Deep Learning 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_files, valid_f
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.
C:\Users\Arpit\Anaconda3\envs\dog-project\lib\site-packages\h5py\__init__.py:
34: FutureWarning: Conversion of the second argument of issubdtype from `floa
t` to `np.floating` is deprecated. In future, it will be treated as `np.float
64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters

There are 133 total dog categories.
There are 8351 total dog images.

There are 6680 training dog images.
There are 835 validation dog images.
There are 836 test dog images.
```

Import Human Dataset

In the code cell below, we import a dataset of human images, where the file paths are stored in the numpy array `human_files`.

In [2]:
```
import random
random.seed(8675309)

# load filenames in shuffled human dataset
human_files = np.array(glob("lfw/*/*"))
random.shuffle(human_files)

# print statistics about the dataset
print('There are %d total human images.' % len(human_files))
```

```
There are 13233 total human images.
```

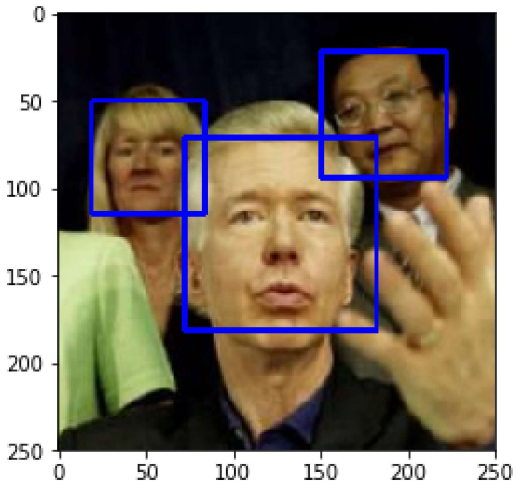
Step 1: Detect Humans

We use OpenCV's implementation of Haar feature-based cascade classifiers (http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html) to detect human faces in images. OpenCV provides many pre-trained face detectors, stored as XML files on github (https://github.com/opencv/opencv/tree/master/data/haarcascades). We have downloaded one of these detectors and stored it in the `haarcascades` directory.

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

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

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

# load color (BGR) image
img = cv2.imread(human_files[12600])
# 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: 3
```



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: Here, the Average Accuracy of detecting human face is about 0.99 Since 99% of human face were detected in the 'human files' and Average Accuracy of detecting Dog are 0.11 Therefore 11% of human face was detected in 'dog files'. So this is not perfect. It is very high accuracy on detecting human face when it exists. But false positive rate is high - 11% in detecting human face when none exists in 'dog files' So, classification of human in 'dog file' is false detection.

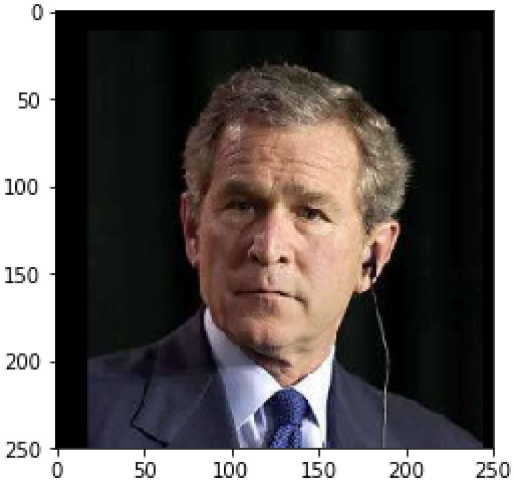
- Percentage of the first 100 images in `human_files` have a detected human face is 99%
- Percentage of the first 100 images in `dog_files` have a detected human face is 11%

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
human_files_detected_human = [face_detector(img) for img in human_files_short]
dog_files_detected_human = [face_detector(img) for img in dog_files_short]
## Average
avg_human_files_detected_human = np.average([face_detector(img) for img in hum
avg_dog_files_detected_human = np.average([face_detector(img) for img in dog_f

## on the images in human_files_short and dog_files_short.
print ('percentage of human face in human files is : {}%'.format(sum(human_fil
print ('percentage of human face in dog files is : {}%'.format(sum(dog_files_d
print ("Average accuracy of human = {}, Average accuracy of Dog = {}".format(a
```

```
percentage of human face in human files is : 99%
percentage of human face in dog files is : 11%
Average accuracy of human = 0.99, Average accuracy of Dog = 0.11
```

In [6]:
```
## (Optional) TODO: Report the performance of another
## face detection algorithm on the LFW dataset
### Feel free to use as many code cells as needed.

## Display the selected image of George Walker Bush.
img = cv2.imread(human_files[5])

#Print the file location:
print("location where this image file is located : ", human_files_short[5])

# check the status of face_detector:
print("Status of Human detected : ", face_detector(human_files_short[5]))

## 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()
```

```
location where this image file is located :  lfw\George_W_Bush\George_W_Bush_
0399.jpg
Status of Human detected :  True
```



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: Here, it is quite rational that users input a direct image of their face since the point is to determine what type of Creature (human or dog or Alien) they look like. It would be hard to determine that if we can't see their face. However, if you look at the fifth image of the human data set, George Walker Bush face is visible, And it is nicely predicted. The face detector is able to detect it as a face, but the face is visible enough that you could make a prediction at what type of creature (human or dog or Alien) he looks like. The detector should be able to handle that such type of image. Looking at the dog data, we see that there are face detections in 11% of the data. Looking through some of the dog data I was able to find a few images where there

were humans with the dogs. We just uses a face detector to check if it is an image of a human then it is going to say that any image with a human in it is an image of a human. Based on all of this if our goal is to make a human detector I would say a better approach would be to just use a pretrained network to detect if a human is in the image. So still my answer is No it is not a reasonable expectation to have humans clearly present their face in a photo for it to be detected. The alternative is to use deep learning to extract features that distinguish humans from dogs even if full face is not presented.

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.

Step 2: Detect Dogs

In this section, we use a pre-trained ResNet-50 (http://ethereon.github.io/netscope/#/gist/db945b393d40bfa26006) model to detect dogs in images. Our first line of code downloads the ResNet-50 model, along with weights that have been trained on ImageNet (http://www.image-net.org/), a very large, very popular dataset used for image classification and other vision tasks. ImageNet contains over 10 million URLs, each linking to an image containing an object from one of 1000 categories (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a). Given an image, this pre-trained ResNet-50 model returns a prediction (derived from the available categories in ImageNet) for the object that is contained in the image.

In [7]:
```
from keras.applications.resnet50 import ResNet50

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

Pre-process the Data

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape

$$(\text{nb\_samples}, \text{rows}, \text{columns}, \text{channels}),$$

where `nb_samples` corresponds to the total number of images (or samples), and `rows`, `columns`, and `channels` correspond to the number of rows, columns, and channels for each image, respectively.

