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.

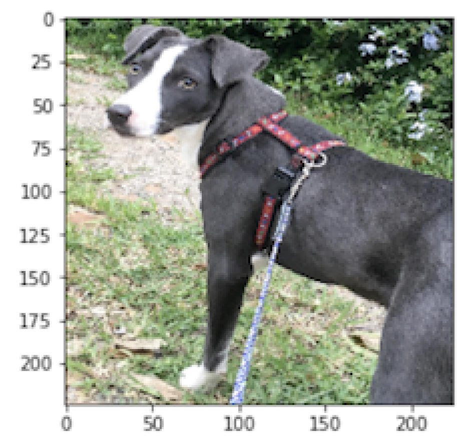
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

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_files, 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 13270 total human images.
```

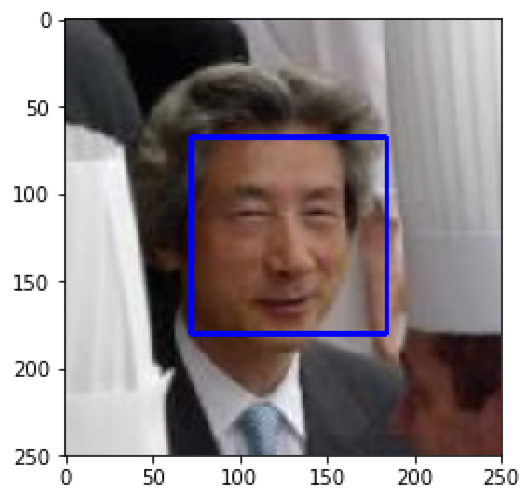
Step 1: Detect Humans

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: 98% from `human_files` have a deteced human face. 11% of `dog_files` have a detected face.

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.

hh = 0  # human detected as human
dh = 0  # dog detected as human
for id in range(0,100):
    if face_detector(human_files_short[id]):
        hh += 1
    if face_detector(dog_files_short[id]):
        dh += 1
print(str(hh) + "% of the first 100 images in human_files have a detected human face")
print(str(dh) + "% of the first 100 images in dog_files have a detected human face")
```

```
98% of the first 100 images in human_files have a detected human face
11% of the first 100 images in dog_files have a detected human face
```

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.

The earlier face detector focuses on detecting human face. It'll be very user-unfriendly if asking user to provide a picture with clear face. Thus, for any picture failed in `face_detector`, the algorithm will continue on `human_detector`. `HOGDescriptor` from OpenCV is used here. However, there is no improvement on this particular project. The accuracy is exactly the same as `face_detector`. I tried to pinpoint to the 2 human image failed in `face_detector` and found there are some objects on the picture disctracting the face recognition. It fails in `face_detector` is not because of the picture doesn't have clear face.

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.

# The earlier face detector was to detect human faces. Thus
def human_detector(img_path):
    if face_detector(img_path):
        return True
    else:
        img = cv2.imread(img_path)
        #plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        hog = cv2.HOGDescriptor()
        hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())
        humen = hog.detectMultiScale(gray)
        #print(humen)
        #print(type(humen[0])==np.ndarray)

hh = 0  # human detected as human
dh = 0  # dog detected as human
for id in range(0,100):
    if human_detector(human_files_short[id]):
        hh += 1
    if human_detector(dog_files_short[id]):
        dh += 1
print(str(hh) + "% of the first 100 images in human_files have a detected human")
print(str(dh) + "% of the first 100 images in dog_files have a detected human")
```

```
98% of the first 100 images in human_files have a detected human
11% of the first 100 images in dog_files have a detected human
```

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: 1% from `human_files` have a deteced dog. 100% of `dog_files` have a dog.

In [11]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
hd = 0  # dog detected as dog
dd = 0  # human detected as dog
for id in range(0,100):
    if dog_detector(human_files_short[id]):
        hd += 1
    if dog_detector(dog_files_short[id]):
        dd += 1
print(str(hd) + "% of the first 100 images in human_files have a detected dog")
print(str(dd) + "% of the first 100 images in dog_files have a detected dog")
```

