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.

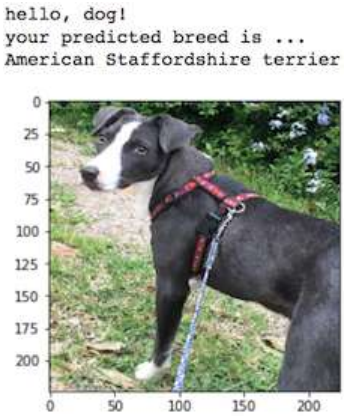
In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes 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!).



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_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 13233 total human images.
```

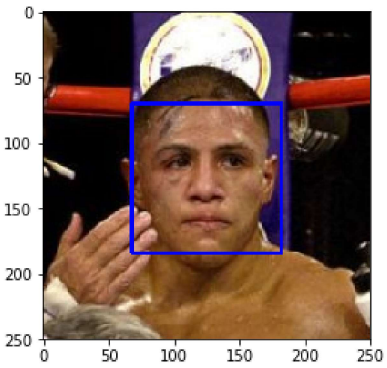
Step 1: Detect Humans

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 [5]:
```
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 [6]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

(IMPLEMENTATION) Assess the Human Face Detector

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

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

Ideally, we would like 100% of human images with a detected face and 0% of dog images with a detected face. You will see that our algorithm falls short of this goal, but still gives acceptable performance. We extract the file paths for the first 100 images from each of the datasets and store them in the numpy arrays `human_files_short` and `dog_files_short`.

Answer:

In [7]:
```
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.

## DONE!
def num_of_faces(img_paths):
    humans = 0
    for img in img_paths:
        if face_detector(img): humans += 1
    return humans

print('Human files: {}% human faces detected.'.format(num_of_faces(human_files_short)))
print('Dog files: {}% human faces detected'.format(num_of_faces(dog_files_short)))
```

```
Human files: 98% human faces detected.
Dog files: 11% human faces detected
```

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:

As long as the software utilizing the algorithm only promises its users that it can only accurately classify clearly visible human faces, it is a reasonable expectation. However if there is a need for software that can detect faces that might be not be 100% visible, this is not a reasonable expectation. Due to limited nature of the Haar-Cascade algorithm being used by OpenCV, requiring a mostly frontal face in order for it to properly detect one, this expectation is required for this current app.

Unless the developer is willing to go through the relatively long process of building a set of classifiers for various different profiles of faces, with varying level of visibility, there is another known method. This method is the usage of deep convolutional neural networks (aka CNNs) as discussed in class. These CNNs have the capabilities to classify many different type of faces, from many different angles and is able to learn this all on it's own, with relatively shorter training times, as long as enough training data is provided.

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 [8]:
```
## (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 [9]:
```
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 [10]:
```
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 [11]:
```
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 [12]:
```
### returns "True" if a dog is detected in the image stored at img_path
def dog_detector(img_path):
    prediction = ResNet50_predict_labels(img_path)
    return ((prediction <= 268) & (prediction >= 151))
```

(IMPLEMENTATION) Assess the Dog Detector

Question 3: Use the code cell below to test the performance of your `dog_detector` function.

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

Answer:

In [13]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

### DONE!
def num_of_dog_faces(img_paths):
    humans = 0
    for img in img_paths:
        if dog_detector(img): humans += 1
    return humans

print('Human files: {}% dogs detected.'.format(num_of_dog_faces(human_files_short)))
print('Dog files: {}% dogs detected'.format(num_of_dog_faces(dog_files_short)))
```

```
Human files: 1% dogs detected.
Dog files: 100% dogs detected
```

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

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

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

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

Brittany	Welsh Springer Spaniel
	

It is not difficult to find other dog breed pairs with minimal inter-class variation (for instance, Curly-Coated Retrievers and American Water Spaniels).

Curly-Coated Retriever	American Water Spaniel
	

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

Yellow Labrador	Chocolate Labrador	Black Labrador

Yellow Labrador	Chocolate Labrador	Black Labrador
		

We also mention that random chance presents an exceptionally low bar: setting aside the fact that the classes are slightly imabalanced, a random guess will provide a correct answer roughly 1 in 133 times, which corresponds to an accuracy of less than 1%.

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different architectures, and trust your intuition. And, of course, have fun!

Pre-process the Data

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

In [14]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True

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