The `path_to_tensor` function below takes a string-valued file path to a color image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The function first loads the image and resizes it to a square image that is 224×224 pixels. Next, the image is converted to an array, which is then resized to a 4D tensor. In this case, since we are working with color images, each image has three channels. Likewise, since we are processing a single image (or sample), the returned tensor will always have shape

$$(1, 224, 224, 3).$$

The `paths_to_tensor` function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape

$$(\text{nb\_samples}, 224, 224, 3).$$

Here, `nb_samples` is the number of samples, or number of images, in the supplied array of image paths. It is best to think of `nb_samples` as the number of 3D tensors (where each 3D tensor corresponds to a different image) in your dataset!

In [8]:
```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths
    return np.vstack(list_of_tensors)
```

Making Predictions with ResNet-50

Getting the 4D tensor ready for ResNet-50, and for any other pre-trained model in Keras, requires some additional processing. First, the RGB image is converted to BGR by reordering the channels. All pre-trained models have the additional normalization step that the mean pixel (expressed in RGB as $[103.939, 116.779, 123.68]$ and calculated from all pixels in all images in ImageNet) must be subtracted from every pixel in each image. This is implemented in the imported function `preprocess_input`. If you're curious, you can check the code for `preprocess_input` here (https://github.com/fchollet/keras/blob/master/keras/applications/imagenet_utils.py).

Now that we have a way to format our image for supplying to ResNet-50, we are now ready to use the model to extract the predictions. This is accomplished with the `predict` method, which returns an array whose i-th entry is the model's predicted probability that the image belongs to the i-th ImageNet category. This is implemented in the `ResNet50_predict_labels` function below.

By taking the argmax of the predicted probability vector, we obtain an integer corresponding to the model's predicted object class, which we can identify with an object category through the use of this dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a).

In [9]:
```
from keras.applications.resnet50 import preprocess_input, decode_predictions

def ResNet50_predict_labels(img_path):
    ## returns prediction vector for image located at img_path
    img = preprocess_input(path_to_tensor(img_path))
    return np.argmax(ResNet50_model.predict(img))
```

Write a Dog Detector

While looking at the dictionary (https://gist.github.com/yrevar/942d3a0ac09ec9e5eb3a), you will notice that the categories corresponding to dogs appear in an uninterrupted sequence and correspond to dictionary keys 151-268, inclusive, to include all categories from `'Chihuahua'` to `'Mexican hairless'`. Thus, in order to check to see if an image is predicted to contain a dog by the pre-trained ResNet-50 model, we need only check if the `ResNet50_predict_labels` function above returns a value between 151 and 268 (inclusive).

We use these ideas to complete the `dog_detector` function below, which returns `True` if a dog is detected in an image (and `False` if not).

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

(IMPLEMENTATION) Assess the Dog Detector

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

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

Answer:

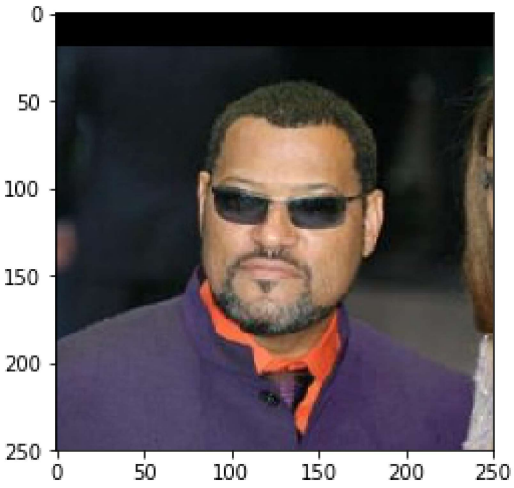
- Percentage of the images in `human_files_short` have a detected dog is 1%
- Percentage of the images in `human_files_short` have a detected dog is 100%

In [11]:
```
### TODO: Test the performance of the dog_detector function.
human_files_detected_dog = np.sum([dog_detector(img) for img in human_files_sh
dog_files_detected_dog = np.sum([dog_detector(img) for img in dog_files_short]

### on the images in human_files_short and dog_files_short.
print ("{} % of images in human_files_short detected a dog.".format(human_file
print ("{} % of images in dog_files_short detected a dog.".format(dog_files_de
```

```
1 % of images in human_files_short detected a dog.
100 % of images in dog_files_short detected a dog.
```

In [12]:
```
## Display the selected image of Laurence Fishburne.
img = cv2.imread(human_files[3])

#Print the file location:
print("Location where this image file is located : ", human_files_short[3])

# check the status of dog_detector:
print("Status of Dog Detector : ", dog_detector(human_files_short[3]))

#ResNet50_predict_labels, print prediction:
prediction = ResNet50_predict_labels(human_files_short[3])
print(prediction)

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

## Display the Image, along with the bounding box
plt.imshow(cv_rgb)
plt.show()
```

```
Location where this image file is located :  lfw\Laurence_Fishburne\Laurence_
Fishburne_0001.jpg
Status of Dog Detector :  False
643
```



Dog detector classify Laurence Fishburne is not a dog. Looking at the prediction from the ResNet model, it's classifying him as a human. Not really seeing it, but the dog detector only has 1% error so it's not doing too bad.