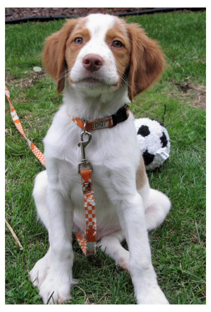
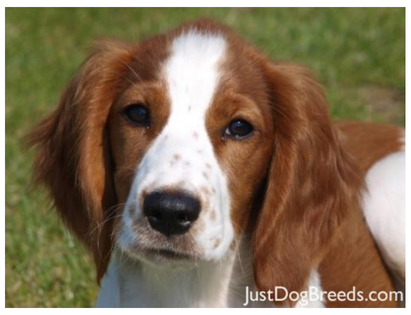
```
1% of the first 100 images in human_files have a detected dog
100% of the first 100 images in dog_files have a detected dog
```

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

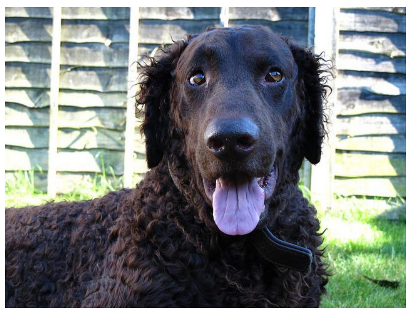
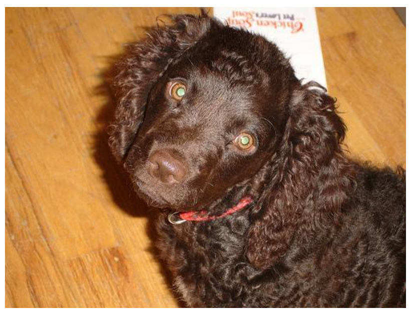
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

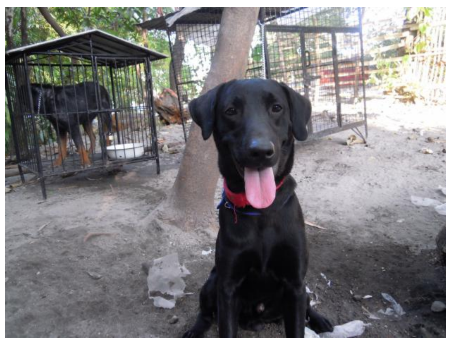
We mention that the task of assigning breed to dogs from images is considered exceptionally challenging. To see why, consider that *even a human* would have great difficulty in distinguishing between a Brittany and a Welsh Springer Spaniel.

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]:

```python
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
```

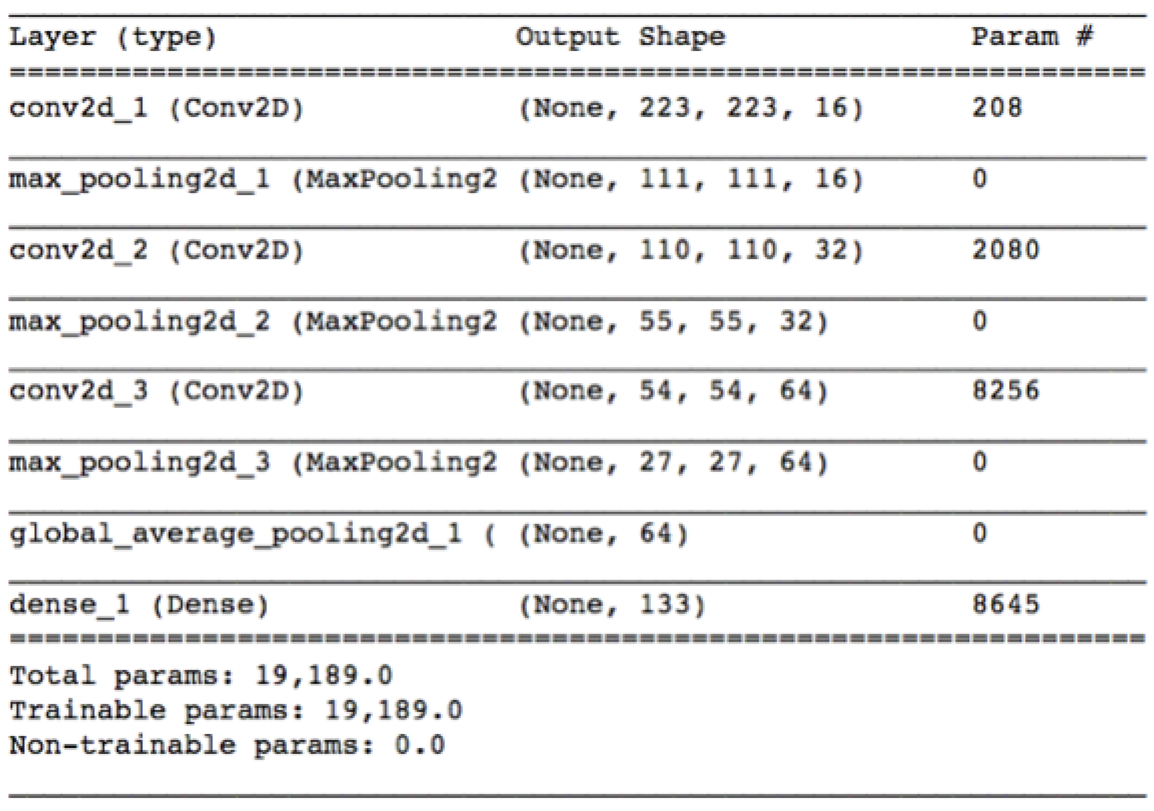
```
100%|██████████| 6680/6680 [02:14<00:00, 49.78it/s]
100%|██████████| 835/835 [00:15<00:00, 61.52it/s]
100%|██████████| 836/836 [00:15<00:00, 55.45it/s]
```

