In [43]:

```
import sys
print(sys.version)

import keras
print(keras.__version__)

from tensorflow.python.client import device_lib
device_lib.list_local_devices()
```

```
3.6.1 |Continuum Analytics, Inc.| (default, May 11 2017, 13:09:58)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)]
2.0.2
```

Out[43]:

```
[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 7662533488137307469, name: "/device:GPU:0"
 device_type: "GPU"
 memory_limit: 15591945012
 locality {
   bus_id: 1
   links {
   }
 }
 incarnation: 3740655472811408007
 physical_device_desc: "device: 0, name: Tesla V100-SXM2-16GB, pci b
us id: 0000:00:1e.0, compute capability: 7.0"]
```

Artificial Intelligence Nanodegree

Convolutional Neural Networks

Project: Write an Algorithm for a Dog Identification App

In this notebook, some template code has already been provided for you, and you will need to implement additional functionality to successfully complete this project. You will not need to modify the included code beyond what is requested. Sections that begin with **'(IMPLEMENTATION)'** in the header indicate that the following block of code will require additional functionality which you must provide. Instructions will be provided for each section, and the specifics of the implementation are marked in the code block with a 'TODO' statement. Please be sure to read the instructions carefully!

> **Note**: Once you have completed all of the code implementations, you need to finalize your work by exporting the iPython Notebook as an HTML document. Before exporting the notebook to html, all of the code cells need to have been run so that reviewers can see the final implementation and output. You can then export the notebook by using the menu above and navigating to \n", "**File -> Download as -> HTML (.html)**. Include the finished document along with this notebook as your submission.

In addition to implementing code, there will be questions that you must answer which relate to the project and your implementation. Each section where you will answer a question is preceded by a **'Question X'** header. Carefully read each question and provide thorough answers in the following text boxes that begin with **'Answer:'**. Your project submission will be evaluated based on your answers to each of the questions and the implementation you provide.

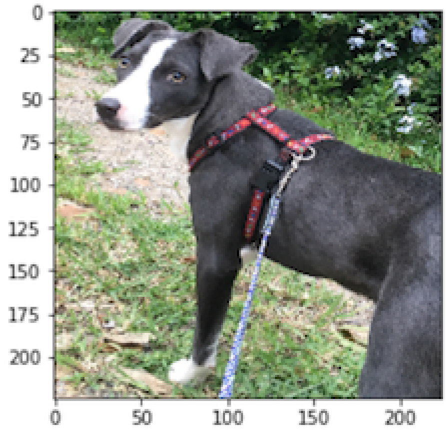
> **Note:** Code and Markdown cells can be executed using the **Shift + Enter** keyboard shortcut. Markdown cells can be edited by double-clicking the cell to enter edit mode.

The rubric contains *optional* "Stand Out Suggestions" for enhancing the project beyond the minimum requirements. If you decide to pursue the "Stand Out Suggestions", you should include the code in this IPython notebook.

Why We're Here

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

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



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

The Road Ahead

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

- Step 0: Import Datasets
- Step 1: Detect Humans
- Step 2: Detect Dogs
- Step 3: Create a CNN to Classify Dog Breeds (from Scratch)
- Step 4: Use a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 5: Create a CNN to Classify Dog Breeds (using Transfer Learning)
- Step 6: Write your Algorithm
- Step 7: Test Your Algorithm

Step 0: Import Datasets

Import Dog Dataset

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

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

In [44]:

```
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_fil
es, 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))
```

```
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 [45]:

```
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.

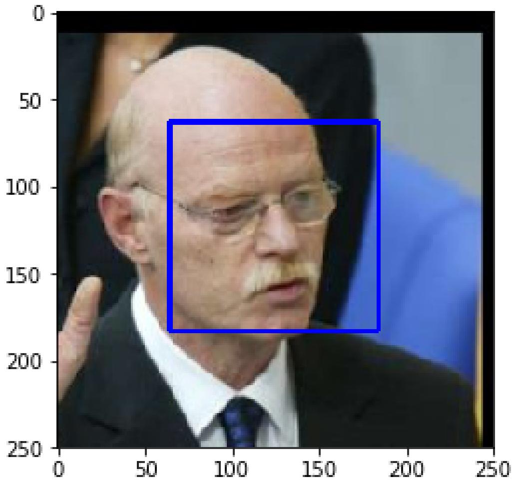
In the next code cell, we demonstrate how to use this detector to find human faces in a sample image.