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 [13]:
```
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 [00:39<00:00, 170.88it/s]
100%|██████████| 835/835 [00:04<00:00, 188.00it/s]
100%|██████████| 836/836 [00:04<00:00, 190.26it/s]
```

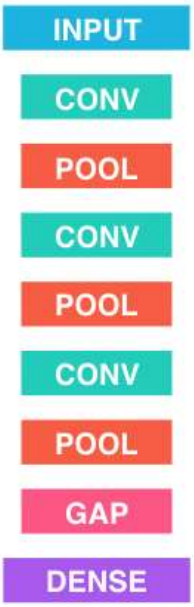
(IMPLEMENTATION) Model Architecture

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

```
model.summary()
```

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

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 223, 223, 16)      208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 111, 111, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 110, 110, 32)      2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 55, 55, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 54, 54, 64)        8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 27, 27, 64)        0
_________________________________________________________________
global_average_pooling2d_1 ( (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 133)               8645
=================================================================
Total params: 19,189.0
Trainable params: 19,189.0
Non-trainable params: 0.0
_________________________________________________________________
```



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

Answer: I decided to chose an architecture that have 4 sets of convolution and max pooling layers. These layers will help the model extract more and more complex features to understand the differences b/w different dog breeds. Next I have added a layer to flatten output but but with a Batch Normalization layer to normalize the output from the GAP layer. The single output layer performed about as good as when adding other dense layers, but took less time to train and was less prone to overfitting. Finally I have an output layer with softmax activation to classify the output into 133 categories

In [55]:
```
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense, BatchNormalization
from keras.models import Sequential


### TODO: Define your architecture.
model = Sequential()

## Convolutional layer-1:
model.add(Conv2D(32, 3, strides=(1,1), padding='same', activation='relu', inpu
model.add(MaxPooling2D((2,2), strides= 2, padding='same'))

## Convolutional layer-2:
model.add(Conv2D(64, 3, strides=(1,1), padding='same', activation='relu', inpu
model.add(MaxPooling2D((2,2), strides= 2, padding='same'))

## Convolutional layer-3:
model.add(Conv2D(128, 3, strides=(1,1), padding='same', activation='relu', inp
model.add(MaxPooling2D((2,2), strides= 2, padding='same'))
model.add(Dropout(0.2))

## Convolutional layer-4:
model.add(Conv2D(128, 3, strides=(1,1), padding='same', activation='relu', inp
model.add(MaxPooling2D((2,2), strides= 2, padding='same'))


## flatten Layer:
model.add(GlobalAveragePooling2D())
model.add(BatchNormalization())
model.add(Dense(133, kernel_initializer='he_normal', bias_initializer='zeros',
model.add(Dropout(0.5))

## fully Connected Layer:
model.add(Dense(133, activation='softmax'))


## Model Summary:
model.summary()
```

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 224, 224, 32)	896
max_pooling2d_24 (MaxPooling	(None, 112, 112, 32)	0
conv2d_7 (Conv2D)	(None, 112, 112, 64)	18496
max_pooling2d_25 (MaxPooling	(None, 56, 56, 64)	0
conv2d_8 (Conv2D)	(None, 56, 56, 128)	73856
max_pooling2d_26 (MaxPooling	(None, 28, 28, 128)	0
dropout_3 (Dropout)	(None, 28, 28, 128)	0
conv2d_9 (Conv2D)	(None, 28, 28, 128)	147584

In [15]:
```
model.save_weights('saved_models/weights.initial_scratch_model.hdf5')
```

Compile the Model

In [16]:
```
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['
```

(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 [17]:
```
from keras.callbacks import ModelCheckpoint

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

epochs = 10


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

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

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

```
Train on 6680 samples, validate on 835 samples
Epoch 1/10
6660/6680 [============================>.] - ETA: 329s - loss: 4.8857 - ac
c: 0.0000e+0 - ETA: 281s - loss: 4.8746 - acc: 0.0000e+0 - ETA: 266s - los
s: 4.9252 - acc: 0.0000e+0 - ETA: 258s - loss: 4.9301 - acc: 0.0000e+0 - ET
A: 252s - loss: 4.9195 - acc: 0.0000e+0 - ETA: 249s - loss: 4.9143 - acc:
0.0000e+0 - ETA: 246s - loss: 4.9132 - acc: 0.0000e+0 - ETA: 243s - loss:
4.9094 - acc: 0.0125    - ETA: 241s - loss: 4.9095 - acc: 0.011 - ETA: 240s
- loss: 4.9069 - acc: 0.010 - ETA: 238s - loss: 4.9058 - acc: 0.009 - ETA:
237s - loss: 4.9038 - acc: 0.008 - ETA: 235s - loss: 4.9025 - acc: 0.007 -
ETA: 234s - loss: 4.9021 - acc: 0.007 - ETA: 233s - loss: 4.9007 - acc: 0.0
06 - ETA: 232s - loss: 4.9001 - acc: 0.006 - ETA: 231s - loss: 4.8996 - ac
c: 0.005 - ETA: 230s - loss: 4.8987 - acc: 0.005 - ETA: 229s - loss: 4.8981
- acc: 0.005 - ETA: 228s - loss: 4.8976 - acc: 0.005 - ETA: 228s - loss: 4.
8968 - acc: 0.004 - ETA: 227s - loss: 4.8965 - acc: 0.004 - ETA: 227s - los
s: 4.8940 - acc: 0.008 - ETA: 226s - loss: 4.8996 - acc: 0.010 - ETA: 225s
- loss: 4.8989 - acc: 0.010 - ETA: 225s - loss: 4.8998 - acc: 0.009 - ETA:
224s - loss: 4.8996 - acc: 0.009 - ETA: 223s - loss: 4.8993 - acc: 0.008 -
ETA: 223s - loss: 4.8989 - acc: 0.010 - ETA: 222s - loss: 4.8980 - acc: 0.0
```

DATA AUGMENTATION

In [18]:
```
import os
from keras.preprocessing.image import ImageDataGenerator
training_generator = ImageDataGenerator(rescale=1.,
                                        rotation_range=15.0,
                                        width_shift_range=0.1,
                                        height_shift_range=0.1,
                                        shear_range=0.1,
                                        zoom_range=0.1,
                                        horizontal_flip=True,
                                        vertical_flip=False,
                                        fill_mode="reflect")

validation_generator = ImageDataGenerator(rescale=1)

training_generator.fit(train_tensors)
validation_generator.fit(valid_tensors)

training_data = training_generator.flow(train_tensors, train_targets, batch_si
validation_data = validation_generator.flow(valid_tensors, valid_targets, batc
```

Test the Model

Try out your model on the test dataset of dog images. Ensure that your test accuracy is greater than 1%.