```
100%|██████████| 6680/6680 [00:50<00:00, 131.81it/s]
100%|██████████| 835/835 [00:05<00:00, 145.90it/s]
100%|██████████| 836/836 [00:05<00:00, 147.54it/s]
```

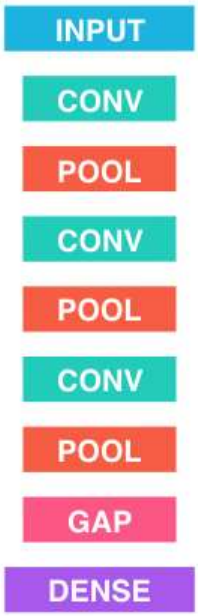
(IMPLEMENTATION) Model Architecture

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 that I chose is very similar to VGGNet, just being a little smaller in scale. Starting with the input size of the architecture, the chosen sequential network has 2 stacks of 2 Conv2D layers and 1 MaxPooling2D layer and then is topped off by a single GlobalAveragePooling2D layer, a Dropout layer and a Dense layer. The reasoning used to choose this architecture is explained below.

For all Conv2D layers in the network, a number of parameter remains fixed. The chosen values are a 3x3 kernel size, 1x1 stride, same padding and an RELU activation function. These choices proved to test best, which makes sense for using a too large kernel size and a larger stride could prove to be lossy of the original input image. It was also shown in testing, that stacking 2 Conv layers the same parameters together performs better in the general case. The RELU activation function was chosen in order to minimize the effects of gradient vanishing when training. The only parameter that changed is the number of filters, increasing from 32 to 64 after every pooling layer. Keeping the number of filters the same in each Conv2D layer pair proved to perform the best during testing and keeping the overall number of filters small helped with keeping training time low.

The type of pooling layers chosen in the network is the MaxPooling2D (MP) layer and the GlobalAveragePooling2D (GAP) layer. Every MP layer has a pool size and stride of 2x2 and same padding. This choice has the provided benefit of reducing the feature shape by half and providing a level of translation invariance in the input. The GAP layer was chosen for papers show that GAP layers not only help reduce dimensionality of the features down a 1x1xN tensor, where N is the number of channels, but also helps with object localization without too much overhead.

At the very end of the network, a single Dropout layer and a Dense, or Fully Connected (FC), layer is utilized. The Dropout layer is set to a rate of 50%, a number which proved to perform consistently well during validation testing, is used to prevent overfitting on the training data. The last FC layer with the Softmax activation function is utilized to shape the output into 133 classes.

In [15]:
```
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.

### DONE!
model.add(Conv2D(filters=32, kernel_size=(3, 3), padding='same', activation='relu',
                 input_shape=train_tensors.shape[1:]))
model.add(Conv2D(filters=32, kernel_size=(3, 3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))

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

model.add(GlobalAveragePooling2D())
model.add(Dropout(rate=0.5))
model.add(Dense(units=133, activation='softmax'))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 32)      896
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 224, 224, 32)      9248
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 64)      18496
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 112, 112, 64)      36928
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 74,213
Trainable params: 74,213
Non-trainable params: 0
_________________________________________________________________
```

Compile the Model

In [16]:
```
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.

