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

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_fil
es, 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.
```

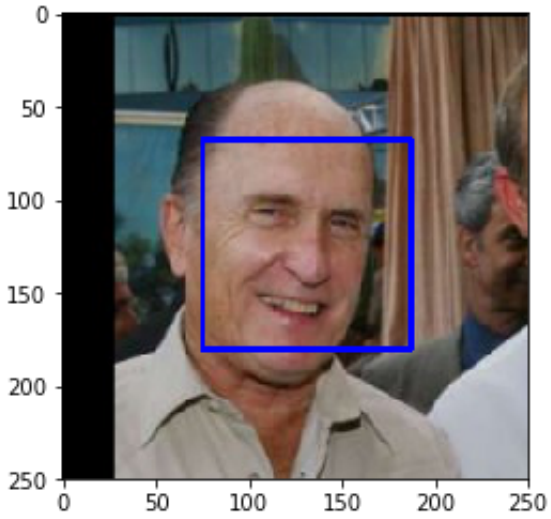
Step 1: Detect Humans

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 [3]:
```
import cv2
import matplotlib.pyplot as plt
%matplotlib inline

# extract pre-trained face detector
face_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_fron
talface_alt.xml')

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

# find faces in image
faces = face_cascade.detectMultiScale(gray)

# print number of faces detected in the image
print('Number of faces detected:', len(faces))

# get bounding box for each detected face
for (x,y,w,h) in faces:
    # add bounding box to color image
    cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)

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

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

```
Number of faces detected: 1
```



Before using any of the face detectors, it is standard procedure to convert the images to grayscale. The `detectMultiScale` function executes the classifier stored in `face_cascade` and takes the grayscale image as a parameter.

In the above code, `faces` is a numpy array of detected faces, where each row corresponds to a detected face. Each detected face is a 1D array with four entries that specifies the bounding box of the detected face. The first two entries in the array (extracted in the above code as `x` and `y`) specify the horizontal and vertical positions of the top left corner of the bounding box. The last two entries in the array (extracted here as `w` and `h`) specify the width and height of the box.

Write a Human Face Detector

We can use this procedure to write a function that returns `True` if a human face is detected in an image and `False` otherwise. This function, aptly named `face_detector`, takes a string-valued file path to an image as input and appears in the code block below.

In [4]:
```
# returns "True" if face is detected in image stored at img_path
def face_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade.detectMultiScale(gray)
    return len(faces) > 0
```

(IMPLEMENTATION) Assess the Human Face Detector

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

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

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

Answer:

```
1. 100% of the first 100 images in `human_files` have a detected human
face.
2. 11% of the first 100 images in `dog_files` have a detected human fac
e.
```

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

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

index =0
detected_count =0
print("Detecting whether human images are detected as human:" )
for human in human_files_short[:100]:
    detected=face_detector(human)
    if detected== True:
        detected_count = detected_count +1

    index = index +1
    if (index%10==0):
        print(index," ", human[-15:], "detected=", detected, ", det
ected_count=", detected_count)

print("{}% of the first 100 images in human_files have a detected h
uman face.\n".format(detected_count))


index =0
detected_count_dog =0
print("Detecting whether dog images are detected as human:")
for dog in dog_files_short[:100]:
    detected=face_detector(dog)
    if detected== True:
        detected_count_dog = detected_count_dog +1

    index = index +1
    if (index%10==0):
        print(index," ", dog[-15:], "detected=", detected, ", detec
ted_count=", detected_count_dog)

print("{}% of the first 100 images in `dog_files` have a detected h
uman face.".format(detected_count_dog))
```

```
Detecting whether human images are detected as human:
10   Stuart_0001.jpg detected= True , detected_count= 10
20   eneman_0008.jpg detected= True , detected_count= 20
30   _Bjorn_0001.jpg detected= True , detected_count= 30
40   odorov_0001.jpg detected= True , detected_count= 40
50   Powell_0004.jpg detected= True , detected_count= 50
60   nnelly_0003.jpg detected= True , detected_count= 60
70   _Ridge_0002.jpg detected= True , detected_count= 70
80   niston_0021.jpg detected= True , detected_count= 80
90   _Blair_0123.jpg detected= True , detected_count= 90
100   rtusch_0001.jpg detected= True , detected_count= 100
100% of the first 100 images in human_files have a detected human
face.

Detecting whether dog images are detected as human:
10   hound_05559.jpg detected= False , detected_count= 1
20   _chow_03650.jpg detected= False , detected_count= 3
30   dland_06982.jpg detected= False , detected_count= 7
40   etter_04407.jpg detected= False , detected_count= 9
50   ehund_07220.jpg detected= False , detected_count= 9
60   aniel_03305.jpg detected= False , detected_count= 9
70   hound_06130.jpg detected= False , detected_count= 10
80   dland_07004.jpg detected= False , detected_count= 11
90   shund_03948.jpg detected= False , detected_count= 11
100   e_dog_00744.jpg detected= False , detected_count= 11
11% of the first 100 images in `dog_files` have a detected human f
ace.
```

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:

In my opinion, this is a reasonable expectation to post on the user as this greatly improve the accuracy of face detection. Since image detectors are using features to detect human face. If no clear view of face, it won't work well with high accuracy. For example, if we mixed dog images with human images, there will be dog images detected as human face (See the above cell's result). This is understoodable as dog face has similar features as human face such as having two eyes, one nose and one mouth etc.

Optional 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.

Answer: I used face detector from OpenCV to detect human images in my algorithm below. It is modified from the https://pythonprogramming.net/haar-cascade-face-eye-detection-python-opencv-tutorial/ (https://pythonprogramming.net/haar-cascade-face-eye-detection-python-opencv-tutorial/) . The program from the above link gives very good example of how to capture a photo using the computer camera and then detect faces and eyes in the captured photos.

Before running, I downloaded the following the face and eyes model files

1. haarcascade_frontalface_default.xml and
2. haarcascade_eye.xml from https://github.com/opencv/opencv/tree/master/data/haarcascades (https://github.com/opencv/opencv/tree/master/data/haarcascades) to the haarcascades folder.

I modified the program such that it can run on my MacBook with my python environment. A face_eyes_detector is written based on the modified code below. The face_eyes_detector will return True as long as one of the faces have eye detected. A face_detector2 is written on using haarcascade_frontalface_default.xml model file.

The performance of face_detector2 and face_eyes_detector are summarized below:

- 100% of the first 100 images in `human_files` have a detected human face.
- 85% of the first 100 images in `human_files` have a detected human face with eyes.
- 100% of the first 100 images in `dog_files` have a detected human face.
- 11% of the first 100 images in `dog_files` have a detected human face with eyes.

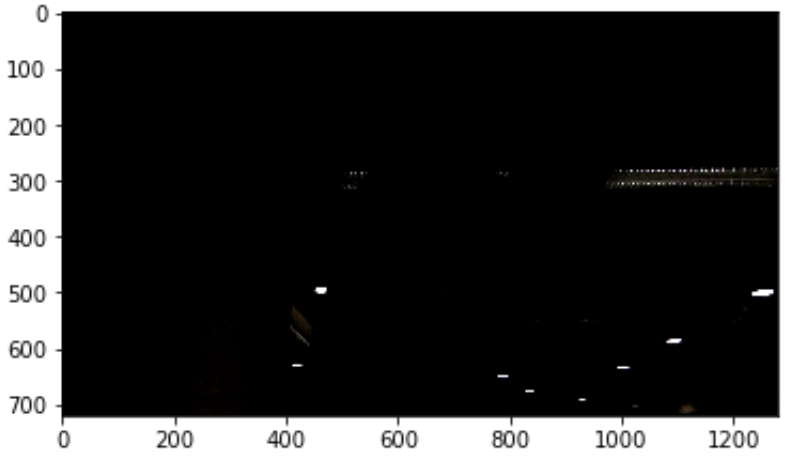
It seems that face_detector2 is unable to differentiate between human face and dog face. If mix human images with dog images, face_eyes_detector's performance also suffers.

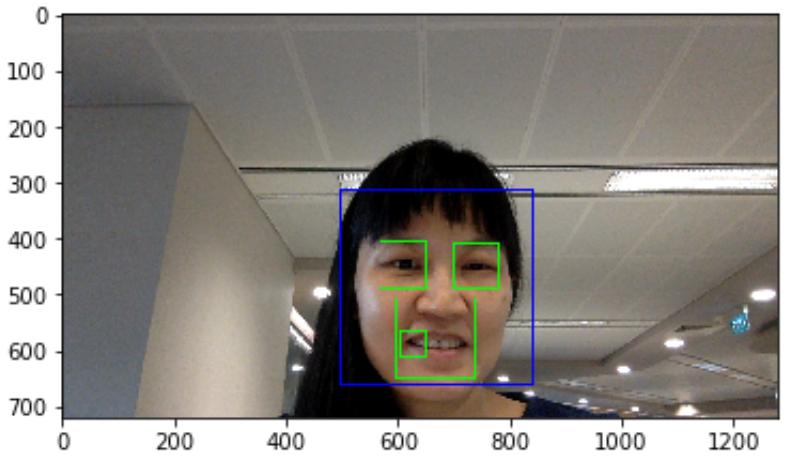
In [6]:
```
from  datetime import date, datetime, timedelta
import time,threading