(IMPLEMENTATION) Model Architecture

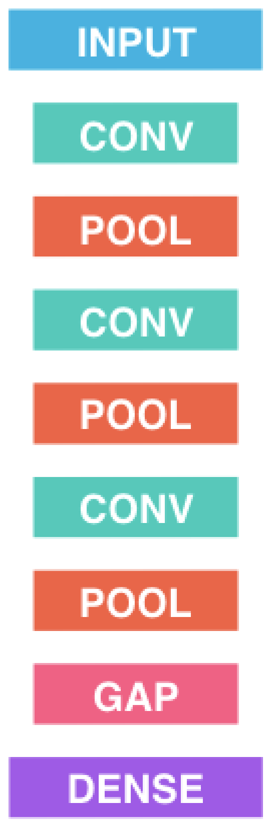
Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
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: The architecture of the hinted model looks great. I'll just build my model based on it, except I will include more layers and filters. The main reason is I'm also feeling the difficulty on classifying dog breeds. And my intuition tells me that CNN needs more features from the picture to achieve a better accuracy.

The first and the last layer of the CNN needs not much effort to explain as the input and output dimensions are fixed. After several rounds of trying, I set the hidden layers to include 3 convolutional layers and 5 pooling layers. One epoch takes about 35s when running on g2.2xlarge instance from AWS. The test accuracy fluctuates between 4% and 7%.

In [13]:
```
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(Conv2D(filters=128, kernel_size=2, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(GlobalAveragePooling2D())
model.add(Dense(133, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
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         
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 28, 28, 128)       32896     
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 14, 14, 128)       0         
_________________________________________________________________
global_average_pooling2d_1 ( (None, 128)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 133)               17157     
=================================================================
Total params: 60,597.0
Trainable params: 60,597.0
Non-trainable params: 0.0
_________________________________________________________________
```

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.

# 10 epochs used here
epochs = 10

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

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.from_scratch.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 1/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8843 - acc: 0.0072      □
improved from inf to 4.86946, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 226s - loss: 4.8844 - acc: 0.0072 - val_loss: 4.8695 - val_acc:
0.0108
Epoch 2/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.8501 - acc: 0.0125      □
improved from 4.86946 to 4.82789, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 332s - loss: 4.8500 - acc: 0.0124 - val_loss: 4.8279 - val_acc:
0.0132
Epoch 3/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.7730 - acc: 0.0174  □
roved from 4.82789 to 4.76801, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 240s - loss: 4.7731 - acc: 0.0174 - val_loss: 4.7680 - val_acc:
 0.0228
Epoch 4/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.7068 - acc: 0.0249      □
 improved from 4.76801 to 4.71624, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 218s - loss: 4.7067 - acc: 0.0249 - val_loss: 4.7162 - val_acc:
 0.0287
Epoch 5/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.6531 - acc: 0.0332  Epoch 00004: val_loss imp
roved from 4.71624 to 4.69320, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 218s - loss: 4.6532 - acc: 0.0332 - val_loss: 4.6932 - val_acc:
 0.0275
Epoch 6/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5994 - acc: 0.0380      □
 improved from 4.69320 to 4.62558, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 219s - loss: 4.6002 - acc: 0.0379 - val_loss: 4.6256 - val_acc:
 0.0395
Epoch 7/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.5325 - acc: 0.0494  □
roved from 4.62558 to 4.57556, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 218s - loss: 4.5329 - acc: 0.0494 - val_loss: 4.5756 - val_acc:
 0.0419
Epoch 8/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.4579 - acc: 0.0506  □
roved from 4.57556 to 4.48544, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 221s - loss: 4.4573 - acc: 0.0507 - val_loss: 4.4854 - val_acc:
 0.0515
Epoch 9/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.3874 - acc: 0.0608      □
 improved from 4.48544 to 4.44342, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 221s - loss: 4.3873 - acc: 0.0608 - val_loss: 4.4434 - val_acc:
 0.0527
Epoch 10/10
6660/6680 [============================>.] - ETA: 0s - loss: 4.3182 - acc: 0.0674      Epoch 00009: val_loss
 improved from 4.44342 to 4.36121, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 235s - loss: 4.3174 - acc: 0.0674 - val_loss: 4.3612 - val_acc:
 0.0599
```

Out[15]:
```
<keras.callbacks.History at 0x12a7f2860>
```

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_targets, axis=1))/len(dog_breed_pr
edictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 6.3397%
```