In [19]:
```
model.load_weights('saved_models/weights.initial_scratch_model.hdf5')
```

In [20]:
```
## 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
model.load_weights('saved_models/weights.best.from_scratch.hdf5')
dog_breed_predictions_DataAug = [np.argmax(model.predict(np.expand_dims(tensor

## Report Test accuracy:
test_accuracy = 100*np.sum(np.array(dog_breed_predictions)==np.argmax(test_tar
print('Test accuracy: %.4f%%' % test_accuracy)

# Report test accuracy with Data Augmentation
test_accuracy = 100*np.sum(np.array(dog_breed_predictions_DataAug)==np.argmax(
print('Test accuracy with Data Augmentation: %.4f%%' % test_accuracy)
```

```
Test accuracy: 0.5981%
Test accuracy with Data Augmentation: 5.3828%
```

So adding Data Augmentation increases the accuracy, however the training time increases from ~ 23s to ~63s per epoch. The training accuracy using data augmentation is much less after 10 epochs, 32% vs 40% without data augmentation. This indicates that the model is more resistant to overtraining and generalizes better.

If the models were trained longer, the model with data augmentation would most likely continue to improve while the model without data augmentation would begin to overfit.

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 [21]:
```
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 [22]:
```
VGG16_model = Sequential()
VGG16_model.add(GlobalAveragePooling2D(input_shape=train_VGG16.shape[1:]))
VGG16_model.add(Dense(133, activation='softmax'))

VGG16_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
global_average_pooling2d_2 ( (None, 512)               0         
_________________________________________________________________
dense_3 (Dense)              (None, 133)               68229     
=================================================================
Total params: 68,229.0
Trainable params: 68,229.0
Non-trainable params: 0.0
_________________________________________________________________
```

Compile the Model

In [23]:
```
VGG16_model.compile(loss='categorical_crossentropy', optimizer='rmsprop', metr
```

Train the Model

In [24]:
```
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=30, batch_size=20, callbacks=[checkpointer], verbose=1);
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/30
6420/6680 [===========================>..] - ETA: 171s - loss: 15.5937 - ac
c: 0.0000e+ - ETA: 11s - loss: 14.8590 - acc: 0.0000e+00 - ETA: 6s - loss:
14.5034 - acc: 0.0113    - ETA: 4s - loss: 14.3413 - acc: 0.01 - ETA: 3s -
loss: 14.1766 - acc: 0.02 - ETA: 2s - loss: 13.9502 - acc: 0.02 - ETA: 2s -
loss: 13.6914 - acc: 0.03 - ETA: 1s - loss: 13.5505 - acc: 0.04 - ETA: 1s -
loss: 13.3916 - acc: 0.05 - ETA: 1s - loss: 13.1653 - acc: 0.06 - ETA: 1s -
loss: 13.0423 - acc: 0.07 - ETA: 1s - loss: 12.8527 - acc: 0.07 - ETA: 0s -
loss: 12.7513 - acc: 0.08 - ETA: 0s - loss: 12.5745 - acc: 0.09 - ETA: 0s -
loss: 12.4666 - acc: 0.10 - ETA: 0s - loss: 12.3234 - acc: 0.10 - ETA: 0s -
loss: 12.2300 - acc: 0.11 - ETA: 0s - loss: 12.1546 - acc: 0.11 - ETA: 0s -
loss: 12.0620 - acc: 0.12 - ETA: 0s - loss: 11.9533 - acc: 0.13 - ETA: 0s -
loss: 11.8885 - acc: 0.13 - ETA: 0s - loss: 11.8150 - acc: 0.14 - ETA: 0s -
loss: 11.7772 - acc: 0.1444Epoch 00000: val_loss improved from inf to 10.34
426, saving model to saved_models/weights.best.VGG16.hdf5
6680/6680 [============================== - 1s - loss: 11.7297 - acc: 0.1472
- val_loss: 10.3443 - val_acc: 0.2228
Epoch 2/30
```

Load the Model with the Best Validation Loss

In [25]:
```
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 [26]:
```
# get index of predicted dog breed for each image in test set
VGG16_predictions = [np.argmax(VGG16_model.predict(np.expand_dims(feature, axi

# report test accuracy
test_accuracy = 100 * np.sum(np.array(VGG16_predictions)==np.argmax(test_targe
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 46.7703%
```

Predict Dog Breed with the Model

In [27]:
```
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.np
z')
train_{network} = bottleneck_features['train']
valid_{network} = bottleneck_features['valid']
test_{network} = bottleneck_features['test']
```

In [28]:
```
### TODO: Obtain bottleneck features from another pre-trained CNN.
from keras.layers import Conv2D, MaxPooling2D, GlobalAveragePooling2D
from keras.layers import Dropout, Flatten, Dense
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization


bottleneck_features = np.load('bottleneck_features/DogResnet50Data.npz')
train_DResnet50= bottleneck_features['train']
valid_DResnet50 = bottleneck_features['valid']
test_DResnet50 = 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:

I tested Resnet, VGG19 and the Inception model and found the performance to be much better for the Resnet model. Once I select the model, then I follow the steps with the VGG16 model given above. I add a global average pooling layer,a dense layer with Relu activation and a dense layer with Softmax activation to classify the output into one of 133 categories. With this architecture I was able to get 84% accuracy. I trained the model for 8 epochs. I found losses stabilized after 2-3 epochs and I picked the model with the lowest validation loss.

In [29]:
```
### TODO: Define your architecture.

DResnet50_model = Sequential()
DResnet50_model.add(GlobalAveragePooling2D(input_shape=train_DResnet50.shape[1
#DResnet50_model.add(BatchNormalization())

DResnet50_model.add(Dense(133, activation='softmax'))
DResnet50_model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
global_average_pooling2d_3 ( (None, 2048)              0
_________________________________________________________________
dense_4 (Dense)              (None, 133)               272517
=================================================================
Total params: 272,517.0
Trainable params: 272,517.0
Non-trainable params: 0.0
_________________________________________________________________
```

(IMPLEMENTATION) Compile the Model

In [30]:
```
### TODO: Compile the model.
from keras.optimizers import Adam
DResnet50_model.compile(optimizer=Adam(), loss='categorical_crossentropy', met
```

(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 [31]:
```
### TODO: Train the model.
from keras.callbacks import ModelCheckpoint
checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.DResnet50.h
                               verbose=0, save_best_only=True)