#print(datetime.now())
begin_time = datetime.now()
current_time = begin_time + timedelta(seconds=10)
print(begin_time, current_time)
```

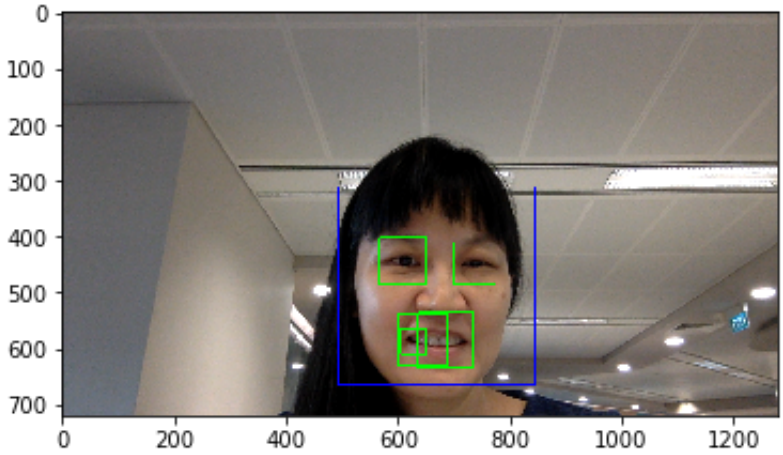
```
2019-02-22 18:13:59.829597 2019-02-22 18:14:09.829597
```

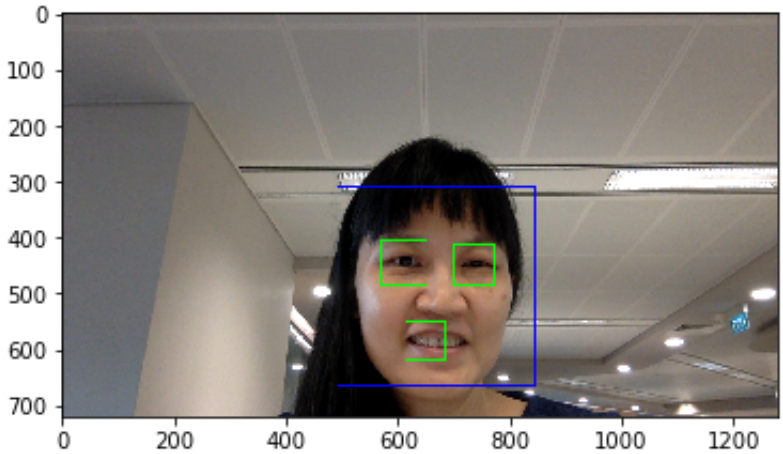
In [7]:
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.
### Ref: https://pythonprogramming.net/haar-cascade-face-eye-detect
ion-python-opencv-tutorial/
### Below is the code from the above link. Before running, I downlo
aded the following two files
### 1. haarcascade_frontalface_default.xml and
### 2. haarcascade_eye.xml
### from https://github.com/opencv/opencv/tree/master/data/haarcasc
ades to the haarcascades folder
import matplotlib.pyplot as plt
import numpy as np
import cv2

%matplotlib inline


# multiple cascades: https://github.com/Itseez/opencv/tree/master/d
ata/haarcascades

#https://github.com/Itseez/opencv/blob/master/data/haarcascades/haa
rcascade_frontalface_default.xml
face_cascade2 = cv2.CascadeClassifier('haarcascades/haarcascade_fro
ntalface_default.xml')
#https://github.com/Itseez/opencv/blob/master/data/haarcascades/haa
rcascade_eye.xml
eye_cascade = cv2.CascadeClassifier('haarcascades/haarcascade_eye.x
ml')

cap = cv2.VideoCapture(0)



count=0
while (count<5):
    ret, img = cap.read()
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade2.detectMultiScale(gray, 1.3, 5)

    for (x,y,w,h) in faces:
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        roi_gray = gray[y:y+h, x:x+w]
```

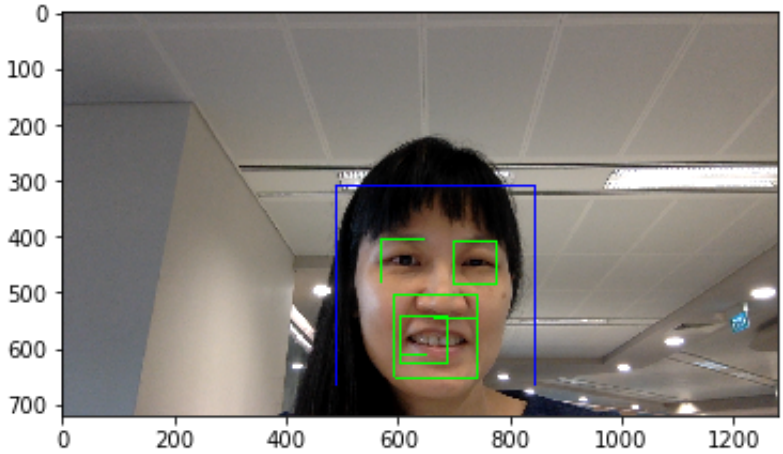
```
        roi_color = img[y:y+h, x:x+w]

        eyes = eye_cascade.detectMultiScale(roi_gray)
        for (ex,ey,ew,eh) in eyes:
            cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0)
,2)

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

    plt.imshow(cv_rgb)
    plt.show()
    count = count+1

cap.release()
```






0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200


0
100
200
300
400
500
600
700
0
200
400
600
800
1000
1200

In [8]:
```
def face_detector2(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade2.detectMultiScale(gray)
    return len(faces) > 0


def face_eyes_detector(img_path):
    img = cv2.imread(img_path)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    faces = face_cascade2.detectMultiScale(gray)

    for (x,y,w,h) in faces:
        cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
        roi_gray = gray[y:y+h, x:x+w]
        roi_color = img[y:y+h, x:x+w]

        eyes = eye_cascade.detectMultiScale(roi_gray)
        for (ex,ey,ew,eh) in eyes:
            cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(0,255,0)
,2)
        return len(eyes) > 0
```

In [9]:
```
def human_face_eyes_dog_detector(img_path):
    if dog_detector(img_path) == True:
        print("A dog is detected!")
    if face_detector(img_path) == True:
        print("A human is detected!")
    if face_eyes_detector(img_path) == True:
        print("A human with face and eyes is detected!")
    elif dog_detector(img_path) == False and face_eyes_detector(img
_path) == False and face_detector(img_path) == False:
        print("No dog or human face or eyes is detected!")
```