In [46]:

```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_frontalface_alt.x
ml')

# load color (BGR) image
img = cv2.imread(human_files[42])
# 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()
```

```
/home/ubuntu/anaconda3/envs/dog-project/lib/python3.6/site-packages/
matplotlib/__init__.py:1085: UserWarning: Duplicate key in file "/ho
me/ubuntu/.config/matplotlib/matplotlibrc", line #2
  (fname, cnt))
/home/ubuntu/anaconda3/envs/dog-project/lib/python3.6/site-packages/
matplotlib/__init__.py:1085: UserWarning: Duplicate key in file "/ho
me/ubuntu/.config/matplotlib/matplotlibrc", line #3
  (fname, cnt))
```

```
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 [47]:

```
# 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:

- Detected 99 human faces out of 100
- Detected 11 human faces among 100 dogs

In [48]:

```
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.
true_human_faces_detected_n = sum([face_detector(h_f) for h_f in human_files_sho
rt])
print('Detected {} human faces out of 100'.format(true_human_faces_detected_n))
false_human_faces_detected_n = sum([face_detector(d_f) for d_f in dog_files_shor
t])
print('Detected {} human faces among 100 dogs'.format(false_human_faces_detected
_n))
```

```
Detected 99 human faces out of 100
Detected 11 human faces among 100 dogs
```

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: Usage of images with a face view is pretty common for detection of humans on images. But I see other ways of detecting humans, e.g. using full height images. A model trained on an approriate dataset can help avoid this limitation.

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 [49]:

```
## (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 [50]:

```
from keras.applications.resnet50 import ResNet50

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

```
WARNING:tensorflow:From /home/ubuntu/anaconda3/envs/dog-project/lib/
python3.6/site-packages/keras/backend/tensorflow_backend.py:1062: ca
lling reduce_prod (from tensorflow.python.ops.math_ops) with keep_di
ms is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
```

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 [51]:

```
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 t
ensor
    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 [52]:

```
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 [53]:

```
### 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:

- Detected 0 dogs amoung 100 humans
- Detected 100 dogs out of 100

In [54]:

```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.
s = sum([dog_detector(h_f) for h_f in human_files_short])
print('Detected {} dogs amoung 100 humans'.format(s))
s = sum([dog_detector(d_f) for d_f in dog_files_short])
print('Detected {} dogs out of 100'.format(s))
```

```
Detected 0 dogs amoung 100 humans
Detected 100 dogs out of 100
```

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

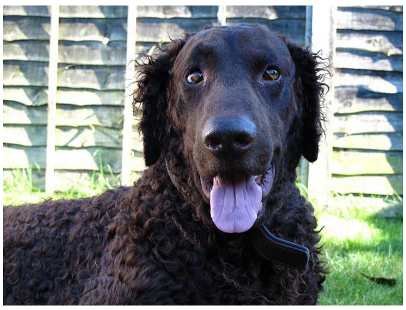
Now that we have functions for detecting humans and dogs in images, we need a way to predict breed from images. In this step, you will create a CNN that classifies dog breeds. You must create your CNN *from scratch* (so, you can't use transfer learning *yet*!), and you must attain a test accuracy of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

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

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

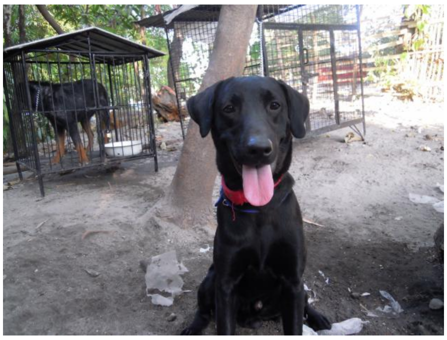
Brittany	Welsh Springer Spaniel
	

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

Curly-Coated Retriever	American Water Spaniel
	

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

Yellow Labrador	Chocolate Labrador	Black Labrador

Yellow Labrador	Chocolate Labrador	Black Labrador
		

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

Remember that the practice is far ahead of the theory in deep learning. Experiment with many different

In [55]:

```
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 [00:54<00:00, 123.49it/s]
100%|██████████| 835/835 [00:06<00:00, 137.68it/s]
100%|██████████| 836/836 [00:06<00:00, 166.76it/s]
```

In [56]:

```
train_tensors.shape
```

Out[56]:

```
(6680, 224, 224, 3)
```

(IMPLEMENTATION) Model Architecture

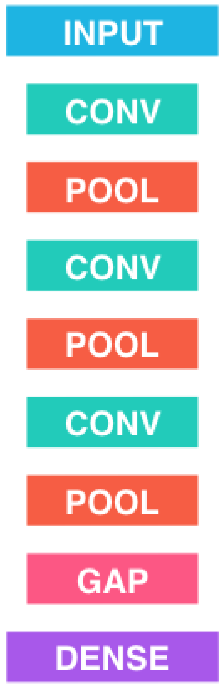
Create a CNN to classify dog breed. At the end of your code cell block, summarize the layers of your model by executing the line:

```
model.summary()
```

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

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 223, 223, 16)	208
max_pooling2d_1 (MaxPooling2	(None, 111, 111, 16)	0
conv2d_2 (Conv2D)	(None, 110, 110, 32)	2080
max_pooling2d_2 (MaxPooling2	(None, 55, 55, 32)	0
conv2d_3 (Conv2D)	(None, 54, 54, 64)	8256
max_pooling2d_3 (MaxPooling2	(None, 27, 27, 64)	0
global_average_pooling2d_1 (	(None, 64)	0
dense_1 (Dense)	(None, 133)	8645

Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0



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

Answer:

- The architecture of the model contains 4 sections. The first 3 sections are convolutional layers with max pooling and the fourth one contains fully connected layers with dropout.
- The convolutional layers are chosen the way that the size of feature maps continuously shrinks while the number of filters increases thus making the network deeper at each next section.
- Each convolutional layer uses minimum filter size (2,2) with stride 1. In order to avoid the problem of vanishing gradients, Relu is used as activation function for convolutional layers.
- The fourth section starts with global average pooling layer that convert the output of the previous layer into one dimentional verctor to reduce dimentionality of the data before dense layers. Then the global average pooling layer is connected to the pair of fully connected layers. The last layer uses softmax as an activation function to convert class scores into class probabilities.

In [57]:

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

model = Sequential()

model.add(Conv2D(64, (2, 2), input_shape=train_tensors.shape[1:]))
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2)))

model.add(Conv2D(128, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2)))

model.add(Conv2D(256, (2, 2)))
model.add(Activation('relu'))
model.add(MaxPooling2D((2,2)))

model.add(GlobalAveragePooling2D())
model.add(Dropout(0.5))
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(133))
model.add(Activation('softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 64)      832       
_________________________________________________________________
activation_50 (Activation)   (None, 223, 223, 64)      0         
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 111, 111, 64)      0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 128)     32896     
_________________________________________________________________
activation_51 (Activation)   (None, 110, 110, 128)     0         
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 55, 55, 128)       0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 256)       131328    
_________________________________________________________________
activation_52 (Activation)   (None, 54, 54, 256)       0         
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 27, 27, 256)       0         
_________________________________________________________________
global_average_pooling2d_8 ( (None, 256)               0         
_________________________________________________________________
dropout_8 (Dropout)          (None, 256)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 128)               32896     
_________________________________________________________________
dropout_9 (Dropout)          (None, 128)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 133)               17157     
_________________________________________________________________
activation_53 (Activation)   (None, 133)               0         
=================================================================
Total params: 215,109.0
Trainable params: 215,109.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [58]:

```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['ac
curacy'])
```

(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 [59]:

```
from keras.callbacks import ModelCheckpoint

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

epochs = 20

### 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 1/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8870
- acc: 0.0095Epoch 00000: val_loss improved from inf to 4.87153, sav
ing model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 13s - loss: 4.8870 - ac
c: 0.0094 - val_loss: 4.8715 - val_acc: 0.0180
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8681
- acc: 0.0138Epoch 00001: val_loss improved from 4.87153 to 4.84639,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8681 - ac
c: 0.0139 - val_loss: 4.8464 - val_acc: 0.0192
Epoch 3/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8252
- acc: 0.0186Epoch 00002: val_loss improved from 4.84639 to 4.81691,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8250 - ac
c: 0.0186 - val_loss: 4.8169 - val_acc: 0.0180
Epoch 4/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8033
- acc: 0.0191Epoch 00003: val_loss improved from 4.81691 to 4.77417,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8034 - ac
c: 0.0190 - val_loss: 4.7742 - val_acc: 0.0228
Epoch 5/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.8012
- acc: 0.0195Epoch 00004: val_loss improved from 4.77417 to 4.77097,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.8012 - ac
c: 0.0196 - val_loss: 4.7710 - val_acc: 0.0263
Epoch 6/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7814
- acc: 0.0257Epoch 00005: val_loss improved from 4.77097 to 4.76786,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7814 - ac
c: 0.0256 - val_loss: 4.7679 - val_acc: 0.0275
Epoch 7/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7613
- acc: 0.0248Epoch 00006: val_loss improved from 4.76786 to 4.70288,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7610 - ac
c: 0.0250 - val_loss: 4.7029 - val_acc: 0.0359
Epoch 8/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7365
- acc: 0.0281Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.7366 - ac
c: 0.0280 - val_loss: 4.7035 - val_acc: 0.0335
Epoch 9/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.7224
- acc: 0.0321Epoch 00008: val_loss improved from 4.70288 to 4.66033,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.7222 - ac
c: 0.0320 - val_loss: 4.6603 - val_acc: 0.0431
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6996
- acc: 0.0288Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6998 - ac
c: 0.0289 - val_loss: 4.6653 - val_acc: 0.0311
Epoch 11/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6950
```