DResnet50_model.fit(train_DResnet50, train_targets,
          validation_data=(valid_DResnet50, valid_targets),
          epochs=50, batch_size=20, callbacks=[checkpointer], verbose=2)
```

```
Train on 6680 samples, validate on 835 samples
Epoch 1/50
2s - loss: 1.6924 - acc: 0.5969 - val_loss: 0.8346 - val_acc: 0.7545
Epoch 2/50
2s - loss: 0.3585 - acc: 0.8996 - val_loss: 0.7365 - val_acc: 0.7737
Epoch 3/50
2s - loss: 0.1641 - acc: 0.9660 - val_loss: 0.6032 - val_acc: 0.8024
Epoch 4/50
2s - loss: 0.0855 - acc: 0.9891 - val_loss: 0.6107 - val_acc: 0.8072
Epoch 5/50
2s - loss: 0.0551 - acc: 0.9936 - val_loss: 0.5624 - val_acc: 0.8275
Epoch 6/50
2s - loss: 0.0391 - acc: 0.9967 - val_loss: 0.5849 - val_acc: 0.8132
Epoch 7/50
2s - loss: 0.0290 - acc: 0.9979 - val_loss: 0.5567 - val_acc: 0.8335
Epoch 8/50
2s - loss: 0.0331 - acc: 0.9946 - val_loss: 0.5747 - val_acc: 0.8347
Epoch 9/50
2s - loss: 0.0203 - acc: 0.9981 - val_loss: 0.5630 - val_acc: 0.8275
Epoch 10/50
2s - loss: 0.0215 - acc: 0.9963 - val_loss: 0.5858 - val_acc: 0.8311
Epoch 11/50
2s - loss: 0.0188 - acc: 0.9963 - val_loss: 0.5871 - val_acc: 0.8335
Epoch 12/50
2s - loss: 0.0258 - acc: 0.9955 - val_loss: 0.6753 - val_acc: 0.8144
Epoch 13/50
2s - loss: 0.0338 - acc: 0.9915 - val_loss: 0.7741 - val_acc: 0.7904
Epoch 14/50
2s - loss: 0.0296 - acc: 0.9951 - val_loss: 0.6358 - val_acc: 0.8240
Epoch 15/50
2s - loss: 0.0210 - acc: 0.9948 - val_loss: 0.7309 - val_acc: 0.8120
Epoch 16/50
2s - loss: 0.0564 - acc: 0.9849 - val_loss: 0.9023 - val_acc: 0.7988
Epoch 17/50
2s - loss: 0.0670 - acc: 0.9823 - val_loss: 0.9064 - val_acc: 0.7988
Epoch 18/50
2s - loss: 0.0225 - acc: 0.9937 - val_loss: 0.7909 - val_acc: 0.8204
Epoch 19/50
2s - loss: 0.0142 - acc: 0.9969 - val_loss: 0.7653 - val_acc: 0.8132
Epoch 20/50
2s - loss: 0.0088 - acc: 0.9982 - val_loss: 0.7329 - val_acc: 0.8204
Epoch 21/50
2s - loss: 0.0117 - acc: 0.9976 - val_loss: 0.7305 - val_acc: 0.8216
Epoch 22/50
2s - loss: 0.0115 - acc: 0.9978 - val_loss: 0.7549 - val_acc: 0.8204
Epoch 23/50
2s - loss: 0.0083 - acc: 0.9982 - val_loss: 0.7241 - val_acc: 0.8156
Epoch 24/50
2s - loss: 0.0112 - acc: 0.9981 - val_loss: 0.7230 - val_acc: 0.8299
Epoch 25/50
2s - loss: 0.0128 - acc: 0.9975 - val_loss: 0.7614 - val_acc: 0.8287
Epoch 26/50
2s - loss: 0.0508 - acc: 0.9843 - val_loss: 1.1204 - val_acc: 0.7713
Epoch 27/50
2s - loss: 0.0811 - acc: 0.9774 - val_loss: 1.1575 - val_acc: 0.7772
Epoch 28/50
2s - loss: 0.0307 - acc: 0.9910 - val_loss: 1.0442 - val_acc: 0.8072
```

```
Epoch 29/50
2s - loss: 0.0097 - acc: 0.9975 - val_loss: 0.9190 - val_acc: 0.8251
Epoch 30/50
2s - loss: 0.0070 - acc: 0.9988 - val_loss: 0.9242 - val_acc: 0.8216
Epoch 31/50
2s - loss: 0.0076 - acc: 0.9987 - val_loss: 0.9238 - val_acc: 0.8216
Epoch 32/50
2s - loss: 0.0100 - acc: 0.9979 - val_loss: 0.9349 - val_acc: 0.8240
Epoch 33/50
2s - loss: 0.0080 - acc: 0.9982 - val_loss: 0.9481 - val_acc: 0.8132
Epoch 34/50
2s - loss: 0.0176 - acc: 0.9952 - val_loss: 1.0529 - val_acc: 0.8048
Epoch 35/50
2s - loss: 0.0160 - acc: 0.9969 - val_loss: 0.9938 - val_acc: 0.8096
Epoch 36/50
2s - loss: 0.0217 - acc: 0.9948 - val_loss: 0.9903 - val_acc: 0.8060
Epoch 37/50
2s - loss: 0.0152 - acc: 0.9963 - val_loss: 0.9372 - val_acc: 0.8108
Epoch 38/50
2s - loss: 0.0098 - acc: 0.9984 - val_loss: 0.8876 - val_acc: 0.8132
Epoch 39/50
2s - loss: 0.0094 - acc: 0.9984 - val_loss: 1.0002 - val_acc: 0.8132
Epoch 40/50
2s - loss: 0.0100 - acc: 0.9976 - val_loss: 0.9091 - val_acc: 0.8192
Epoch 41/50
2s - loss: 0.0087 - acc: 0.9984 - val_loss: 0.9293 - val_acc: 0.8156
Epoch 42/50
2s - loss: 0.0080 - acc: 0.9987 - val_loss: 0.9543 - val_acc: 0.8108
Epoch 43/50
2s - loss: 0.0198 - acc: 0.9952 - val_loss: 1.1453 - val_acc: 0.8048
Epoch 44/50
2s - loss: 0.0689 - acc: 0.9822 - val_loss: 1.2164 - val_acc: 0.7952
Epoch 45/50
2s - loss: 0.0188 - acc: 0.9955 - val_loss: 1.1286 - val_acc: 0.8024
Epoch 46/50
2s - loss: 0.0160 - acc: 0.9967 - val_loss: 1.1280 - val_acc: 0.7880
Epoch 47/50
2s - loss: 0.0106 - acc: 0.9984 - val_loss: 1.0370 - val_acc: 0.8072
Epoch 48/50
2s - loss: 0.0149 - acc: 0.9984 - val_loss: 1.0076 - val_acc: 0.8072
Epoch 49/50
2s - loss: 0.0125 - acc: 0.9984 - val_loss: 1.0469 - val_acc: 0.8012
Epoch 50/50
2s - loss: 0.0083 - acc: 0.9988 - val_loss: 0.9653 - val_acc: 0.8120
```

Out[31]: `<keras.callbacks.History at 0x1be31618be0>`

(IMPLEMENTATION) Load the Model with the Best Validation Loss

In [32]:
```
### TODO: Load the model weights with the best validation loss.
DResnet50_model.load_weights('saved_models/weights.best.DResnet50.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 [33]:
```
### TODO: Calculate classification accuracy on the test dataset.

# get index of predicted dog breed for each image in test set
DResnet50_predictions = [np.argmax(DResnet50_model.predict(np.expand_dims(feat

# report test accuracy
test_accuracy = 100*np.sum(np.array(DResnet50_predictions)==np.argmax(test_tar
print('Test accuracy: %.4f%%' % test_accuracy)
```

```
Test accuracy: 83.3732%
```