In [10]:
```
index =0
detected_count_face =0
detected_count_eyes=0
print("Detecting whether human images are detected as human:" )
for human in human_files_short[:100]:
    detected=face_detector2(human)
    detected_eyes=face_eyes_detector(human)
    if detected== True:
        detected_count_face = detected_count_face +1
    if detected_eyes== True:
        detected_count_eyes = detected_count_eyes +1


    index = index +1
    if (index%10==0):
        print(index," ", human[-15:], "detected=", detected, "detec
ted_count=", detected_count_face)

print("{}% of the first 100 images in human_files have a detected h
uman face.".format(detected_count_face))
print("{}% of the first 100 images in human_files have a detected h
uman face with eyes.\n".format(detected_count_eyes))



index =0
detected_count_dog =0
detected_count_dog_eyes =0

print("Detecting whether dog images are detected as human:")
for dog in dog_files_short[:100]:
    detected=face_detector2(human)
    detected_eyes=face_eyes_detector(dog)
    if detected== True:
        detected_count_dog = detected_count_dog +1
    if detected_eyes== True:
        detected_count_dog_eyes = detected_count_dog_eyes +1


    index = index +1
    if (index%10==0):
        print(index," ", dog[-15:], "detected=", detected, "detecte
d_count=", detected_count_dog)

print("{}% of the first 100 images in `dog_files` have a detected h
uman face.".format(detected_count_dog))
print("{}% of the first 100 images in `dog_files` have a detected h
uman face with eyes.".format(detected_count_dog_eyes))
```

```
Detecting whether human images are detected as human:
10   Stuart_0001.jpg detected= True detected_count= 10
20   eneman_0008.jpg detected= True detected_count= 20
30   _Bjorn_0001.jpg detected= True detected_count= 30
40   odorov_0001.jpg detected= True detected_count= 40
50   Powell_0004.jpg detected= True detected_count= 50
60   nnelly_0003.jpg detected= True detected_count= 60
70   _Ridge_0002.jpg detected= True detected_count= 70
80   niston_0021.jpg detected= True detected_count= 80
90   _Blair_0123.jpg detected= True detected_count= 90
100   rtusch_0001.jpg detected= True detected_count= 100
100% of the first 100 images in human_files have a detected human
face.
85% of the first 100 images in human_files have a detected human f
ace with eyes.

Detecting whether dog images are detected as human:
10   hound_05559.jpg detected= True detected_count= 10
20   _chow_03650.jpg detected= True detected_count= 20
30   dland_06982.jpg detected= True detected_count= 30
40   etter_04407.jpg detected= True detected_count= 40
50   ehund_07220.jpg detected= True detected_count= 50
60   aniel_03305.jpg detected= True detected_count= 60
70   hound_06130.jpg detected= True detected_count= 70
80   dland_07004.jpg detected= True detected_count= 80
90   shund_03948.jpg detected= True detected_count= 90
100   e_dog_00744.jpg detected= True detected_count= 100
100% of the first 100 images in `dog_files` have a detected human
face.
11% of the first 100 images in `dog_files` have a detected human f
ace with eyes.
```

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 [11]:
```
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 [12]:
```
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, 22
4, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) an
d 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 tqd
m(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 [13]:
```
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 [14]:
```
### 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))
```

In [15]:
```
### TODO: Test the performance of the dog_detector function
### on the images in human_files_short and dog_files_short.

index =0
detected_count =0
print("Detecting whether human images are detected as human:" )
for human in human_files_short[:100]:
    detected=dog_detector(human)
    if detected== True:
        detected_count = detected_count +1

    index = index +1
    if (index%10==0):
        print(index," ", human[-15:], "detected=", detected, "detec
ted_count=", detected_count)

print("{}% of the images in human_files_short have a detected dog.\
n".format(detected_count))


index =0
detected_count_dog =0
print("Detecting whether dog images are detected as human:")
for dog in dog_files_short[:100]:
    detected=dog_detector(dog)
    if detected== True:
        detected_count_dog = detected_count_dog +1

    index = index +1
    if (index%10==0):
        print(index," ", dog[-15:], "detected=", detected, "detecte
d_count=", detected_count_dog)

print("{}% of the images in dog_files_short have a detected dog.".f
ormat(detected_count_dog))
```

```
Detecting whether human images are detected as human:
10   Stuart_0001.jpg detected= False detected_count= 0
20   eneman_0008.jpg detected= False detected_count= 0
30   _Bjorn_0001.jpg detected= False detected_count= 0
40   odorov_0001.jpg detected= False detected_count= 0
50   Powell_0004.jpg detected= False detected_count= 0
60   nnelly_0003.jpg detected= False detected_count= 0
70   _Ridge_0002.jpg detected= False detected_count= 0
80   niston_0021.jpg detected= False detected_count= 0
90   _Blair_0123.jpg detected= False detected_count= 0
100   rtusch_0001.jpg detected= False detected_count= 0
0% of the images in human_files_short have a detected dog.

Detecting whether dog images are detected as human:
10   hound_05559.jpg detected= True detected_count= 10
20   _chow_03650.jpg detected= True detected_count= 20
30   dland_06982.jpg detected= True detected_count= 30
40   etter_04407.jpg detected= True detected_count= 40
50   ehund_07220.jpg detected= True detected_count= 50
60   aniel_03305.jpg detected= True detected_count= 60
70   hound_06130.jpg detected= True detected_count= 70
80   dland_07004.jpg detected= True detected_count= 80
90   shund_03948.jpg detected= True detected_count= 90
100   e_dog_00744.jpg detected= True detected_count= 100
100% of the images in dog_files_short have a detected dog.
```

(IMPLEMENTATION) Assess the Dog Detector

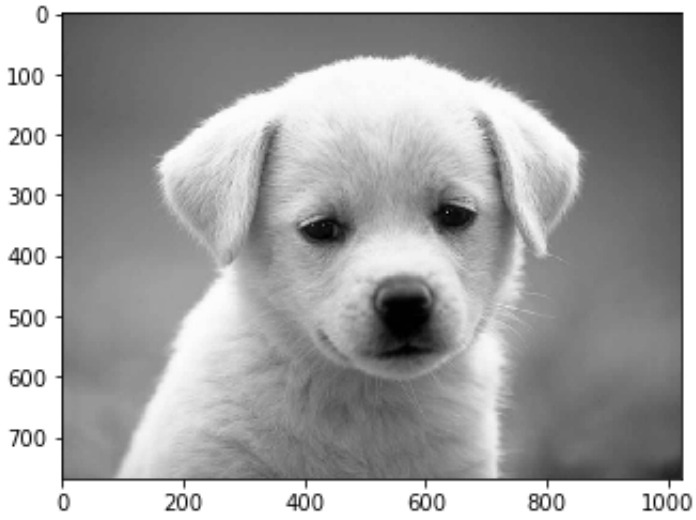
Question 3: Use the code cell below to test the performance of your `dog_detector` function.

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

Answer:

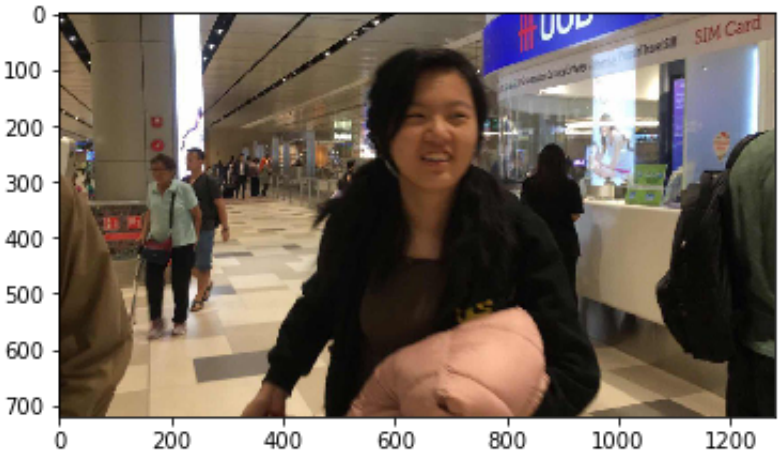
1. 0% of the images in `human_files_short` have a detected dog.
2. 100% of the images in `dog_files_short` have a detected dog.

In [16]:
```
import cv2
from os import listdir
from os.path import join
images_path= 'my_Images/'
imageNames = listdir(images_path)

image_number =1;
for imagename in imageNames:
    if imagename.endswith(".jpg"):
        img = cv2.imread(join(images_path,imagename))
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()
        print("==================", image_number, imagename, "=====
=============")
        human_face_eyes_dog_detector(join(images_path,imagename))
        image_number = image_number +1
        print("==================================================
=============")
```

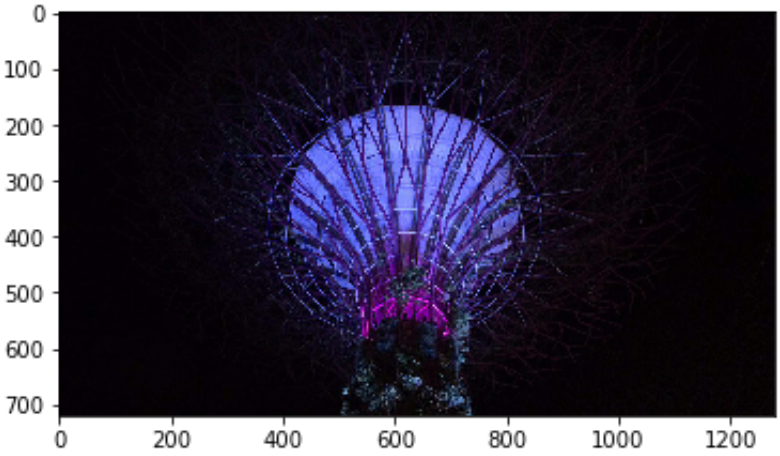