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

VGG16_model.summary()
```

```
(6680, 7, 7, 512)
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_2 ( (None, 512)               0         
_________________________________________________________________
dense_2 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [20]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['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
6580/6680 [============================>.] - ETA: 0s - loss: 12.5586 - acc: 0.1111  Epoch 00000: val_loss im
proved from inf to 10.87219, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 12.5313 - acc: 0.1129 - val_loss: 10.8722 - val_acc:
0.2072
Epoch 2/20
6600/6680 [============================>.] - ETA: 0s - loss: 10.2240 - acc: 0.2779Epoch 00001: val_loss impr
oved from 10.87219 to 10.21891, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.2004 - acc: 0.2793 - val_loss: 10.2189 - val_acc:
0.2814
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 9.7387 - acc: 0.3364 Epoch 00002: val_loss impr
oved from 10.21891 to 10.11384, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.7335 - acc: 0.3365 - val_loss: 10.1138 - val_acc:
 0.2958
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 9.4780 - acc: 0.3638Epoch 00003: val_loss impro
ved from 10.11384 to 9.89917, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.4816 - acc: 0.3636 - val_loss: 9.8992 - val_acc:
 0.3078
Epoch 5/20
6420/6680 [===========================>..] - ETA: 0s - loss: 9.2192 - acc: 0.3942Epoch 00004: val_loss impro
ved from 9.89917 to 9.67080, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.2400 - acc: 0.3927 - val_loss: 9.6708 - val_acc:
 0.3401
Epoch 6/20
6640/6680 [============================>.] - ETA: 0s - loss: 9.1662 - acc: 0.4066 Epoch 00005: val_loss impr
oved from 9.67080 to 9.63821, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1568 - acc: 0.4069 - val_loss: 9.6382 - val_acc:
 0.3305
Epoch 7/20
6560/6680 [============================>.] - ETA: 0s - loss: 8.9627 - acc: 0.4162Epoch 00006: val_loss impro
ved from 9.63821 to 9.34059, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.9757 - acc: 0.4154 - val_loss: 9.3406 - val_acc:
 0.3485
Epoch 8/20
6620/6680 [============================>.] - ETA: 0s - loss: 8.8265 - acc: 0.4290Epoch 00007: val_loss impro
ved from 9.34059 to 9.27874, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.8179 - acc: 0.4290 - val_loss: 9.2787 - val_acc:
 0.3653
Epoch 9/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.6210 - acc: 0.4348Epoch 00008: val_loss impro
ved from 9.27874 to 8.93720, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5990 - acc: 0.4361 - val_loss: 8.9372 - val_acc:
 0.3737
Epoch 10/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.3766 - acc: 0.4585Epoch 00009: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 8.3677 - acc: 0.4597 - val_loss: 8.9471 - val_acc:
 0.3808
Epoch 11/20
6580/6680 [============================>.] - ETA: 0s - loss: 8.2618 - acc: 0.4723Epoch 00010: val_loss impro
ved from 8.93720 to 8.78080, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.2784 - acc: 0.4714 - val_loss: 8.7808 - val_acc:
 0.3904
Epoch 12/20
6520/6680 [============================>.] - ETA: 0s - loss: 8.0372 - acc: 0.4756Epoch 00011: val_loss impro
ved from 8.78080 to 8.55705, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0433 - acc: 0.4754 - val_loss: 8.5571 - val_acc:
 0.4012
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.8561 - acc: 0.4932Epoch 00012: val_loss impro
ved from 8.55705 to 8.52710, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.8620 - acc: 0.4928 - val_loss: 8.5271 - val_acc:
 0.3940
Epoch 14/20
6480/6680 [============================>.] - ETA: 0s - loss: 7.8208 - acc: 0.4991Epoch 00013: val_loss impro
ved from 8.52710 to 8.44602, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.8088 - acc: 0.4997 - val_loss: 8.4460 - val_acc:
 0.4180
Epoch 15/20
6480/6680 [============================>.] - ETA: 0s - loss: 7.7466 - acc: 0.5069Epoch 00014: val_loss impro
ved from 8.44602 to 8.31616, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.7251 - acc: 0.5084 - val_loss: 8.3162 - val_acc:
 0.4132
Epoch 16/20
6480/6680 [============================>.] - ETA: 0s - loss: 7.6508 - acc: 0.5139Epoch 00015: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 7.6623 - acc: 0.5130 - val_loss: 8.3657 - val_acc:
 0.4168
Epoch 17/20
6620/6680 [============================>.] - ETA: 0s - loss: 7.6270 - acc: 0.5168Epoch 00016: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 7.6254 - acc: 0.5166 - val_loss: 8.3618 - val_acc:
 0.4108
Epoch 18/20
6660/6680 [============================>.] - ETA: 0s - loss: 7.4943 - acc: 0.5212Epoch 00017: val_loss impro
```