Data generation

In [34]:
```
from keras.utils.np_utils import to_categorical
DResnet50_generator = ImageDataGenerator(rescale=1/255)
test_data_path = 'dogImages/test/'
batch_size = 32

DResnet50_test_data = DResnet50_generator.flow_from_directory(test_data_path,
                                                              target_size=(350, 3
                                                              batch_size=batch_si
                                                              shuffle=False,
                                                              class_mode='categor

test_samples = len(DResnet50_test_data.filenames)
test_labels = to_categorical(DResnet50_test_data.classes)
dog_names = [item[20:-1] for item in sorted(glob("dogImages/train/*/"))]
```

```
Found 836 images belonging to 133 classes.
```

(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 [35]:
```
from extract_bottleneck_features import *

def classify_Creature_breed(img_path):
    # extract bottleneck features
    bottleneck_feature = extract_Resnet50(path_to_tensor(img_path))
    # obtain predicted vector
    predicted_vector = DResnet50_model.predict(bottleneck_feature)
    # return dog breed that is predicted by the model
    return dog_names[np.argmax(predicted_vector)]

    # Use same Image Pipeline as used earlier
    img = cv2.imread(img_path)
    # Convert from BGR to RGB
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    # Plot the

    plt.imshow(cv_rgb)
    plt.show()
```

Step 6: Write your Algorithm

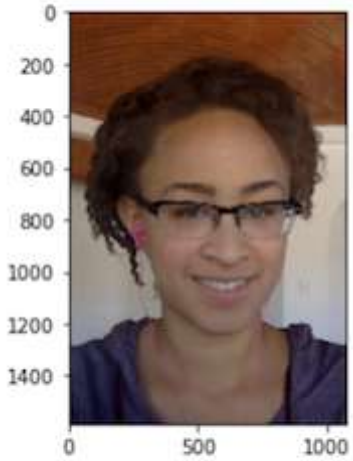
Write an algorithm that accepts a file path to an image and first determines whether the image contains a human, dog, or neither. Then,

- if a **dog** is detected in the image, return the predicted breed.
- if a **human** is detected in the image, return the resembling dog breed.
- if **neither** is detected in the image, provide output that indicates an error.

You are welcome to write your own functions for detecting humans and dogs in images, but feel free to use the `face_detector` and `dog_detector` functions developed above. You are **required** to use your CNN from Step 5 to predict dog breed.

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

```
hello, human!
```




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

(IMPLEMENTATION) Write your Algorithm

In [36]:
```
### TODO: Write your algorithm.
### Feel free to use as many code cells as needed.
# Use same Image Pipeline as used earlier

def Creature_breed(image_path):

    img = cv2.imread(image_path)
        # Convert from BGR to RGB
    cv_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
         # Plot the image

    if face_detector(image_path):
        print("Hello, human! ✋😎 <|o_o|>")
    elif dog_detector(image_path):
        print("Hello, Dog! 🐶 🐕")
    else:
        print("No human face or dog detected 🐶 👽 👹\n\n OMG! it is an Alien

    plt.imshow(cv_rgb)
    plt.show()

    breed = classify_Creature_breed(image_path)
    print("You look like a ...")
    print(breed)
    print()
```

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 of the model is pretty good. Better than I expected.😁😎 On images it has never seen below, it was able to correctly identify 9 of the 9 dogs, it was able to identify 6 of the 6 humans and correctly identify 2 of the 2 creatures who are neither human nor dog. However there is still scope for improvement.I have identified the following points for improvement:

The model prediction accuracy is not the same for all dog Breeds. For some classes it only has 3-4 dog images to train. Model can be made more accurate by adding more data so, Model will able to identified Dog breed accurately through input direct image otherwise, it works fine. During several runs of the algorithm, I found that the predicted dog category for one of the images changed. It would be good if model can also output the certainity it has with the prediction Different model architectures can be explored to reduce prediction time while maintaining accuracy.