```
================== 1 dog.jpg ==================
A dog is detected!
================================================================
```

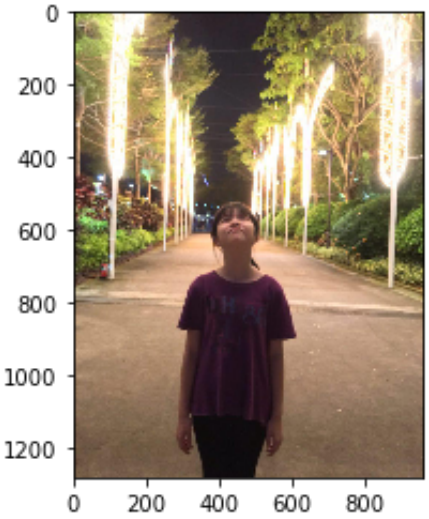

```
================== 2 1600px-Lava_the_sled_dog.jpg ================
==
A dog is detected!
====================================================================
```



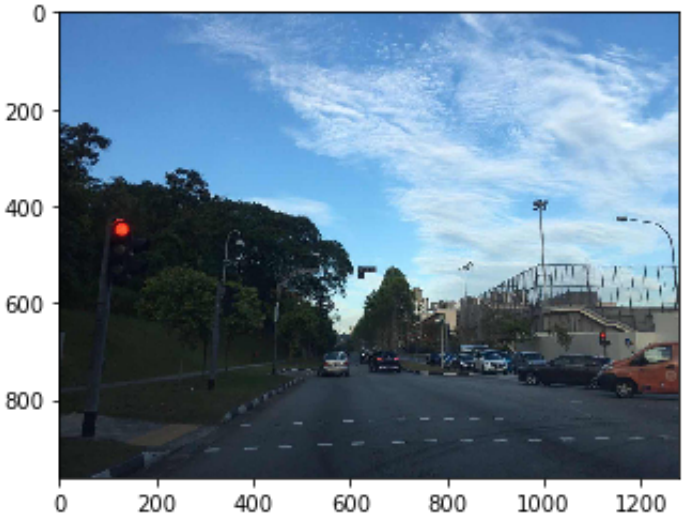

```
================== 3 anna.jpg ==================
A human is detected!
====================================================================
```



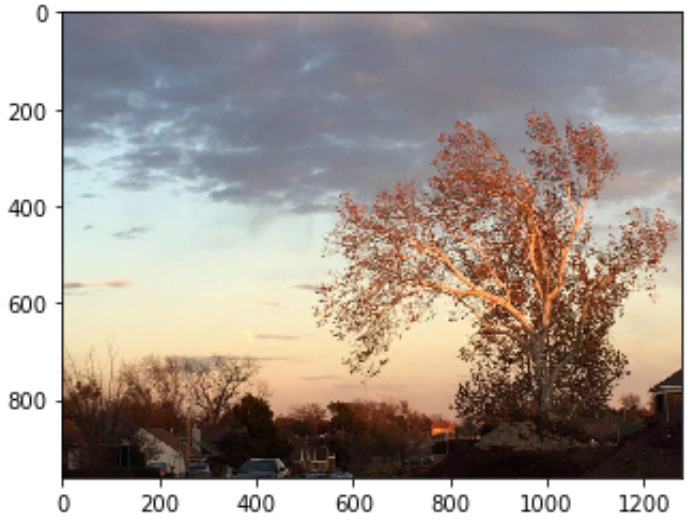
```
================== 4 ele_tree.jpg ==================
A human is detected!
====================================================================
```



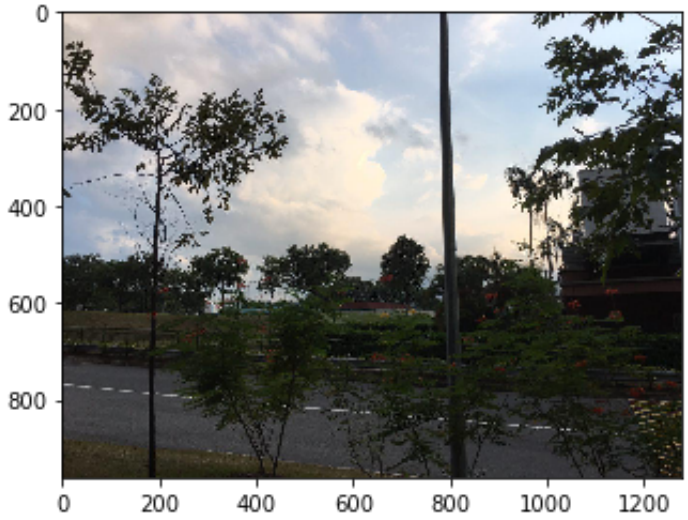

```
================== 5 dian.jpg ==================
A human is detected!
====================================================================
```



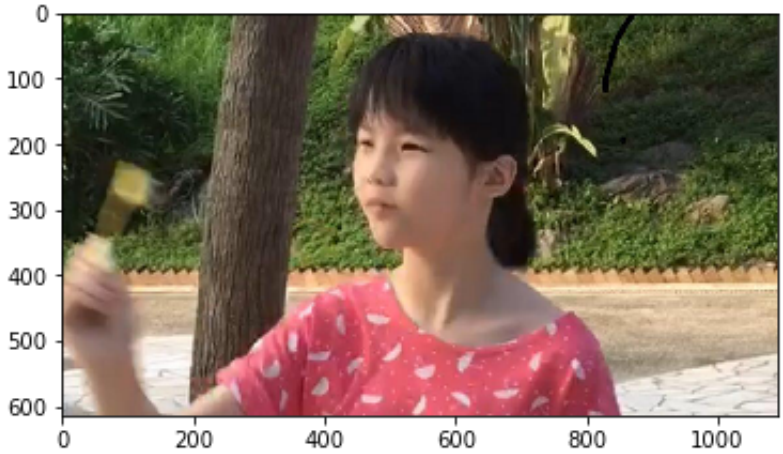

```
================== 6 sky.jpg ==================
====================================================================
```




```
================== 7 tree.jpg ==================
No dog or human face or eyes is detected!
================================================================
```




```
================== 8 trees.jpg ==================
No dog or human face or eyes is detected!
================================================================
```



```
================== 9 dian3.jpg ==================
A human is detected!
A human with face and eyes is detected!
================================================================
```


```
================== 10 dian2.jpg ==================
A human is detected!
A human with face and eyes is 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**