```
ved from 8.31616 to 8.13656, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.4936 - acc: 0.5213 - val_loss: 8.1366 - val_acc:
 0.4180
Epoch 19/20
6600/6680 [============================>.] - ETA: 0s - loss: 7.4410 - acc: 0.5279Epoch 00018: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 7.4318 - acc: 0.5284 - val_loss: 8.1665 - val_acc:
 0.4323
Epoch 20/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.4199 - acc: 0.5312Epoch 00019: val_loss impro
ved from 8.13656 to 8.10904, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.4148 - acc: 0.5317 - val_loss: 8.1090 - val_acc:
 0.4323
```

Out[21]:
```
<keras.callbacks.History at 0x12ceeee48>
```

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, axis=0))) for feature in test_VGG1
6]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(test_targets, axis=1))/len(VGG16_prediction
s)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 43.5407%
```

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)]
```

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 [25]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.

# Below code is almost the same as the VGG16 example. And in this project I choose ResNet50.
bottleneck_features = np.load('bottleneck_features/DogResnet50Data.npz')
train_Resnet50 = bottleneck_features['train']
valid_Resnet50 = bottleneck_features['valid']
test_Resnet50 = 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: In the transfer learning we can use a pre-built model to train a customized model must faster. I started to try with just adding two new layers and found the performance is good enough. The first layer is an avarage pooling layer. There are 2048 filters from previous layer and that's a lot. The last layer is a fully connected layer with output dimension of 133, needlessly to explain why it's 133. And the test accuracy is around 80%.

In [26]:
```
### TODO: Define your architecture.
Resnet50_model = Sequential()
# Print the dimension of model.
print(train_Resnet50.shape)
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[1:]))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
(6680, 1, 1, 2048)
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0         
_________________________________________________________________
dense_3 (Dense)              (None, 133)               272517    
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [27]:
```
### TODO: Compile the model.
Resnet50_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 [28]:
```
### TODO: Train the model.
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.Resnet50.hdf5', 
                               verbose=1, save_best_only=True)