```
- acc: 0.0357Epoch 00010: val_loss improved from 4.66033 to 4.59538,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6944 - ac
c: 0.0358 - val_loss: 4.5954 - val_acc: 0.0431
Epoch 12/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6855
- acc: 0.0368Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6848 - ac
c: 0.0368 - val_loss: 4.5992 - val_acc: 0.0407
Epoch 13/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6632
- acc: 0.0387Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6640 - ac
c: 0.0386 - val_loss: 4.6242 - val_acc: 0.0359
Epoch 14/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6540
- acc: 0.0356Epoch 00013: val_loss improved from 4.59538 to 4.58360,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6546 - ac
c: 0.0358 - val_loss: 4.5836 - val_acc: 0.0491
Epoch 15/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6568
- acc: 0.0405Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6568 - ac
c: 0.0406 - val_loss: 4.5971 - val_acc: 0.0455
Epoch 16/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6526
- acc: 0.0390Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6524 - ac
c: 0.0392 - val_loss: 4.5838 - val_acc: 0.0431
Epoch 17/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6483
- acc: 0.0375Epoch 00016: val_loss improved from 4.58360 to 4.58242,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6480 - ac
c: 0.0376 - val_loss: 4.5824 - val_acc: 0.0503
Epoch 18/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6472
- acc: 0.0386Epoch 00017: val_loss improved from 4.58242 to 4.55060,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6471 - ac
c: 0.0386 - val_loss: 4.5506 - val_acc: 0.0539
Epoch 19/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6423
- acc: 0.0429Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 12s - loss: 4.6422 - ac
c: 0.0428 - val_loss: 4.5875 - val_acc: 0.0467
Epoch 20/20
6660/6680 [============================>.] - ETA: 0s - loss: 4.6451
- acc: 0.0369Epoch 00019: val_loss improved from 4.55060 to 4.54096,
saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 12s - loss: 4.6448 - ac
c: 0.0370 - val_loss: 4.5410 - val_acc: 0.0539
```

Out[59]:

```
<keras.callbacks.History at 0x7fa6f924bcc0>
```

Load the Model with the Best Validation Loss

In [60]:

```
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 [61]:

```
# 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_targe
ts, axis=1))/len(dog_breed_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 5.6220%
```

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 [3]:

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

In [7]:

```
train_VGG16.shape
```

Out[7]:

```
(6680, 7, 7, 512)
```

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 [6]:

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

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

VGG16_model.summary()
```

```
WARNING:tensorflow:From /home/ubuntu/anaconda3/envs/dog-project/lib/
python3.6/site-packages/keras/backend/tensorflow_backend.py:1123: ca
lling reduce_mean (from tensorflow.python.ops.math_ops) with keep_di
ms is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_1 ( (None, 512)               0         
_________________________________________________________________
dense_1 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [8]:

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

```
WARNING:tensorflow:From /home/ubuntu/anaconda3/envs/dog-project/lib/
python3.6/site-packages/keras/backend/tensorflow_backend.py:2548: ca
lling reduce_sum (from tensorflow.python.ops.math_ops) with keep_dim
s is deprecated and will be removed in a future version.
Instructions for updating:
keep_dims is deprecated, use keepdims instead
```