### DONE.
epochs = 5

### 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 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8877 - acc: 0.0089Epoch 00000: val_loss improved
from inf to 4.88102, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 76s - loss: 4.8878 - acc: 0.0088 - val_loss: 4.8810 - val_acc: 0.0
108
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8769 - acc: 0.0104Epoch 00001: val_loss improved
from 4.88102 to 4.87140, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 75s - loss: 4.8769 - acc: 0.0103 - val_loss: 4.8714 - val_acc: 0.0
108
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8705 - acc: 0.0119Epoch 00002: val_loss improved
from 4.87140 to 4.86373, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 76s - loss: 4.8703 - acc: 0.0120 - val_loss: 4.8637 - val_acc: 0.0
168
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8538 - acc: 0.0140Epoch 00003: val_loss improved
from 4.86373 to 4.84561, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 76s - loss: 4.8537 - acc: 0.0141 - val_loss: 4.8456 - val_acc: 0.0
228
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.8246 - acc: 0.0245Epoch 00004: val_loss improved
from 4.84561 to 4.80506, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 76s - loss: 4.8244 - acc: 0.0244 - val_loss: 4.8051 - val_acc: 0.0
228
```

Out[15]:
```
<keras.callbacks.History at 0x7f2e00079710>
```

Load the Model with the Best Validation Loss

In [17]:
```
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 [18]:
```
# 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_predic
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 1.9139%
```

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 [19]:
```
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 [20]:
```
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_2 ( (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 [21]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

Train the Model

In [21]:
```
from keras.callbacks import ModelCheckpoint

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
6560/6680 [============================>.] - ETA: 0s - loss: 12.5812 - acc: 0.1163Epoch 00000: val_loss imp
roved from inf to 11.33340, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 12.5677 - acc: 0.1163 - val_loss: 11.3334 - val_ac
c: 0.1856
Epoch 2/20
6480/6680 [============================>.] - ETA: 0s - loss: 10.4674 - acc: 0.2602Epoch 00001: val_loss imp
roved from 11.33340 to 10.45707, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.4663 - acc: 0.2608 - val_loss: 10.4571 - val_ac
c: 0.2647
Epoch 3/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.9204 - acc: 0.3265Epoch 00002: val_loss impr
oved from 10.45707 to 10.22513, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.9381 - acc: 0.3256 - val_loss: 10.2251 - val_acc:
0.2922
Epoch 4/20
6500/6680 [============================>.] - ETA: 0s - loss: 9.5773 - acc: 0.3600Epoch 00003: val_loss impr
oved from 10.22513 to 9.91603, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.5902 - acc: 0.3597 - val_loss: 9.9160 - val_acc:
0.3210
Epoch 5/20
6460/6680 [============================>.] - ETA: 0s - loss: 9.2973 - acc: 0.3865Epoch 00004: val_loss impr
oved from 9.91603 to 9.77970, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.2914 - acc: 0.3873 - val_loss: 9.7797 - val_acc:
0.3090
Epoch 6/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.9761 - acc: 0.4112Epoch 00005: val_loss impr
oved from 9.77970 to 9.42077, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.9919 - acc: 0.4102 - val_loss: 9.4208 - val_acc:
0.3365
Epoch 7/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.8036 - acc: 0.4294Epoch 00006: val_loss impr
oved from 9.42077 to 9.37855, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7953 - acc: 0.4296 - val_loss: 9.3785 - val_acc:
0.3449
Epoch 8/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.7493 - acc: 0.4408Epoch 00007: val_loss impr
oved from 9.37855 to 9.37224, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7439 - acc: 0.4410 - val_loss: 9.3722 - val_acc:
0.3557
Epoch 9/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.7418 - acc: 0.4463Epoch 00008: val_loss impr
oved from 9.37224 to 9.29816, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7128 - acc: 0.4475 - val_loss: 9.2982 - val_acc:
0.3629
Epoch 10/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.5670 - acc: 0.4537Epoch 00009: val_loss impr
oved from 9.29816 to 9.12352, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.5614 - acc: 0.4539 - val_loss: 9.1235 - val_acc:
0.3629
Epoch 11/20
6440/6680 [============================>..] - ETA: 0s - loss: 8.3363 - acc: 0.4688Epoch 00010: val_loss impr
oved from 9.12352 to 9.03474, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.3442 - acc: 0.4686 - val_loss: 9.0347 - val_acc:
0.3808
Epoch 12/20
6440/6680 [============================>..] - ETA: 0s - loss: 8.3004 - acc: 0.4776Epoch 00011: val_loss impr
oved from 9.03474 to 9.00125, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.3047 - acc: 0.4774 - val_loss: 9.0012 - val_acc:
0.3808
Epoch 13/20
6440/6680 [============================>..] - ETA: 0s - loss: 8.3015 - acc: 0.4807Epoch 00012: val_loss impr
oved from 9.00125 to 8.95973, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.2982 - acc: 0.4808 - val_loss: 8.9597 - val_acc:
0.3868
Epoch 14/20
```

```
6480/6680 [============================>.] - ETA: 0s - loss: 8.3421 - acc: 0.4792Epoch 00013: val_loss did
not improve
6680/6680 [==============================] - 1s - loss: 8.2855 - acc: 0.4828 - val_loss: 8.9844 - val_acc:
0.3868
Epoch 15/20
6440/6680 [===========================>..] - ETA: 0s - loss: 8.3018 - acc: 0.4831Epoch 00014: val_loss impr
oved from 8.95973 to 8.94020, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.2792 - acc: 0.4843 - val_loss: 8.9402 - val_acc:
0.3892
Epoch 16/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.2806 - acc: 0.4823Epoch 00015: val_loss did
not improve
6680/6680 [==============================] - 1s - loss: 8.2397 - acc: 0.4847 - val_loss: 8.9622 - val_acc:
0.3892
Epoch 17/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.1493 - acc: 0.4857Epoch 00016: val_loss impr
oved from 8.94020 to 8.82670, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.1536 - acc: 0.4855 - val_loss: 8.8267 - val_acc:
0.3928
Epoch 18/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.0895 - acc: 0.4940Epoch 00017: val_loss impr
oved from 8.82670 to 8.82132, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0839 - acc: 0.4945 - val_loss: 8.8213 - val_acc:
0.4000
Epoch 19/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.0853 - acc: 0.4958Epoch 00018: val_loss did
not improve
6680/6680 [==============================] - 1s - loss: 8.0769 - acc: 0.4963 - val_loss: 8.8748 - val_acc:
0.3928
Epoch 20/20
6500/6680 [============================>.] - ETA: 0s - loss: 8.0470 - acc: 0.4938Epoch 00019: val_loss impr
oved from 8.82132 to 8.76304, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0410 - acc: 0.4942 - val_loss: 8.7630 - val_acc:
0.4024
```