Resnet50_model.fit(train_Resnet50, train_targets, 
          validation_data=(valid_Resnet50, valid_targets),
          epochs=20, batch_size=20, callbacks=[checkpointer], verbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.6445 - acc: 0.6011      Epoch 00000: val_loss
improved from inf to 0.81065, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 2s - loss: 1.6292 - acc: 0.6039 - val_loss: 0.8107 - val_acc:
 0.7509
Epoch 2/20
6500/6680 [============================>.] - ETA: 0s - loss: 0.4422 - acc: 0.8603Epoch 00001: val_loss impro
ved from 0.81065 to 0.65303, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.4369 - acc: 0.8623 - val_loss: 0.6530 - val_acc:
 0.7976
Epoch 3/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.2588 - acc: 0.9183Epoch 00002: val_loss impro
ved from 0.65303 to 0.65176, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.2588 - acc: 0.9183 - val_loss: 0.6518 - val_acc:
 0.8012
Epoch 4/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.1753 - acc: 0.9456Epoch 00003: val_loss impro
ved from 0.65176 to 0.62659, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.1750 - acc: 0.9455 - val_loss: 0.6266 - val_acc:
 0.8132
Epoch 5/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.1223 - acc: 0.9617Epoch 00004: val_loss impro
ved from 0.62659 to 0.61862, saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 0.1211 - acc: 0.9621 - val_loss: 0.6186 - val_acc:
 0.8287
Epoch 6/20
6660/6680 [============================>.] - ETA: 0s - loss: 0.0876 - acc: 0.9727Epoch 00005: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0876 - acc: 0.9726 - val_loss: 0.6868 - val_acc:
 0.8371
Epoch 7/20
6640/6680 [============================>.] - ETA: 0s - loss: 0.0627 - acc: 0.9807Epoch 00006: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0625 - acc: 0.9807 - val_loss: 0.6496 - val_acc:
 0.8275
Epoch 8/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0490 - acc: 0.9861Epoch 00007: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0490 - acc: 0.9861 - val_loss: 0.6984 - val_acc:
 0.8275
Epoch 9/20
6460/6680 [============================>.] - ETA: 0s - loss: 0.0358 - acc: 0.9898Epoch 00008: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0367 - acc: 0.9895 - val_loss: 0.7208 - val_acc:
 0.8311
Epoch 10/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0268 - acc: 0.9932Epoch 00009: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0267 - acc: 0.9933 - val_loss: 0.7487 - val_acc:
 0.8251
Epoch 11/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0206 - acc: 0.9949    Epoch 00010: val_loss d
id not improve
6680/6680 [==============================] - 1s - loss: 0.0204 - acc: 0.9949 - val_loss: 0.7474 - val_acc:
 0.8228
Epoch 12/20
6440/6680 [===========================>..] - ETA: 0s - loss: 0.0158 - acc: 0.9961Epoch 00011: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0168 - acc: 0.9960 - val_loss: 0.8037 - val_acc:
 0.8180
Epoch 13/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0138 - acc: 0.9962    Epoch 00012: val_loss d
id not improve
6680/6680 [==============================] - 1s - loss: 0.0143 - acc: 0.9960 - val_loss: 0.8159 - val_acc:
 0.8228
Epoch 14/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0109 - acc: 0.9970Epoch 00013: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0108 - acc: 0.9970 - val_loss: 0.8636 - val_acc:
 0.8108
Epoch 15/20
6460/6680 [============================>.] - ETA: 0s - loss: 0.0099 - acc: 0.9977Epoch 00014: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0102 - acc: 0.9975 - val_loss: 0.8452 - val_acc:
 0.8216
Epoch 16/20
6520/6680 [============================>.] - ETA: 0s - loss: 0.0081 - acc: 0.9983Epoch 00015: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0081 - acc: 0.9982 - val_loss: 0.8595 - val_acc:
 0.8156
Epoch 17/20
6440/6680 [===========================>..] - ETA: 0s - loss: 0.0052 - acc: 0.9988Epoch 00016: val_loss did n
ot improve
6680/6680 [==============================] - 1s - loss: 0.0056 - acc: 0.9987 - val_loss: 0.8760 - val_acc:
 0.8395
Epoch 18/20
6460/6680 [============================>.] - ETA: 0s - loss: 0.0067 - acc: 0.9980    Epoch 00017: val_loss d
```

```
id not improve
6680/6680 [==============================] - 1s - loss: 0.0066 - acc: 0.9981 - val_loss: 0.8896 - val_acc:
 0.8275
Epoch 19/20
6440/6680 [===========================>..] - ETA: 0s - loss: 0.0060 - acc: 0.9980    Epoch 00018: val_loss d
id not improve
6680/6680 [==============================] - 1s - loss: 0.0059 - acc: 0.9981 - val_loss: 0.9147 - val_acc:
 0.8216
Epoch 20/20
6620/6680 [============================>.] - ETA: 0s - loss: 0.0048 - acc: 0.9985    Epoch 00019: val_loss d
id not improve
6680/6680 [==============================] - 1s - loss: 0.0048 - acc: 0.9985 - val_loss: 0.9141 - val_acc:
 0.8359
```