Train the Model

In [11]:

```
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
6580/6680 [============================>.] - ETA: 0s - loss: 12.0358
- acc: 0.1278Epoch 00000: val_loss improved from inf to 10.43550, sa
ving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 3s - loss: 12.0196 - ac
c: 0.1286 - val_loss: 10.4355 - val_acc: 0.2323
Epoch 2/20
6660/6680 [============================>.] - ETA: 0s - loss: 10.0413
- acc: 0.2920Epoch 00001: val_loss improved from 10.43550 to 10.0555
8, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 10.0402 - ac
c: 0.2919 - val_loss: 10.0556 - val_acc: 0.2862
Epoch 3/20
6600/6680 [============================>.] - ETA: 0s - loss: 9.6335
- acc: 0.3521Epoch 00002: val_loss improved from 10.05558 to 9.7993
9, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.6415 - ac
c: 0.3519 - val_loss: 9.7994 - val_acc: 0.3114
Epoch 4/20
6640/6680 [============================>.] - ETA: 0s - loss: 9.4986
- acc: 0.3736Epoch 00003: val_loss improved from 9.79939 to 9.66339,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.4924 - ac
c: 0.3743 - val_loss: 9.6634 - val_acc: 0.3377
Epoch 5/20
6380/6680 [===========================>..] - ETA: 0s - loss: 9.3190
- acc: 0.3897Epoch 00004: val_loss improved from 9.66339 to 9.49949,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.3218 - ac
c: 0.3891 - val_loss: 9.4995 - val_acc: 0.3401
Epoch 6/20
6360/6680 [===========================>..] - ETA: 0s - loss: 9.1519
- acc: 0.4066Epoch 00005: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 9.1586 - ac
c: 0.4060 - val_loss: 9.5273 - val_acc: 0.3497
Epoch 7/20
6660/6680 [============================>.] - ETA: 0s - loss: 9.1254
- acc: 0.4164Epoch 00006: val_loss improved from 9.49949 to 9.49656,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.1222 - ac
c: 0.4166 - val_loss: 9.4966 - val_acc: 0.3521
Epoch 8/20
6620/6680 [============================>.] - ETA: 0s - loss: 9.0767
- acc: 0.4186Epoch 00007: val_loss improved from 9.49656 to 9.35545,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 9.0727 - ac
c: 0.4187 - val_loss: 9.3555 - val_acc: 0.3545
Epoch 9/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.9202
- acc: 0.4295Epoch 00008: val_loss improved from 9.35545 to 9.26071,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.8953 - ac
c: 0.4307 - val_loss: 9.2607 - val_acc: 0.3581
Epoch 10/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.7467
- acc: 0.4420Epoch 00009: val_loss improved from 9.26071 to 9.11387,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.7461 - ac
c: 0.4416 - val_loss: 9.1139 - val_acc: 0.3784
Epoch 11/20
```

```
6480/6680 [============================>.] - ETA: 0s - loss: 8.6339
- acc: 0.4508Epoch 00010: val_loss improved from 9.11387 to 9.02608,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.6271 - ac
c: 0.4510 - val_loss: 9.0261 - val_acc: 0.3796
Epoch 12/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.4477
- acc: 0.4608Epoch 00011: val_loss improved from 9.02608 to 8.90457,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.4542 - ac
c: 0.4603 - val_loss: 8.9046 - val_acc: 0.3677
Epoch 13/20
6440/6680 [===========================>..] - ETA: 0s - loss: 8.1827
- acc: 0.4697Epoch 00012: val_loss improved from 8.90457 to 8.66224,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.1867 - ac
c: 0.4693 - val_loss: 8.6622 - val_acc: 0.3976
Epoch 14/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.1045
- acc: 0.4841Epoch 00013: val_loss improved from 8.66224 to 8.63392,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0833 - ac
c: 0.4856 - val_loss: 8.6339 - val_acc: 0.4036
Epoch 15/20
6480/6680 [============================>.] - ETA: 0s - loss: 8.0725
- acc: 0.4904Epoch 00014: val_loss improved from 8.63392 to 8.57727,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0544 - ac
c: 0.4916 - val_loss: 8.5773 - val_acc: 0.4036
Epoch 16/20
6460/6680 [============================>.] - ETA: 0s - loss: 8.0392
- acc: 0.4950Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 8.0394 - ac
c: 0.4951 - val_loss: 8.6136 - val_acc: 0.4120
Epoch 17/20
6420/6680 [===========================>..] - ETA: 0s - loss: 8.0185
- acc: 0.4977Epoch 00016: val_loss improved from 8.57727 to 8.53568,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 8.0110 - ac
c: 0.4976 - val_loss: 8.5357 - val_acc: 0.4084
Epoch 18/20
6440/6680 [===========================>..] - ETA: 0s - loss: 7.8588
- acc: 0.5054Epoch 00017: val_loss improved from 8.53568 to 8.48561,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.8835 - ac
c: 0.5037 - val_loss: 8.4856 - val_acc: 0.4132
Epoch 19/20
6460/6680 [============================>.] - ETA: 0s - loss: 7.8601
- acc: 0.5065Epoch 00018: val_loss improved from 8.48561 to 8.46331,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.8493 - ac
c: 0.5070 - val_loss: 8.4633 - val_acc: 0.4192
Epoch 20/20
6420/6680 [===========================>..] - ETA: 0s - loss: 7.6583
- acc: 0.5160Epoch 00019: val_loss improved from 8.46331 to 8.33328,
saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 7.6990 - ac
c: 0.5132 - val_loss: 8.3333 - val_acc: 0.4132
```

Out[11]:

```
<keras.callbacks.History at 0x7fa80c3b1780>
```

Load the Model with the Best Validation Loss

In [12]:

```
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 [13]:

```
# 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: 41.1483%
```

Predict Dog Breed with the Model

In [15]:

```
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 [16]:

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

- After several experiments, the following minimal architecture was chosen as it proved to have the best validation accuracy.
- It contains of two separate parts. The first one is the ResNet50 model with the dense layers removed. The second part has just two layers: global average pooling for converting feature maps into one dimentional vectors and a standard fully connected layer with softmax activation function for predicting probabilities of image classes.
- The first part is resonsible for utilizing powerful ResNet50 model trained on millions of images of different types without focus on dog breeds. The second part specializes in detecting the classes of interest - different dog breeds.