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 [17]:
```
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 [01:52<00:00, 59.20it/s]
100%|██████████| 835/835 [00:13<00:00, 61.65it/s]
100%|██████████| 836/836 [00:13<00:00, 62.27it/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:

Sample CNN

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

Answer:

I decided to use the hinted architeture above for image recognition. It consists of a series of convolutional layers with periodic spatial reduction via pooling and a fully connected layer bridging the information learning during convolutions and the final output predictions.

I have also tested a slightly different architecture, which I probably seen on the web but I can't find the link now, in model2 below for image recognition. The major differences are using slightly larger kernel size for feature pattern with larger spatial reduction ratio 3 via pooling and added dropout to prevent overfitting.

Both architectures give similar accuracy >1%. I have tried to change the kernel size but it doesn't imporve the accuracy much. I have also changed the padding to "same" so that we don't lost information from layer to layer. But that also don't improve the accuracy much. Increase the number of layers and epochs will talk longer time. I decided to move forward to transfer learning. Transfer learning may produce much better accuracy since it reuses pre-trained layers.

In [18]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling
2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential

model = Sequential()

### TODO: Define your architecture.
img_width, img_height = 224, 224
dog_breeds = len(dog_names)
model.add(Conv2D(filters = 16, kernel_size = (2,2), strides = (1,1)
, padding = 'valid', activation = 'relu',
          input_shape = (img_width, img_height, 3))) #RGB image
model.add(MaxPooling2D(pool_size=(2, 2), strides=None, padding='val
id'))
model.add(Conv2D(filters = 32, kernel_size = (2,2), strides = (1,1)
, padding = 'valid', activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=None, padding='val
id'))
model.add(Conv2D(filters = 64, kernel_size = (2,2), strides = (1,1)
, padding = 'valid', activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=None, padding='val
id'))
model.add(GlobalAveragePooling2D())
model.add(Dense(dog_breeds, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_4 (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
_________________________________________________________________
```

Compile the Model

In [19]:
```
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 [20]:
```
from keras.callbacks import ModelCheckpoint

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

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], v
erbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.883
9 - acc: 0.0102      Epoch 00000: val_loss improved from inf to 4.
86621, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 157s - loss: 4.8838 -
acc: 0.0102 - val_loss: 4.8662 - val_acc: 0.0156
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.848
3 - acc: 0.0144      Epoch 00001: val_loss improved from 4.86621 t
o 4.82501, saving model to saved_models/weights.best.from_scratch.
hdf5
6680/6680 [==============================] - 145s - loss: 4.8482 -
acc: 0.0144 - val_loss: 4.8250 - val_acc: 0.0180
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.801
4 - acc: 0.0182      Epoch 00002: val_loss improved from 4.82501 t
o 4.79356, saving model to saved_models/weights.best.from_scratch.
hdf5
6680/6680 [==============================] - 128s - loss: 4.8017 -
acc: 0.0183 - val_loss: 4.7936 - val_acc: 0.0192
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.765
9 - acc: 0.0189  Epoch 00003: val_loss improved from 4.79356 to 4.
76874, saving model to saved_models/weights.best.from_scratch.hdf5
6680/6680 [==============================] - 137s - loss: 4.7661 -
acc: 0.0189 - val_loss: 4.7687 - val_acc: 0.0228
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.734
2 - acc: 0.0222      Epoch 00004: val_loss improved from 4.76874 t
o 4.74793, saving model to saved_models/weights.best.from_scratch.
hdf5
6680/6680 [==============================] - 150s - loss: 4.7354 -
acc: 0.0222 - val_loss: 4.7479 - val_acc: 0.0263
```

Out[20]: <keras.callbacks.History at 0x125e74e48>

Load the Model with the Best Validation Loss

In [21]:
```
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 [22]:
```
# get index of predicted dog breed for each image in test set
dog_breed_predictions = [np.argmax(model.predict(np.expand_dims(ten
sor, axis=0))) for tensor in test_tensors]

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

```
Test accuracy: 3.8278%
```

Try another model, model2, and see how is the accuracy

In [23]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling
2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras import regularizers

### TODO: Define your architecture.
img_width, img_height = 224, 224
dog_breeds = len(dog_names)

model2 = Sequential()
model2.add(Conv2D(filters = 16, kernel_size = (5,5), strides = (2,2
), padding = 'valid', activation = 'relu',
          input_shape = (img_width, img_height, 3))) #RGB image
model2.add(Conv2D(filters = 32, kernel_size = (5,5), strides = (4,4
), padding = 'valid', activation = 'relu'))
model2.add(MaxPooling2D(pool_size=(3, 3), strides=None, padding='va
lid'))
model2.add(Conv2D(filters = 64, kernel_size = (2,2), strides = (2,2
), padding = 'valid', activation = 'relu'))
model2.add(GlobalAveragePooling2D())
model2.add(Dense(200, activation='relu'))
model2.add(Dropout(0.4))
model2.add(Dense(dog_breeds, activation='softmax'))

model2.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 110, 110, 16)      1216
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 27, 27, 32)        12832
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 9, 9, 32)          0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 4, 4, 64)          8256
_________________________________________________________________
global_average_pooling2d_2 ( (None, 64)                0
_________________________________________________________________
dense_2 (Dense)              (None, 200)               13000
_________________________________________________________________
dropout_1 (Dropout)          (None, 200)               0
_________________________________________________________________
dense_3 (Dense)              (None, 133)               26733
=================================================================
Total params: 62,037.0
Trainable params: 62,037.0
Non-trainable params: 0.0
_________________________________________________________________
```

In [24]:
```
model2.compile(optimizer='rmsprop', loss='categorical_crossentropy'
, metrics=['accuracy'])
```

In [25]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 5

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

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.
from_scratch3.hdf5',
                               verbose=1, save_best_only=True)

model2.fit(train_tensors, train_targets,
          validation_data=(valid_tensors, valid_targets),
          epochs=epochs, batch_size=20, callbacks=[checkpointer], v
erbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.886
0 - acc: 0.0074      Epoch 00000: val_loss improved from inf to 4.
87368, saving model to saved_models/weights.best.from_scratch3.hdf
5
6680/6680 [==============================] - 50s - loss: 4.8861 -
acc: 0.0073 - val_loss: 4.8737 - val_acc: 0.0108
Epoch 2/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.869
5 - acc: 0.0113     Epoch 00001: val_loss improved from 4.87368 to
4.84312, saving model to saved_models/weights.best.from_scratch3.h
df5
6680/6680 [==============================] - 49s - loss: 4.8696 -
acc: 0.0112 - val_loss: 4.8431 - val_acc: 0.0192
Epoch 3/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.802
6 - acc: 0.0176     Epoch 00002: val_loss improved from 4.84312 to
4.74602, saving model to saved_models/weights.best.from_scratch3.h
df5
6680/6680 [==============================] - 49s - loss: 4.8026 -
acc: 0.0175 - val_loss: 4.7460 - val_acc: 0.0251
Epoch 4/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.729
9 - acc: 0.0197     Epoch 00003: val_loss did not improve
6680/6680 [==============================] - 50s - loss: 4.7301 -
acc: 0.0198 - val_loss: 4.7628 - val_acc: 0.0251
Epoch 5/5
6660/6680 [============================>.] - ETA: 0s - loss: 4.659
0 - acc: 0.0246     Epoch 00004: val_loss improved from 4.74602 to
4.63336, saving model to saved_models/weights.best.from_scratch3.h
df5
6680/6680 [==============================] - 50s - loss: 4.6592 -
acc: 0.0246 - val_loss: 4.6334 - val_acc: 0.0240
```

Out[25]: <keras.callbacks.History at 0x125c37f98>

In [26]:
```
model2.load_weights('saved_models/weights.best.from_scratch3.hdf5')
```

In [27]:
```
# get index of predicted dog breed for each image in test set
dog_breed_predictions2 = [np.argmax(model2.predict(np.expand_dims(t
ensor, axis=0))) for tensor in test_tensors]

# report test accuracy
test_accuracy2 = 100*np.sum(np.array(dog_breed_predictions2)==np.ar
gmax(test_targets, axis=1))/len(dog_breed_predictions)
print('Test accuracy of model2: %.4f%%' % test_accuracy2)
```

```
Test accuracy of model2: 3.8278%
```

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 [28]:
```
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 [29]:
```
from keras import regularizers

VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(180, activation='relu', kernel_regularizer=regularizers.l2(0.01)))

VGG16_model.add(Dropout(0.4))

VGG16_model.add(Dense(133, activation='softmax'))


VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_3 ( (None, 512)               0         
_________________________________________________________________
dense_4 (Dense)              (None, 180)               92340     
_________________________________________________________________
dropout_2 (Dropout)          (None, 180)               0         
_________________________________________________________________
dense_5 (Dense)              (None, 133)               24073     
=================================================================
Total params: 116,413.0
Trainable params: 116,413.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [30]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])
```