```
Out[28]: <keras.callbacks.History at 0x120e60d30>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

```
In [29]: ### TODO: Load the model weights with the best validation loss.
         Resnet50_model.load_weights('saved_models/weights.best.Resnet50.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 [30]: ### TODO: Calculate classification accuracy on the test dataset.

         # get index of predicted dog breed for each image in test set
         Resnet50_predictions = [np.argmax(Resnet50_model.predict(np.expand_dims(feature, axis=0))) for feature in tes
         t_Resnet50]

         # report test accuracy
         test_accuracy = 100*np.sum(np.array(Resnet50_predictions)==np.argmax(test_targets, axis=1))/len(Resnet50_pred
         ictions)
         print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 81.4593%
```

(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 [31]: ### 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 Resnet50_predict_breed(img_path):
             # extract bottleneck features
             bottleneck_feature = extract_Resnet50(path_to_tensor(img_path))
             # obtain predicted vector
             predicted_vector = Resnet50_model.predict(bottleneck_feature)
             # return dog breed that is predicted by the model
             return dog_names[np.argmax(predicted_vector)]
```

Step 6: Write your Algorithm

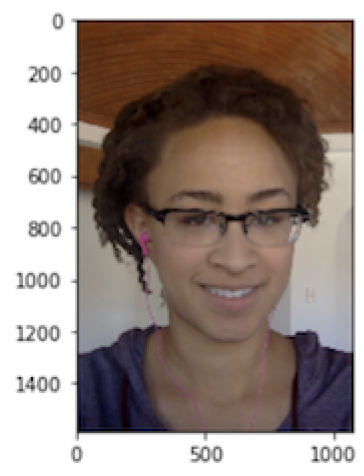
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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 `human_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!
```

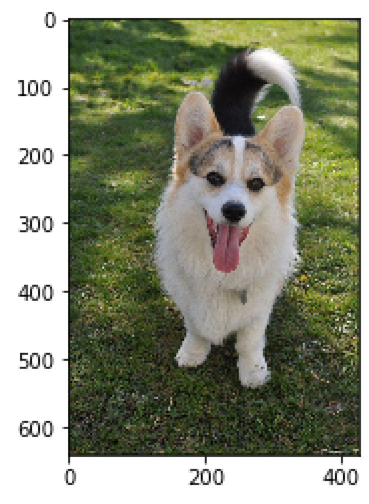


```
You look like a ...
Chinese_shar-pei
```

(IMPLEMENTATION) Write your Algorithm

In [32]:
```python
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.

# hord means "human or dog", input is the file address, output is the result of test.
def hord(testImage):
    # Read the image for later use.
    img = cv2.imread(testImage)
    plt.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    if dog_detector(testImage):
        #print("Dog detected.")
        dogBreed = Resnet50_predict_breed(testImage)
        #print("The predicted dog breed is", dogBreed)
        response = 'Dog detected.\nThe predicted dog breed is ' + dogBreed + '.'
        return response
    elif face_detector(testImage):
        #print("Human face detected")
        dogBreed = Resnet50_predict_breed(testImage)
        #print("And you look like a", dogBreed)
        response = 'Human face detected.\nAnd you look like a ' + dogBreed + '.'
        return response
    else:
        #print("I don't know who/what you are.")
        reponse = "I don't know who/what you are."
        return reponse

testImage = train_files[100]
print(hord(testImage))
```

```
Dog detected.
The predicted dog breed is Cardigan_welsh_corgi.
```



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.