In [35]:

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

Resnet50_model = Sequential()
Resnet50_model.add(GlobalAveragePooling2D(input_shape=train_Resnet50.shape[1:]))
# Resnet50_model.add(Dropout(0.5))
# Resnet50_model.add(Dense(128))
# Resnet50_model.add(Dropout(0.5))
Resnet50_model.add(Dense(133, activation='softmax'))

Resnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_7 ( (None, 2048)              0         
_________________________________________________________________
dense_10 (Dense)             (None, 133)               272517    
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [36]:

```
Resnet50_model.compile(optimizer='rmsprop', loss='categorical_crossentropy', met
rics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

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

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

In [37]:

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.Resnet50.hdf
5',
                               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
6600/6680 [============================>.] - ETA: 0s - loss: 1.5987
- acc: 0.6030Epoch 00000: val_loss improved from inf to 0.80527, sav
ing model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 1s - loss: 1.5876 - ac
c: 0.6052 - val_loss: 0.8053 - val_acc: 0.7533
Epoch 2/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.4364
- acc: 0.8661Epoch 00001: val_loss improved from 0.80527 to 0.67652,
saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 0s - loss: 0.4358 - ac
c: 0.8668 - val_loss: 0.6765 - val_acc: 0.8000
Epoch 3/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.2605
- acc: 0.9201Epoch 00002: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.2611 - ac
c: 0.9202 - val_loss: 0.6782 - val_acc: 0.7952
Epoch 4/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.1740
- acc: 0.9437Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.1743 - ac
c: 0.9434 - val_loss: 0.7109 - val_acc: 0.7952
Epoch 5/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.1215
- acc: 0.9625Epoch 00004: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.1213 - ac
c: 0.9623 - val_loss: 0.6846 - val_acc: 0.8192
Epoch 6/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0899
- acc: 0.9743Epoch 00005: val_loss improved from 0.67652 to 0.66046,
saving model to saved_models/weights.best.Resnet50.hdf5
6680/6680 [==============================] - 0s - loss: 0.0890 - ac
c: 0.9746 - val_loss: 0.6605 - val_acc: 0.8240
Epoch 7/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0653
- acc: 0.9812Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0650 - ac
c: 0.9813 - val_loss: 0.6698 - val_acc: 0.8180
Epoch 8/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0504
- acc: 0.9872Epoch 00007: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0503 - ac
c: 0.9873 - val_loss: 0.6995 - val_acc: 0.8192
Epoch 9/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0381
- acc: 0.9894Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0388 - ac
c: 0.9889 - val_loss: 0.7285 - val_acc: 0.8299
Epoch 10/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0274
- acc: 0.9936Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0271 - ac
c: 0.9937 - val_loss: 0.7616 - val_acc: 0.8335
Epoch 11/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0217
- acc: 0.9945Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0226 - ac
c: 0.9943 - val_loss: 0.7183 - val_acc: 0.8395
Epoch 12/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0184
```

```
- acc: 0.9964Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0183 - ac
c: 0.9964 - val_loss: 0.7846 - val_acc: 0.8263
Epoch 13/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0140
- acc: 0.9968Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0141 - ac
c: 0.9967 - val_loss: 0.7884 - val_acc: 0.8251
Epoch 14/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0146
- acc: 0.9962Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0145 - ac
c: 0.9963 - val_loss: 0.8069 - val_acc: 0.8323
Epoch 15/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0115
- acc: 0.9968Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0114 - ac
c: 0.9969 - val_loss: 0.8769 - val_acc: 0.8251
Epoch 16/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0098
- acc: 0.9979Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0098 - ac
c: 0.9979 - val_loss: 0.8562 - val_acc: 0.8263
Epoch 17/20
6540/6680 [============================>.] - ETA: 0s - loss: 0.0085
- acc: 0.9980Epoch 00016: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0086 - ac
c: 0.9979 - val_loss: 0.8835 - val_acc: 0.8240
Epoch 18/20
6580/6680 [============================>.] - ETA: 0s - loss: 0.0073
- acc: 0.9982Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0075 - ac
c: 0.9981 - val_loss: 0.9068 - val_acc: 0.8240
Epoch 19/20
6560/6680 [============================>.] - ETA: 0s - loss: 0.0059
- acc: 0.9977Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0058 - ac
c: 0.9978 - val_loss: 0.8324 - val_acc: 0.8311
Epoch 20/20
6600/6680 [============================>.] - ETA: 0s - loss: 0.0072
- acc: 0.9980Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 0s - loss: 0.0075 - ac
c: 0.9979 - val_loss: 0.9488 - val_acc: 0.8240
```