Out[21]: <keras.callbacks.History at 0x7f2df87e77b8>

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_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: 42.9426%
```

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.

### DONE!
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:

The chosen architecture takes the bottleneck features of an already trained Resnet50 model as an input and just proceeds to feed them into a single GAP layer, a dropout layer and a final FC layer with the softmax activation function for the final classification. There are number of reasons why this relatively simple architecture was chosen:

- The Resnet50 already does a great job, and adding more convolutional/pooling layers did not result any addtional accuracy in testing. It did, however, add more training time and a longer prediction time as well.
- The GAP layer is included to reduce the dimensionality of the features and to help with object localization.
- The dropout layer with a rate of 50% is used to reduce overfitting of the classification layer.
- The last layer uses the softmax activation which is pretty standard for multiclass classification problems.

In [26]:
```
### TODO: Define your architecture.

### DONE!
Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[1:]))
Resnet50_model.add(Dropout(rate=0.5))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0
_________________________________________________________________
dropout_2 (Dropout)          (None, 2048)              0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               272517
=================================================================
Total params: 272,517
Trainable params: 272,517
Non-trainable params: 0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [27]:
```
### TODO: Compile the model.

### DONE!
Resnet50_model.compile(loss='categorical_crossentropy', optimizer='adam', 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.

### DONE!
from keras.callbacks import ModelCheckpoint
from keras.preprocessing.image import ImageDataGenerator

datagen_train = ImageDataGenerator(
    rotation_range=40,
    width_shift_range=0.2,  # randomly shift images horizontally (10% of total width)
    height_shift_range=0.2,  # randomly shift images vertically (10% of total height)
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True) # randomly flip images horizontally

datagen_train.fit(train_Resnet50)

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.RESNET50.hdf5',
                               verbose=1, save_best_only=True)
num_epochs = 20
batch_size = 20

Resnet50_model.fit_generator(datagen_train.flow(train_Resnet50, train_targets, batch_size=batch_size),
                             epochs=num_epochs, steps_per_epoch=train_Resnet50.shape[0] // batch_size,
                             validation_data=(valid_Resnet50, valid_targets),
                             validation_steps=valid_Resnet50.shape[0] // batch_size,
                             callbacks=[checkpointer], verbose=1)
```

```
/usr/local/lib/python3.5/dist-packages/Keras-2.0.8-py3.5.egg/keras/preprocessing/image.py:653: UserWarning: Exp
ected input to be images (as Numpy array) following the data format convention "channels_last" (channels on axi
s 3), i.e. expected either 1, 3 or 4 channels on axis 3. However, it was passed an array with shape (6680, 1,
1, 2048) (2048 channels).
/usr/local/lib/python3.5/dist-packages/Keras-2.0.8-py3.5.egg/keras/preprocessing/image.py:787: UserWarning: Num
pyArrayIterator is set to use the data format convention "channels_last" (channels on axis 3), i.e. expected ei
ther 1, 3 or 4 channels on axis 3. However, it was passed an array with shape (6680, 1, 1, 2048) (2048 channel
s).

Epoch 1/20
333/334 [============================>.] - ETA: 0s - loss: 2.3528 - acc: 0.4503Epoch 00000: val_loss improv
ed from inf to 0.86818, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 284s - loss: 2.3492 - acc: 0.4509 - val_loss: 0.8682 - val_acc:
0.7557
Epoch 2/20
333/334 [============================>.] - ETA: 0s - loss: 0.7610 - acc: 0.7677Epoch 00001: val_loss improv
ed from 0.86818 to 0.75065, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 303s - loss: 0.7604 - acc: 0.7681 - val_loss: 0.7507 - val_acc:
0.7725
Epoch 3/20
333/334 [============================>.] - ETA: 0s - loss: 0.4658 - acc: 0.8527Epoch 00002: val_loss improv