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

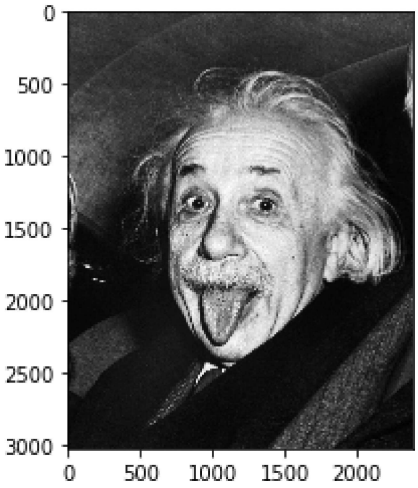
In [38]:
```
Creature_breed('images/img8.jpg')
```

```
Hello, human! ✋😎 <|o_o|>
```


```
You look like a ...
American_foxhound
```

In [39]:
```
Creature_breed('images/img4.jpg')
```

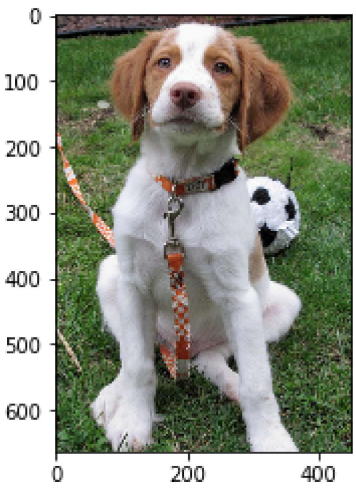
```
Hello, human! ✋😎 <|o_o|>
```




```
You look like a ...
Dogue_de_bordeaux
```

In [40]:
```
Creature_breed('images/img12.jpg')
```

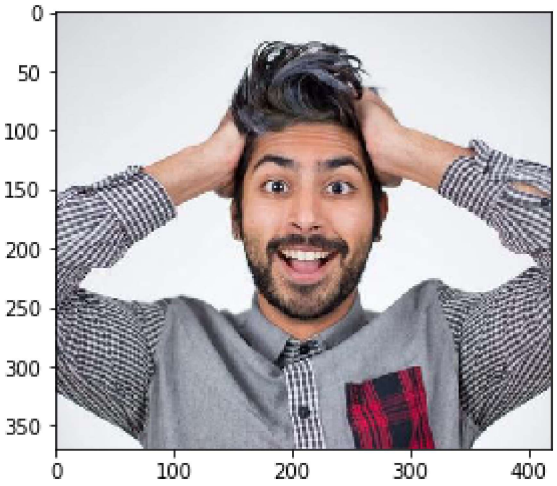
```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
Brittany
```

In [41]:
```
Creature_breed('images/img1.jpg')
```

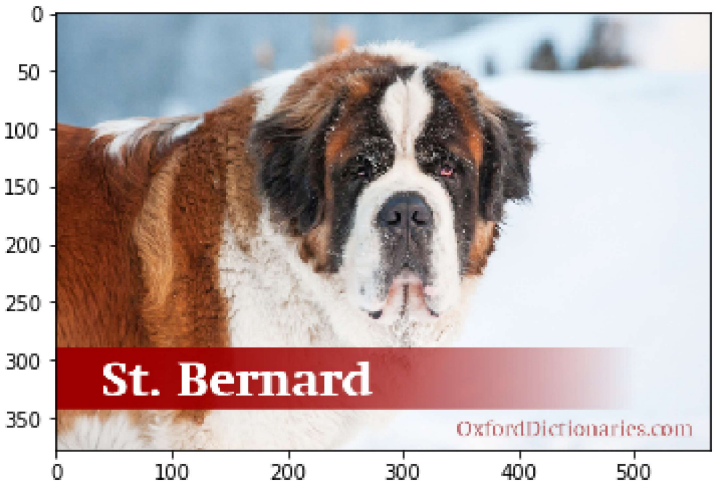
```
Hello, human! ✋😎 <|o_o|>
```



```
You look like a ...
Lowchen
```

In [42]:
```
Creature_breed('images/img5.png')
```

```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
Saint_bernard
```

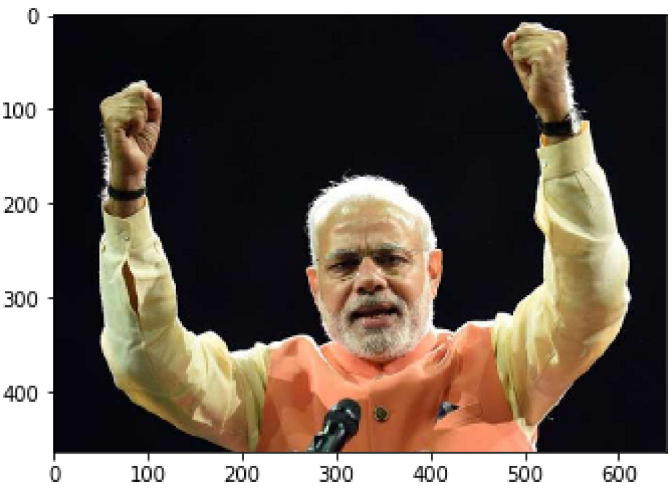
In [43]:
```
Creature_breed('images/img6.png')
```

```
Hello, Dog! 🐶 🐕
```


```
You look like a ...
Dachshund
```

In [44]:
```
Creature_breed('images/img7.jpg')
```

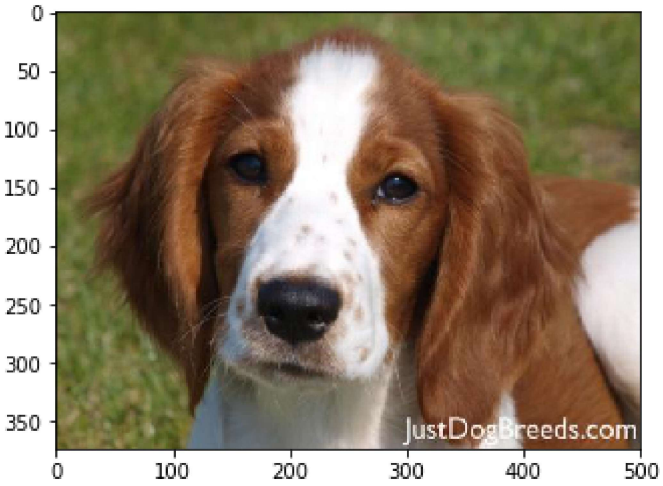
```
Hello, human! ✋😎 <|o_o|>
```



```
You look like a ...
Chinese_crested
```

In [45]:
```
Creature_breed('images/img3.jpg')
```

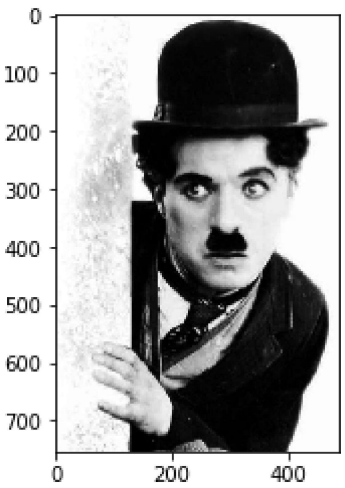
```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
Welsh_springer_spaniel
```