Train the Model

In [31]:
```
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
6540/6680 [============================>.] - ETA: 0s - loss: 6.735
4 - acc: 0.1220   Epoch 00000: val_loss improved from inf to 3.409
91, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 6.6787 - a
cc: 0.1253 - val_loss: 3.4099 - val_acc: 0.3593
Epoch 2/20
6400/6680 [===========================>..] - ETA: 0s - loss: 3.118
2 - acc: 0.3817Epoch 00001: val_loss improved from 3.40991 to 1.92
343, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 3.0972 - a
cc: 0.3840 - val_loss: 1.9234 - val_acc: 0.6012
Epoch 3/20
6540/6680 [============================>.] - ETA: 0s - loss: 2.282
0 - acc: 0.5229Epoch 00002: val_loss improved from 1.92343 to 1.69
201, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 2.2794 - a
cc: 0.5234 - val_loss: 1.6920 - val_acc: 0.6587
Epoch 4/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.937
0 - acc: 0.5884Epoch 00003: val_loss improved from 1.69201 to 1.47
527, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 1.9361 - a
cc: 0.5883 - val_loss: 1.4753 - val_acc: 0.6838
Epoch 5/20
6400/6680 [===========================>..] - ETA: 0s - loss: 1.757
3 - acc: 0.6259Epoch 00004: val_loss improved from 1.47527 to 1.41
798, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 1.7466 - a
cc: 0.6278 - val_loss: 1.4180 - val_acc: 0.6958
Epoch 6/20
6640/6680 [============================>.] - ETA: 0s - loss: 1.606
4 - acc: 0.6512Epoch 00005: val_loss improved from 1.41798 to 1.30
674, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 1.6079 - a
cc: 0.6513 - val_loss: 1.3067 - val_acc: 0.7353
Epoch 7/20
6440/6680 [===========================>..] - ETA: 0s - loss: 1.552
8 - acc: 0.6741Epoch 00006: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.5501 - a
cc: 0.6747 - val_loss: 1.3844 - val_acc: 0.7030
Epoch 8/20
6460/6680 [============================>.] - ETA: 0s - loss: 1.483
3 - acc: 0.6827Epoch 00007: val_loss improved from 1.30674 to 1.23
632, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 1.4872 - a
cc: 0.6817 - val_loss: 1.2363 - val_acc: 0.7413
Epoch 9/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.453
7 - acc: 0.6859Epoch 00008: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.4504 - a
cc: 0.6870 - val_loss: 1.3782 - val_acc: 0.7102
Epoch 10/20
6660/6680 [============================>.] - ETA: 0s - loss: 1.431
```

```
8 - acc: 0.7021Epoch 00009: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.4306 - a
cc: 0.7022 - val_loss: 1.3607 - val_acc: 0.7413
Epoch 11/20
6480/6680 [============================>.] - ETA: 0s - loss: 1.398
0 - acc: 0.7065Epoch 00010: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.4040 - a
cc: 0.7055 - val_loss: 1.2491 - val_acc: 0.7413
Epoch 12/20
6520/6680 [============================>.] - ETA: 0s - loss: 1.382
4 - acc: 0.7074Epoch 00011: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3850 - a
cc: 0.7072 - val_loss: 1.2671 - val_acc: 0.7389
Epoch 13/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.359
0 - acc: 0.7163Epoch 00012: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3657 - a
cc: 0.7147 - val_loss: 1.2686 - val_acc: 0.7437
Epoch 14/20
6560/6680 [============================>.] - ETA: 0s - loss: 1.379
4 - acc: 0.7084Epoch 00013: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3828 - a
cc: 0.7082 - val_loss: 1.3354 - val_acc: 0.7449
Epoch 15/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.349
0 - acc: 0.7333Epoch 00014: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3504 - a
cc: 0.7334 - val_loss: 1.2970 - val_acc: 0.7593
Epoch 16/20
6580/6680 [============================>.] - ETA: 0s - loss: 1.372
1 - acc: 0.7269Epoch 00015: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3684 - a
cc: 0.7271 - val_loss: 1.3789 - val_acc: 0.7377
Epoch 17/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.349
1 - acc: 0.7185Epoch 00016: val_loss improved from 1.23632 to 1.22
077, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [==============================] - 1s - loss: 1.3476 - a
cc: 0.7192 - val_loss: 1.2208 - val_acc: 0.7222
Epoch 18/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.342
3 - acc: 0.7302Epoch 00017: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3397 - a
cc: 0.7307 - val_loss: 1.2968 - val_acc: 0.7581
Epoch 19/20
6600/6680 [============================>.] - ETA: 0s - loss: 1.363
6 - acc: 0.7286Epoch 00018: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3594 - a
cc: 0.7292 - val_loss: 1.3643 - val_acc: 0.7329
Epoch 20/20
6500/6680 [============================>.] - ETA: 0s - loss: 1.321
5 - acc: 0.7323Epoch 00019: val_loss did not improve
6680/6680 [==============================] - 1s - loss: 1.3213 - a
```

```
cc: 0.7326 - val_loss: 1.4948 - val_acc: 0.7389
```

Out[31]: <keras.callbacks.History at 0x10ef01c18>

Load the Model with the Best Validation Loss

In [32]:
```
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 [33]:
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(f
eature, axis=0))) for feature in test_VGG16]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG16_predictions)==np.argmax(t
est_targets, axis=1))/len(VGG16_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 71.5311%
```

Predict Dog Breed with the Model

In [34]:
```
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 [35]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.
import numpy as np
bottleneck_features = np.load('bottleneck_features/DogVGG19Data.npz
')
train_VGG19 = bottleneck_features['train']
valid_VGG19 = bottleneck_features['valid']
test_VGG19 = 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:

For my final CNN architecture, I used the top layers from the pre-trained model and added Global Average Pooling layer for spatial reduction, a full dense layer to capture information from my set of data, and a drop out layer to avoid overfitting and another full dense layer to connect the information train with the output classes.

This architecture make good use of the pre-trained model to capture the common features and replace the final layers to capture the fine details of the set of images under study to attain much better accuracy than creating a CNN to classify dog breeds from scratch. By making use of transfer learning's pre-trained model top layers, much better accuracy can be obtained within limited computation resources and short training time.

In [36]:
```
### TODO: Define your architecture.
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling
2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras import regularizers
VGG19_model = Sequential()
VGG19_model.add(GlobalAveragePooling2D(input_shape=train_VGG19.shap
e[1:]))
VGG19_model.add(Dense(150, activation='relu', kernel_regularizer=re
gularizers.l2(0.005)))
VGG19_model.add(Dropout(0.4))

VGG19_model.add(Dense(133, activation='softmax'))

VGG19_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_4 ( (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 150)               76950
_________________________________________________________________
dropout_3 (Dropout)          (None, 150)               0
_________________________________________________________________
dense_7 (Dense)              (None, 133)               20083
=================================================================
Total params: 97,033.0
Trainable params: 97,033.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [37]:
```
### TODO: Compile the model.
VGG19_model.compile(loss='categorical_crossentropy', optimizer='rms
prop', metrics=['accuracy'])
```

(IMPLEMENTATION) Train the Model

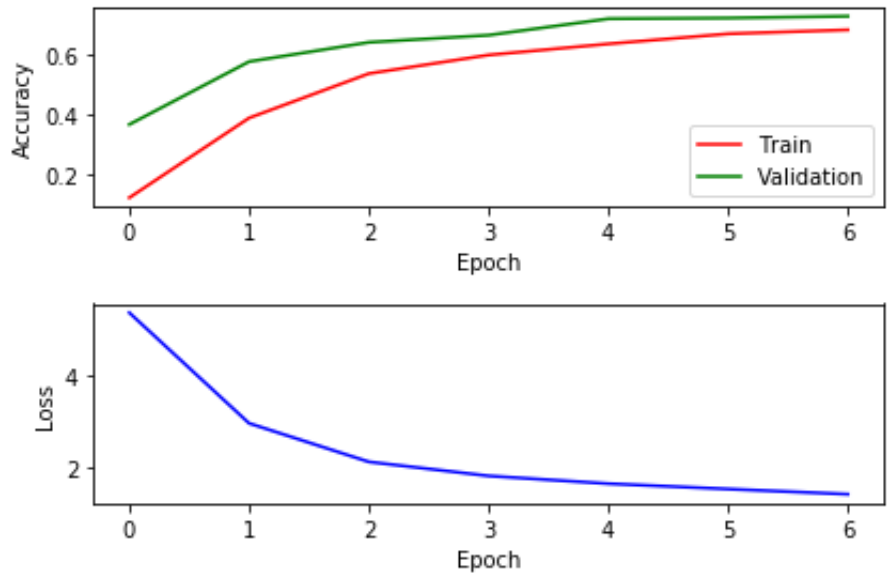
Train your model in the code cell below. Use model checkpointing to save the model that attains the best validation loss.