ed from 0.75065 to 0.67977, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 329s - loss: 0.4667 - acc: 0.8522 - val_loss: 0.6798 - val_acc:
0.7808
Epoch 4/20
333/334 [============================>.] - ETA: 0s - loss: 0.3469 - acc: 0.8857Epoch 00003: val_loss improv
ed from 0.67977 to 0.64736, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 311s - loss: 0.3471 - acc: 0.8858 - val_loss: 0.6474 - val_acc:
0.7940
Epoch 5/20
333/334 [============================>.] - ETA: 0s - loss: 0.2769 - acc: 0.9075Epoch 00004: val_loss did no
t improve
334/334 [==============================] - 307s - loss: 0.2769 - acc: 0.9075 - val_loss: 0.6627 - val_acc:
0.8036
Epoch 6/20
333/334 [============================>.] - ETA: 0s - loss: 0.2152 - acc: 0.9338Epoch 00005: val_loss did no
t improve
334/334 [==============================] - 312s - loss: 0.2149 - acc: 0.9340 - val_loss: 0.6710 - val_acc:
0.8024
Epoch 7/20
333/334 [============================>.] - ETA: 0s - loss: 0.1811 - acc: 0.9423Epoch 00006: val_loss did no
t improve
334/334 [==============================] - 286s - loss: 0.1810 - acc: 0.9425 - val_loss: 0.6781 - val_acc:
0.7928
Epoch 8/20
333/334 [============================>.] - ETA: 0s - loss: 0.1619 - acc: 0.9503Epoch 00007: val_loss improv
ed from 0.64736 to 0.64647, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 306s - loss: 0.1626 - acc: 0.9500 - val_loss: 0.6465 - val_acc:
```

```
0.8132
Epoch 9/20
333/334 [============================>.] - ETA: 0s - loss: 0.1424 - acc: 0.9544Epoch 00008: val_loss improv
ed from 0.64647 to 0.62643, saving model to saved_models/weights.best.RESNET50.hdf5
334/334 [==============================] - 325s - loss: 0.1425 - acc: 0.9543 - val_loss: 0.6264 - val_acc:
0.8216
Epoch 10/20
333/334 [============================>.] - ETA: 0s - loss: 0.1245 - acc: 0.9608Epoch 00009: val_loss did no
t improve
334/334 [==============================] - 287s - loss: 0.1248 - acc: 0.9606 - val_loss: 0.6967 - val_acc:
0.8108
Epoch 11/20
333/334 [============================>.] - ETA: 1s - loss: 0.1157 - acc: 0.9643Epoch 00010: val_loss did no
t improve
334/334 [==============================] - 338s - loss: 0.1155 - acc: 0.9644 - val_loss: 0.6688 - val_acc:
0.8192
Epoch 12/20
333/334 [============================>.] - ETA: 0s - loss: 0.1202 - acc: 0.9614Epoch 00011: val_loss did no
t improve
334/334 [==============================] - 287s - loss: 0.1202 - acc: 0.9615 - val_loss: 0.7087 - val_acc:
0.8024
Epoch 13/20
333/334 [============================>.] - ETA: 0s - loss: 0.1096 - acc: 0.9658Epoch 00012: val_loss did no
t improve
334/334 [==============================] - 308s - loss: 0.1097 - acc: 0.9656 - val_loss: 0.7050 - val_acc:
0.7976
Epoch 14/20
333/334 [============================>.] - ETA: 0s - loss: 0.1010 - acc: 0.9686Epoch 00013: val_loss did no
t improve
334/334 [==============================] - 295s - loss: 0.1014 - acc: 0.9686 - val_loss: 0.7148 - val_acc:
0.8072
Epoch 15/20
333/334 [============================>.] - ETA: 0s - loss: 0.1007 - acc: 0.9653Epoch 00014: val_loss did no
t improve
334/334 [==============================] - 297s - loss: 0.1011 - acc: 0.9653 - val_loss: 0.7524 - val_acc:
0.8108
Epoch 16/20
333/334 [============================>.] - ETA: 0s - loss: 0.0912 - acc: 0.9715Epoch 00015: val_loss did no
t improve
334/334 [==============================] - 299s - loss: 0.0911 - acc: 0.9716 - val_loss: 0.7523 - val_acc:
0.8192
Epoch 17/20
333/334 [============================>.] - ETA: 0s - loss: 0.0825 - acc: 0.9730Epoch 00016: val_loss did no
t improve
334/334 [==============================] - 302s - loss: 0.0828 - acc: 0.9729 - val_loss: 0.7772 - val_acc:
0.8024
Epoch 18/20
333/334 [============================>.] - ETA: 0s - loss: 0.0856 - acc: 0.9733Epoch 00017: val_loss did no
t improve
334/334 [==============================] - 291s - loss: 0.0854 - acc: 0.9734 - val_loss: 0.8355 - val_acc:
0.7988
Epoch 19/20
333/334 [============================>.] - ETA: 0s - loss: 0.0871 - acc: 0.9712Epoch 00018: val_loss did no
t improve
334/334 [==============================] - 301s - loss: 0.0869 - acc: 0.9713 - val_loss: 0.8322 - val_acc:
0.7988
Epoch 20/20
333/334 [============================>.] - ETA: 0s - loss: 0.0800 - acc: 0.9722Epoch 00019: val_loss did no
t improve
334/334 [==============================] - 296s - loss: 0.0803 - acc: 0.9719 - val_loss: 0.8587 - val_acc:
0.7904
```

