person. So computer has no idea how to deal with people. Second, algorithm can not recogize a human if he/she wears a cap or hat(1st row 3rd image). I guess I need to teach computer that human being can wear cap or hat depends on their mood and it is still a human being. Third, algorithm shows weak point to augmented images (e.g not fully showing or tilted image). For example, I tested with a girl with cap image which is cut a little at right top(1st row 3rd image). However, this model could not recognize if it is a human being. Also, human being with a special effect can not be recognized as a human (1st row 2nd image). In my opinion, training data should be reinforced with augmented data. If an image were a little bit tilted, computer would not have recognized if it is a human being either. Lastly, I was surprised that this model can distinguish between wolf and dog. 2nd row 3rd image is a wolf and the result confirmed that it is not a dog nor human being (of course it can't be said that it is a wolf either). Well trained CNN with good training data is an absolutely amazing!

In [35]:

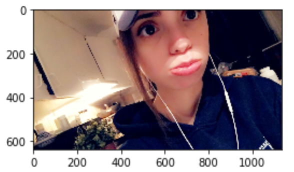
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

# set subplot size in inches (W, H)
fig = plt.figure(figsize= (24,18))
for i in range(10):
    img_path = 'images/sample/sample%d.jpg'%i

    # load images
    img = cv2.imread(img_path)
    # convert BGR image to RGB for plotting
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

    # set subplot axis as 2 by 5 matrix.
    ax = fig.add_subplot(2,5,i+1)
    ax.imshow(cv_rgb)

    # predict the doggy breed.
    dog_name = InceptionV3_predict_breed(img_path)
    # classify image category
    if face_detector(img_path):
        ax.set_title('Hello, human!')
        # dog name guessing test
        ax.set_xlabel('and you look like a ' + dog_name)
    else:
        if dog_detector(img_path):
            ax.set_xlabel('it looks like a ...\n'+ dog_name)
        else:
            ax.set_xlabel('ERROR: neither human nor dog')

# display the image
plt.show()
```



it looks like a ...
Labrador_retriever

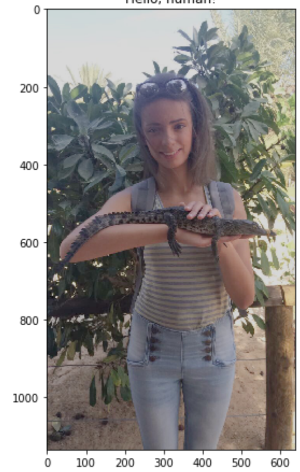


ERROR: neither human nor dog

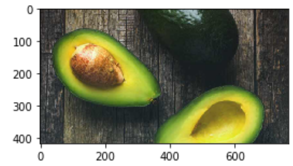


ERROR: neither human nor dog

Hello, human!

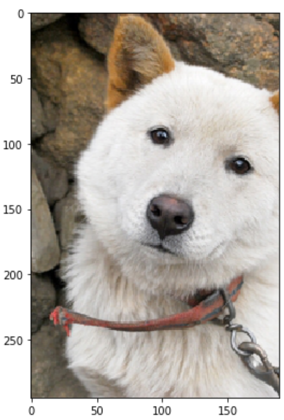


and you look like a Greyhound

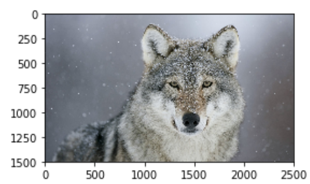


ERROR: neither human nor dog

ERROR: neither human nor dog

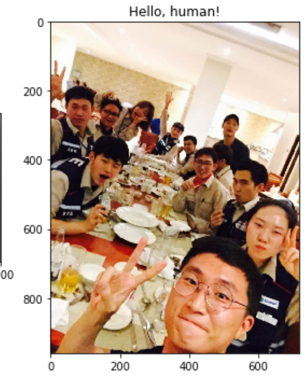


it looks like a ...
Akita



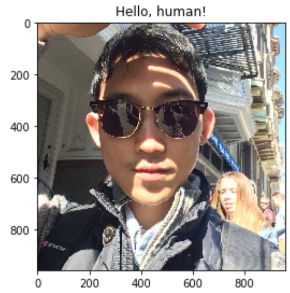
ERROR: neither human nor dog

Hello, human!



and you look like a Cavalier_king_charles_spaniel

Hello, human!



and you look like a Cardigan_welsh_corgi