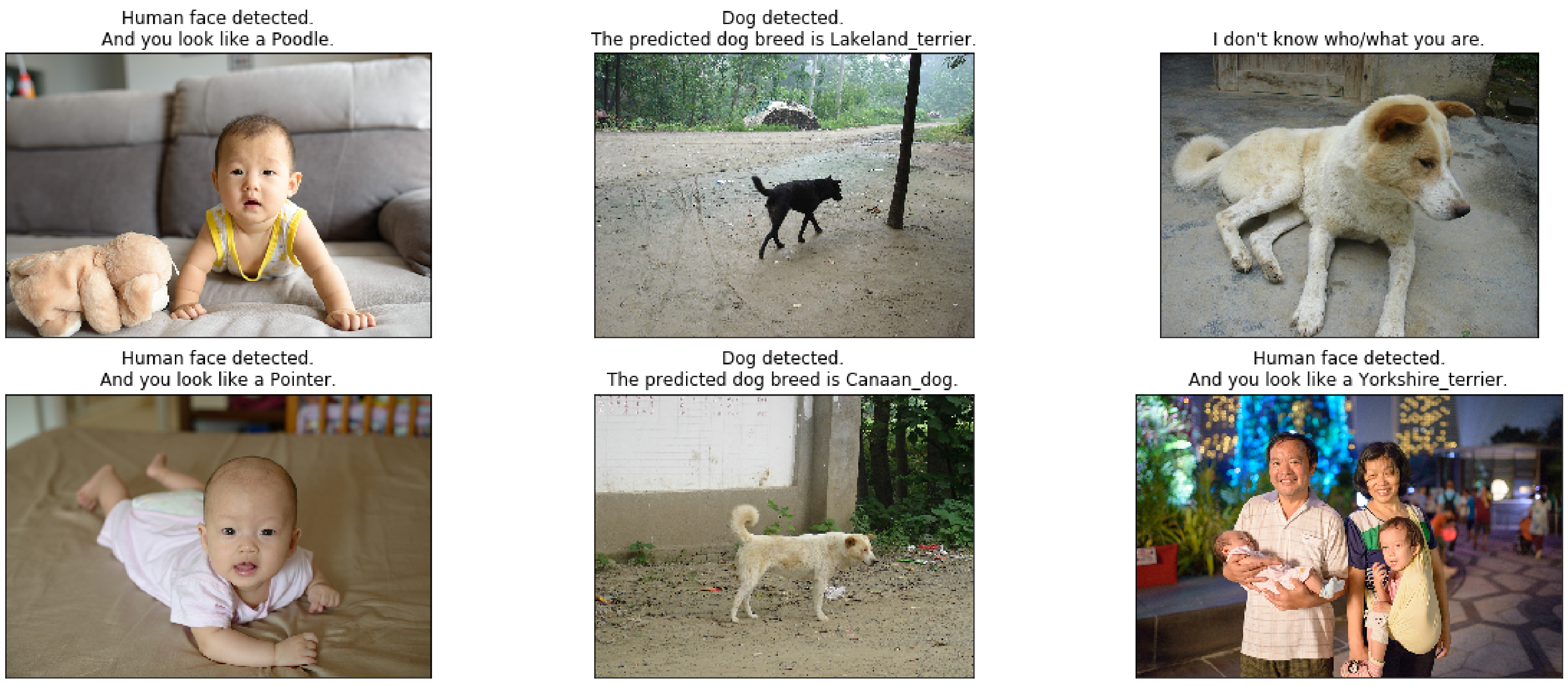
Question 6: Is the output better than you expected :) ? Or worse :(? Provide at least three possible points of improvement for your algorithm.

Answer: The output works like a charm, despite it failed to classify one picture correctly. Maybe a positive result will be produced if my dog is facing to the camera.

The result is based on one model with two layers added for transfer learning. So a better result should be achieved if below points are implemented.

1. Utilizing multiple models, besides just one ResNet50. It's relatively cheap as we use transfer learning here. Then we can output based on the most voted result.
2. Adding more layers instead of just 2. Again, it's relatively cheap as transfer learning is used here. In the current implementation, a global average pooling layer is added after the pre-built model, maybe many useful features are still waiting to be digged out.
3. There are some dog images are not detected as dogs. Some human images are also not detected as human. A further study on Type I and Type II error when running each dectetor algorithm could also help improve the final result.

In [33]:
```python
## 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.
mydata = load_files('myimages')
myfiles = np.array(mydata['filenames'])

fig = plt.figure(figsize=(20, 8))
for i in range(len(myfiles)):
    img = cv2.imread(myfiles[i])
    ax = fig.add_subplot(2, 3, i + 1, xticks=[], yticks=[])
    ax.imshow(cv2.cvtColor(img, cv2.COLOR_BGR2RGB))
    ax.set_title("{}".format(hord(myfiles[i])))
```


Human face detected.
And you look like a Poodle.
Dog detected.
The predicted dog breed is Lakeland_terrier.
I don't know who/what you are.
Human face detected.
And you look like a Pointer.
Dog detected.
The predicted dog breed is Canaan_dog.
Human face detected.
And you look like a Yorkshire_terrier.