Out[28]: <keras.callbacks.History at 0x7f2db44a33c8>

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [28]:
```
### TODO: Load the model weights with the best validation loss.

### DONE!
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 [29]:
```
### TODO: Calculate classification accuracy on the test dataset.

### DONE!
Resnet50_predictions = [np.argmax(Resnet50_model.predict(np.expand_dims(feature, axis=0))) for feature in test_Re

# report test accuracy
test_accuracy = 100*np.sum(np.array(Resnet50_predictions)==np.argmax(test_targets, axis=1))/len(Resnet50_predicti
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 [30]:
```
### TODO: Write a function that takes a path to an image as input
### and returns the dog breed that is predicted by the model.

### DONE!
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

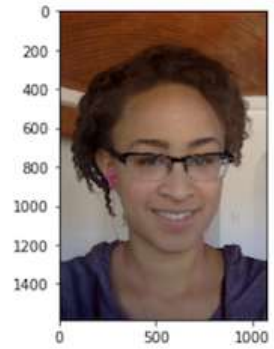
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- 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 `dog_detector` functions developed above. You are **required** to use your CNN from Step 5 to predict dog breed.

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

```
hello, human!
```




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

(IMPLEMENTATION) Write your Algorithm

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

### DONE!
import re

def plot_image(img_path):
    img = cv2.imread(img_path)
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.imshow(cv_rgb)
    plt.show()

def human_dog_detector(img_path):
    '''Determines if provied image is a human or a dog, and closest dog breed.
    '''
    species = ''

    # The order of detectors is really important here.
    # Turns out the face detector is less reliable than
    # the dog detector.
    if dog_detector(img_path):
        species = 'dog'
    elif face_detector(img_path):
        species = 'human'
    else:
        print("You're not a human nor a dog.....WHAT ARE YOU!?")
        plot_image(img_path)
        return

    breed = RESNET50_predict_breed(img_path)

    print('Hello {}!\nYou look like a {}.' \
      .format(species, " ".join(re.findall(r'[A-Za-z]+', breed))))
    plot_image(img_path)
```

Step 7: Test Your Algorithm

In this section, you will take your new algorithm for a spin! What kind of dog does the algorithm think that **you** look like? If you have a dog, does it predict your dog's breed accurately? If you have a cat, does it mistakenly think that your cat is a dog?

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

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

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

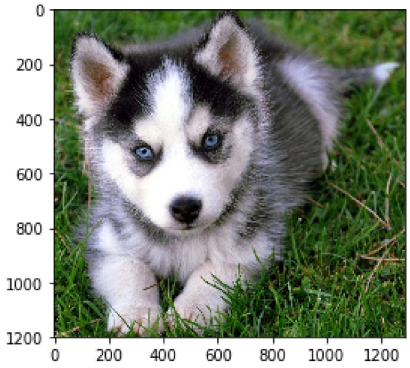
Answer:

I believe the algorithm perform pretty well. Of the 7 provided images, the algorithm was able to successfully distinguish between humans and non-humans for all them and for the most part get at least very close estimates of the breed. There are of course a couple of ways the algorithm could be improved:

- More training data could help the algorithm learn the more subtle differences between certain types of breed. Currently, the algorithm isn't able to tell the difference between similar breeds. For example, it currently detects a Yorkshire Terrier as a Silky Terrier and a Pug as Bulldog sometimes.
- "We need to go deeper." Inception quotes asside, a cnn, especially a variant of the ResNet architecture which is more suited for training deeper networks, has a huge capacity to improve performance with more layers.
- It could have been worth trying other optimizers to get a better training result. Previously the submitted algorithm used the RMSprop optimizer, but now it uses as the Adam optimizer. This switch proved to be more performant. It's possible that a different optimizer might add another small bump in accuracy.