In [46]:
```
Creature_breed('images/img5.jpg')
```

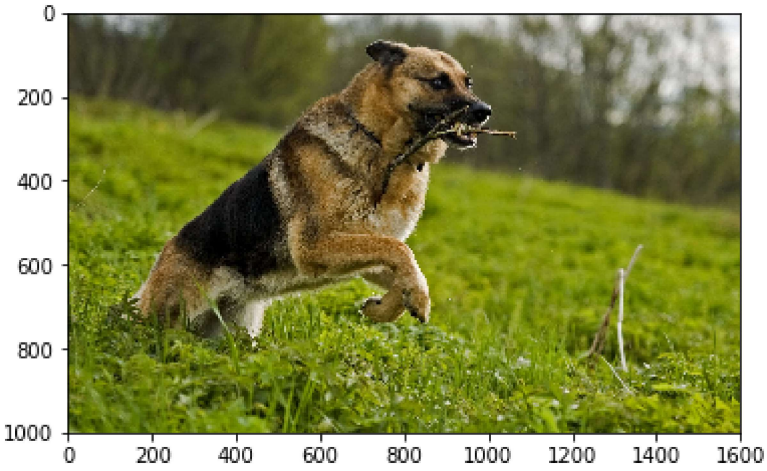
```
Hello, human! ✋😎 <|o_o|>
```




```
You look like a ...
Cane_corso
```

In [47]:
```
Creature_breed('images/img6.jpg')
```

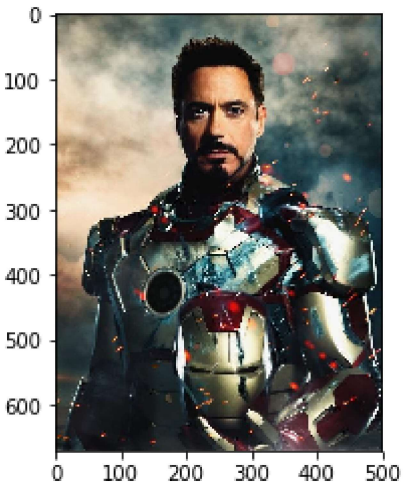
```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
German_shepherd_dog
```

In [48]:
```
Creature_breed('images/img9.jpg')
```

```
Hello, human! ✋😎 <|o_o|>
```




```
You look like a ...
Dalmatian
```

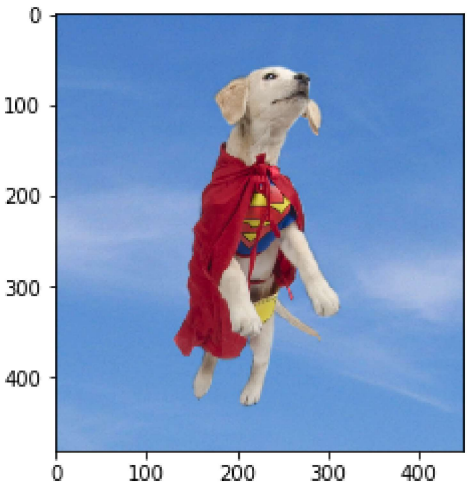
In [49]:
```
Creature_breed('images/img10.jpg')
```

```
No human face or dog detected 🐶 👽 😈

 OMG! it is an Alien! 😀 😁 😂
```


```
You look like a ...
Xoloitzcuintli
```

In [50]:
```
Creature_breed('images/img11.jpg')
```

```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
Dalmatian
```

In [51]:
```
Creature_breed('images/American_water_spaniel_00648.jpg')
```

```
Hello, Dog! 🐶 🐕
```


```
You look like a ...
American_water_spaniel
```

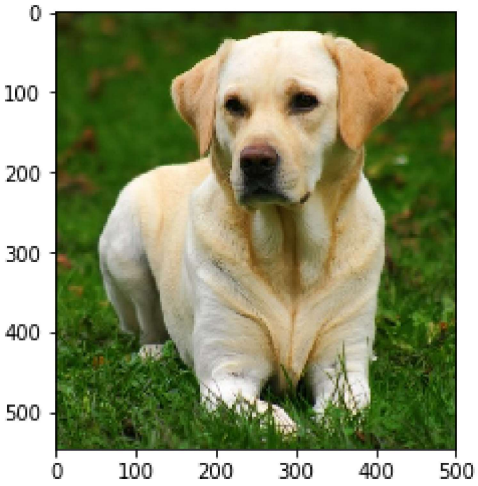
In [52]:
```
Creature_breed('images/Labrador_retriever_06449.jpg')
```

```
Hello, Dog! 🐶 🐕
```


```
You look like a ...
Labrador_retriever
```

In [53]:
```
Creature_breed('images/Labrador_retriever_06457.jpg')
```

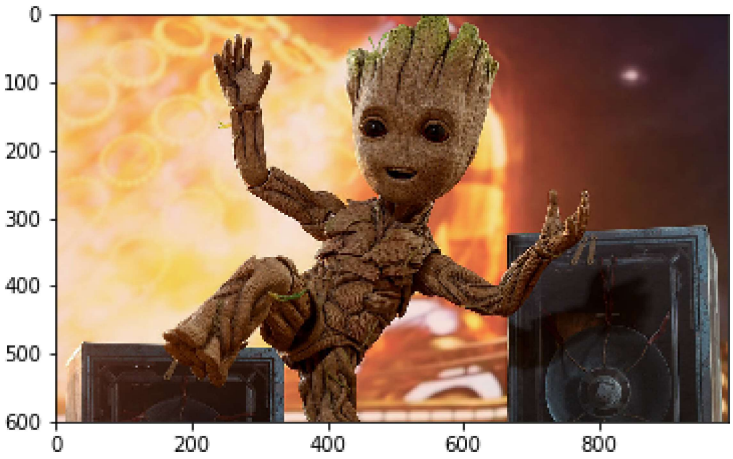
```
Hello, Dog! 🐶 🐕
```




```
You look like a ...
Labrador_retriever
```

In [54]:
```
Creature_breed('images/img13.jpg')
```

```
No human face or dog detected 🐶 👽 😈

 OMG! it is an Alien! 😀 😁 😂
```




```
You look like a ...
German_pinscher
```