Out[37]:

```
<keras.callbacks.History at 0x7fa7768bce80>
```

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [38]:

```
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 [40]:

```
Resnet50_predictions = [np.argmax(Resnet50_model.predict(np.expand_dims(feature,
 axis=0)))
                        for feature in test_Resnet50]

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

```
Test accuracy: 79.4258%
```

(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 [42]:

```
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)]
```

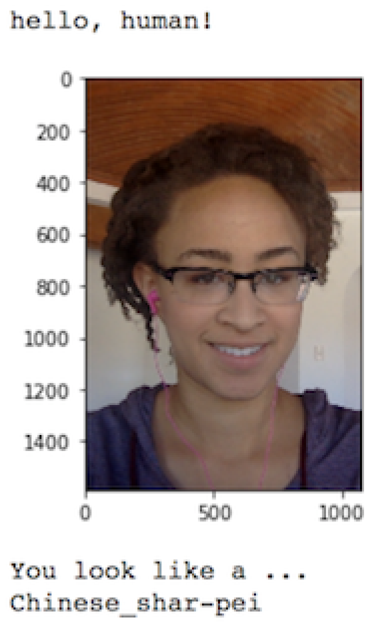
Step 6: Write your Algorithm

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!



(IMPLEMENTATION) Write your Algorithm

In [106]:

```
def dog_human_predict(img_path):
    if dog_detector(img_path):
        dog_breed = Resnet50_predict_breed(img_path)
        print("Hey doggie, you are a {}, aren't you?".format(dog_breed))
    elif face_detector(img_path):
        dog_breed = Resnet50_predict_breed(img_path)
        print("Hi human, you remind me a {}".format(dog_breed))
    else:
        print('Neither dog nor face was detected on the image.')
```

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 performance of the algorithms seems to be below the expectations. The models see people on the images with a car and don't recognize most of the dog breeds correctly.

Points of improvement:

- Transfer learning with even deeper than ResNet50 model.
- Data augmentation to increase the number of specialized image examples (with dogs).
- Experimenting with specialized layers. Use more dense layers or even include one convolutional layer.

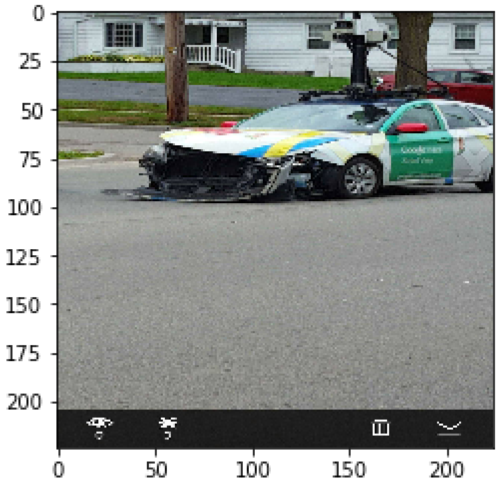
In [108]:

```
from pathlib import Path
base_path = Path('test_images')
```

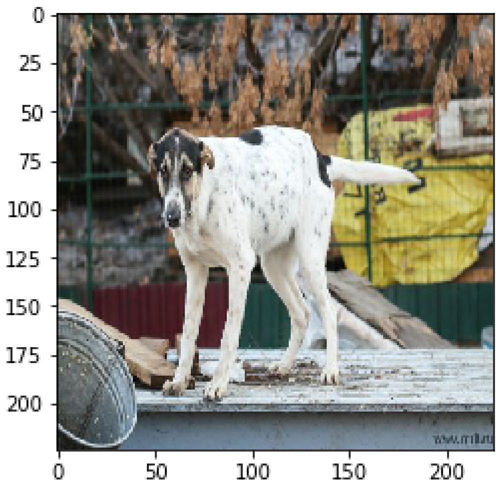
In [109]:

```
for img_path in base_path.glob('*.jpg'):
    print(img_path)
    img = image.load_img(img_path, target_size=(224, 224))
    plt.imshow(img)
    plt.show()
    dog_human_predict(str(img_path))
    print('*'*100)
```

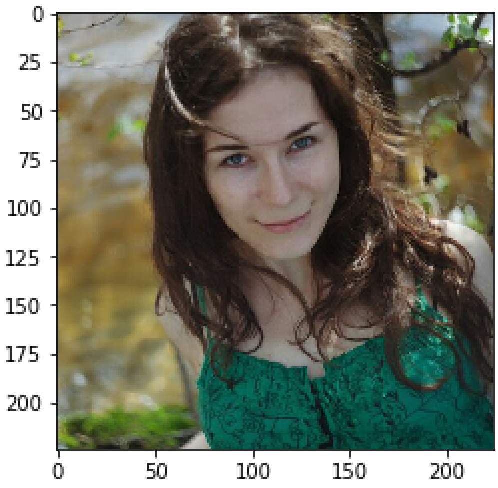
test_images/googlestreetviewcrash.jpg



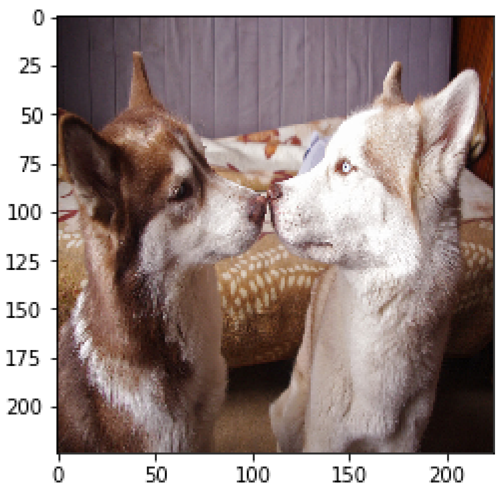
Hi human, you remind me a Greyhound
****
****
test_images/dog.jpg



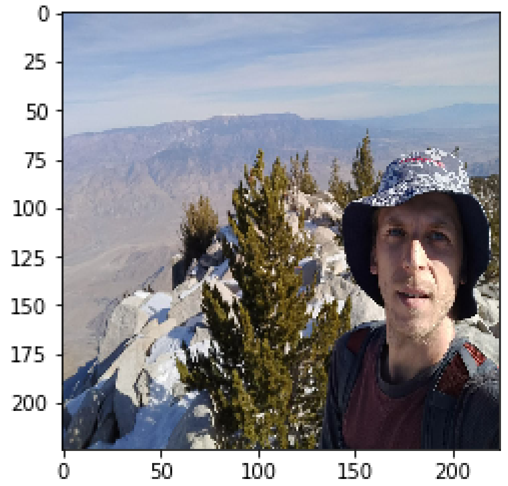
Hey doggie, you are a Great_dane, aren't you?
****
****
test_images/anna.jpg



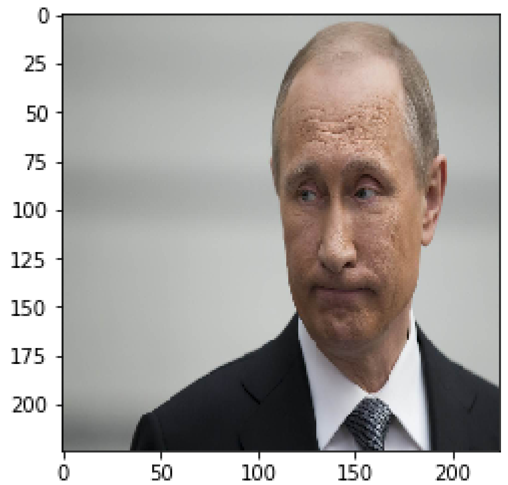

```
Hi human, you remind me a English_toy_spaniel
********************************************************************
********************************
test_images/dog2.jpg
```



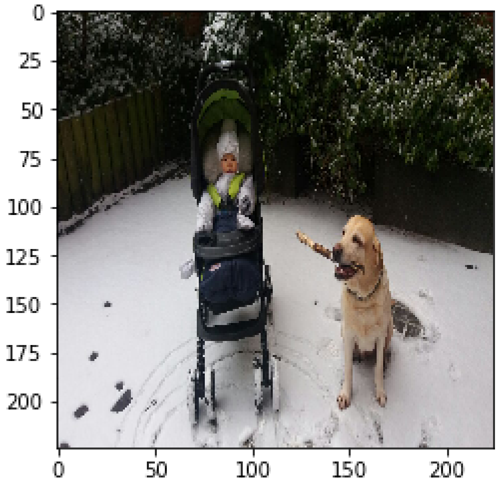

```
Hey doggie, you are a Ibizan_hound, aren't you?
********************************************************************
********************************
test_images/me.jpg
```



Hi human, you remind me a American_water_spaniel
****
****
test_images/putin.jpg



Hi human, you remind me a Brussels_griffon
****
****
test_images/labrador.jpg




```
Hey doggie, you are a Greyhound, aren't you?
********************************************************************
********************************
```