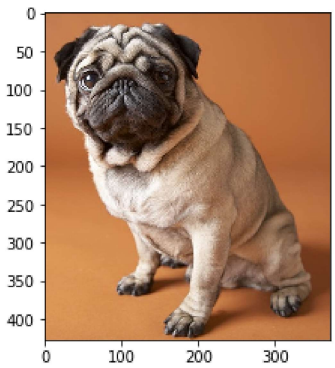
In [36]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

### 1st DONE!

img_path = 'images/step6_test/husky1-1.jpg'

human_dog_detector(img_path)
```

```
Hello dog!
You look like a Alaskan malamute.
```



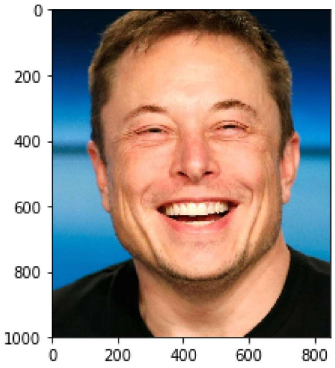
In [37]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

### 2nd DONE!

img_path = 'images/step6_test/pug1-1.jpg'

human_dog_detector(img_path)
```

```
Hello dog!
You look like a Bulldog.
```



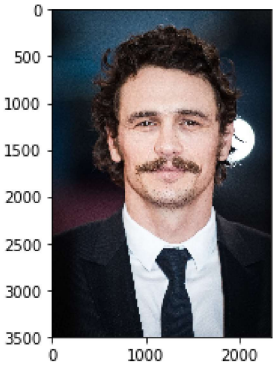
In [38]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

### 3rd DONE!

img_path = 'images/step6_test/elon1-2.jpg'

human_dog_detector(img_path)
```

```
Hello human!
You look like a Chesapeake bay retriever.
```



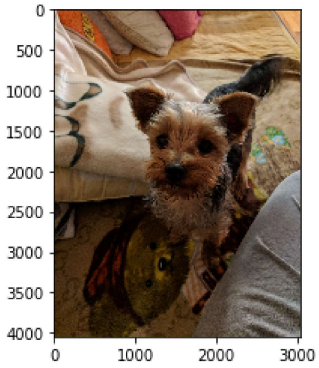
In [39]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

### 4th DONE!

img_path = 'images/step6_test/franco1-1.jpg'

human_dog_detector(img_path)
```

```
Hello human!
You look like a American foxhound.
```



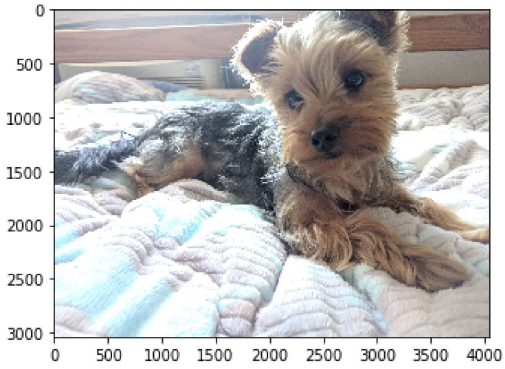
In [40]:
```
## TODO: Execute your algorithm from Step 6 on
## at least 6 images on your computer.
## Feel free to use as many code cells as needed.

### 5th DONE!

img_path = 'images/step6_test/yorkie1-1.jpg'

human_dog_detector(img_path)
```

```
Hello dog!
You look like a Yorkshire terrier.
```



In [41]:
```
## 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.

### 6th DONE!

img_path = 'images/step6_test/yorkie1-2.jpg'

human_dog_detector(img_path)
```

```
Hello dog!
You look like a Silky terrier.
```



In [42]:
```
## 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.

### 7th DONE!

# This should fail.
img_path = 'images/step6_test/banana.jpg'

human_dog_detector(img_path)
```

```
You're not a human nor a dog.....WHAT ARE YOU!?
```