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 [38]:
```
### TODO: Train the model.
from keras.callbacks import ModelCheckpoint

epochs=7

checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.
VGG19.hdf5',
                               verbose=1, save_best_only=True)

VGG19_hist = VGG19_model.fit(train_VGG19, train_targets,
          validation_data=(valid_VGG19, valid_targets),
          epochs=epochs, batch_size=20, callbacks=[checkpointer], v
erbose=1)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/7
6480/6680 [============================>.] - ETA: 0s - loss: 5.443
6 - acc: 0.1157   Epoch 00000: val_loss improved from inf to 3.124
10, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 2s - loss: 5.3901 - a
cc: 0.1198 - val_loss: 3.1241 - val_acc: 0.3653
Epoch 2/7
6400/6680 [===========================>..] - ETA: 0s - loss: 2.972
9 - acc: 0.3850Epoch 00001: val_loss improved from 3.12410 to 1.86
865, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 2.9595 - a
cc: 0.3871 - val_loss: 1.8687 - val_acc: 0.5760
Epoch 3/7
6380/6680 [===========================>..] - ETA: 0s - loss: 2.127
6 - acc: 0.5321Epoch 00002: val_loss improved from 1.86865 to 1.53
124, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 2.1122 - a
cc: 0.5359 - val_loss: 1.5312 - val_acc: 0.6407
Epoch 4/7
6380/6680 [===========================>..] - ETA: 0s - loss: 1.802
5 - acc: 0.5976Epoch 00003: val_loss improved from 1.53124 to 1.41
188, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 1.8028 - a
cc: 0.5982 - val_loss: 1.4119 - val_acc: 0.6647
Epoch 5/7
6660/6680 [============================>.] - ETA: 0s - loss: 1.633
2 - acc: 0.6356Epoch 00004: val_loss improved from 1.41188 to 1.21
550, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 1.6327 - a
cc: 0.6355 - val_loss: 1.2155 - val_acc: 0.7198
Epoch 6/7
6360/6680 [===========================>..] - ETA: 0s - loss: 1.516
3 - acc: 0.6701Epoch 00005: val_loss improved from 1.21550 to 1.19
917, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 1.5143 - a
cc: 0.6693 - val_loss: 1.1992 - val_acc: 0.7222
Epoch 7/7
6380/6680 [===========================>..] - ETA: 0s - loss: 1.403
0 - acc: 0.6829Epoch 00006: val_loss improved from 1.19917 to 1.18
150, saving model to saved_models/weights.best.VGG19.hdf5
6680/6680 [==============================] - 1s - loss: 1.4009 - a
cc: 0.6825 - val_loss: 1.1815 - val_acc: 0.7281
```

In [39]:
```
# Reference https://github.com/betulays/dog-project-udacity.git
# Below code for showing the training and validation accuracy is fr
om the above link.

import matplotlib.pyplot as plt
plt.subplot(211)
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.plot(VGG19_hist.history["acc"], color="r", label="Train")
plt.plot(VGG19_hist.history["val_acc"], color="g", label="Validatio
n")
plt.legend(loc="best")

plt.subplot(212)
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.plot(VGG19_hist.history["loss"], color="b", label="Train")

plt.tight_layout()
plt.show()
```



(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [40]:
```
### TODO: Load the model weights with the best validation loss.
VGG19_model.load_weights('saved_models/weights.best.VGG19.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 [41]:
```
### TODO: Calculate classification accuracy on the test dataset.
VGG19_predictions = [np.argmax(VGG19_model.predict(np.expand_dims(f
eature, axis=0))) for feature in test_VGG19]

# report test accuracy
test_accuracy = 100*np.sum(np.array(VGG19_predictions)==np.argmax(t
est_targets, axis=1))/len(VGG19_predictions)
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 73.5646%
```

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

from extract_bottleneck_features import *
from keras.applications.vgg19 import VGG19, preprocess_input
def VGG19_predict_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_VGG19(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = VGG19_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]
```

In [43]:
```
VGG19_predict_breed("dogimages/test/001.Affenpinscher/Affenpinscher_00023.jpg")
```

Out[43]:
```
'Affenpinscher'
```

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!

Sample Human Output

(IMPLEMENTATION) Write your Algorithm

In [44]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
### Ref:  Ref https://github.com/betulays/dog-project-udacity.git
### Below code is modified from the above reference link and replaced the CNN with my own CNN from step 5.
### Since there may be dog and human in the same photo, so I have changed the criteria as below.


def dog_breed_detector(img_path):
    if dog_detector(img_path) == True:
        print("A dog is detected!")
        print("The dog breed is ...")
        return VGG19_predict_breed(img_path)
    if face_detector(img_path) == True:
        print("A human is detected!")
        print("The human looks like a: ")
        return VGG19_predict_breed(img_path)
    else:
        print("No dog or human face is detected!")
```

Step 7: Test Your Algorithm

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

(IMPLEMENTATION) Test Your Algorithm on Sample Images!

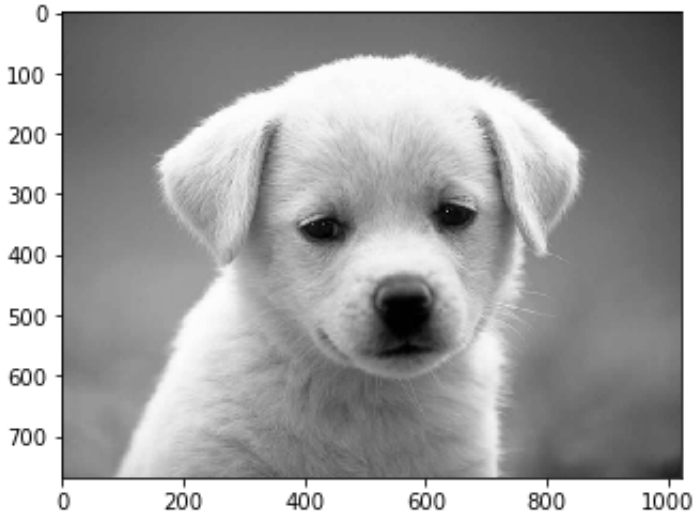
Test your algorithm at least six images on your computer. Feel free to use any images you like. Use at least two human and two dog images.

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

Answer: The output is worse than what I expected. The electricity tree is detected as a human. To improve it, below lists the possible points.

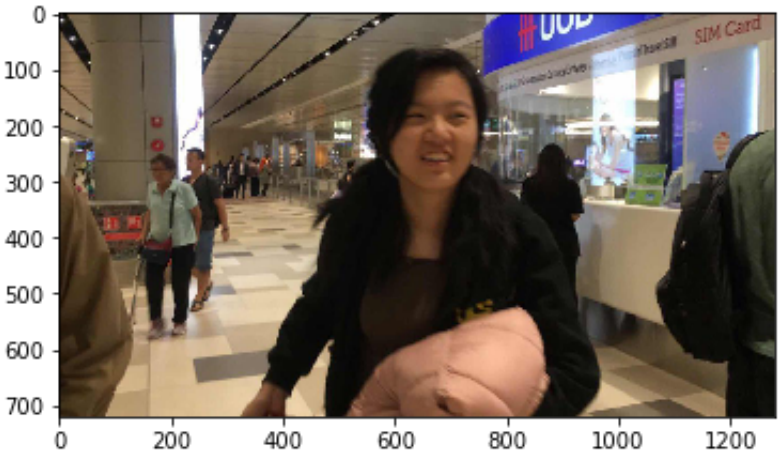
1. Replace the human_detector with a better human detector
2. Use transfer training to train a better human detector
3. Use augmented images include both human and trees and other objects to train the human detector

In [45]:
```
## 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.
## Ref https://github.com/betulays/dog-project-udacity.git
## Below code is modified from the above reference link. The above
link provides a good way get file names in a directory.


import cv2
from os import listdir
from os.path import join
images_path= 'my_Images/'
imageNames = listdir(images_path)

image_number =1;
for imagename in imageNames:
    if imagename.endswith(".jpg"):
        img = cv2.imread(join(images_path,imagename))
        cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        plt.imshow(cv_rgb)
        plt.show()
        print("==================", image_number, imagename, "=====
=============")
        print(dog_breed_detector(join(images_path,imagename)))
        image_number = image_number +1
```

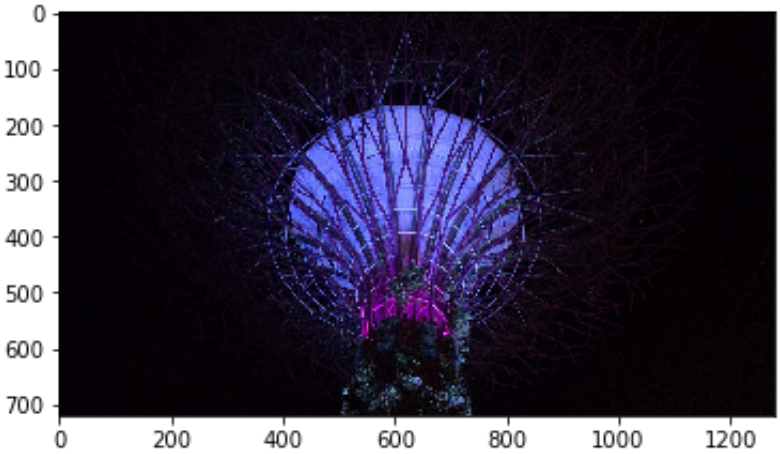



```
================== 1 dog.jpg ==================
A dog is detected!
The dog breed is ...
Labrador_retriever
```

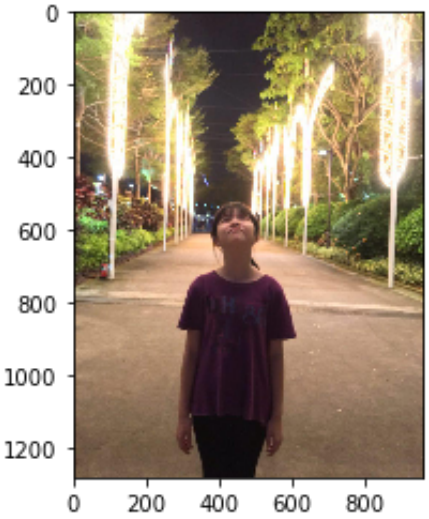

```
================== 2 1600px-Lava_the_sled_dog.jpg ================
==
A dog is detected!
The dog breed is ...
Alaskan_malamute
```



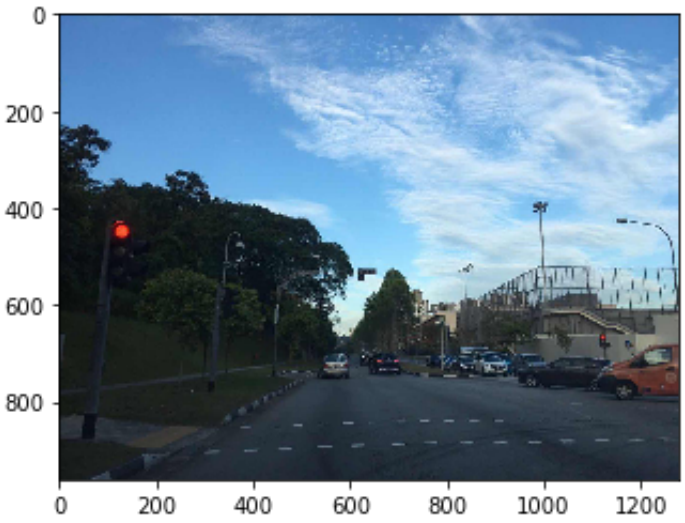

```
================== 3 anna.jpg ==================
A human is detected!
The human looks like a:
Dogue_de_bordeaux
```



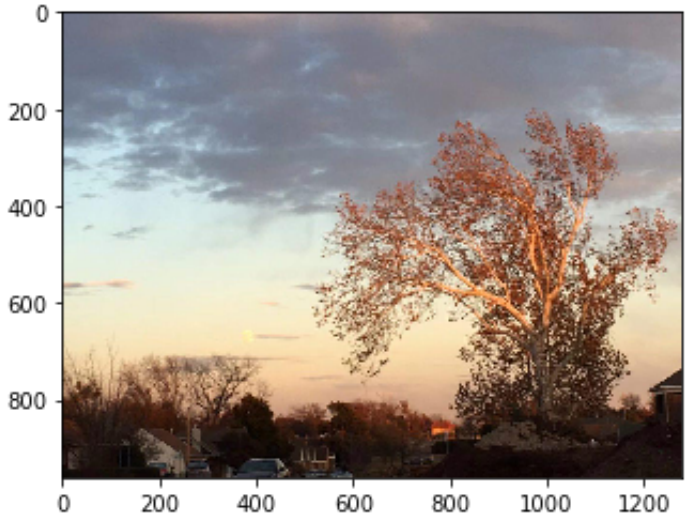
```
================== 4 ele_tree.jpg ==================
A human is detected!
The human looks like a:
Yorkshire_terrier
```



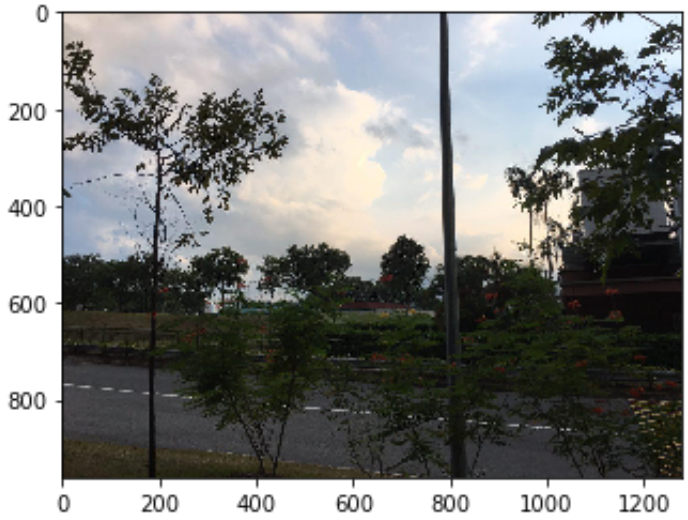

```
================== 5 dian.jpg ==================
A human is detected!
The human looks like a:
Affenpinscher
```



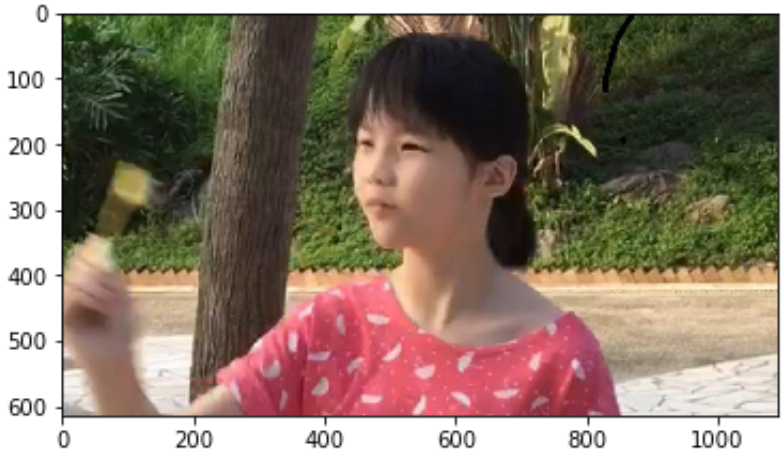

```
================== 6 sky.jpg ==================
No dog or human face is detected!
None
```




```
================== 7 tree.jpg ==================
No dog or human face is detected!
None
```




```
================== 8 trees.jpg ==================
No dog or human face is detected!
None
```



```
================== 9 dian3.jpg ==================
A human is detected!
The human looks like a:
German_pinscher
```


```
================== 10 dian2.jpg ==================
A human is detected!
The human looks like a:
German_pinscher
```

In [ ]: